Make it or Break it: Mining Anomalies in Linux Kbuild

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 has long been an interesting subject of study in terms of its source code. Recently, it has also been studied in terms of its variability since the Linux kernel can be configured to include or omit certain features according to the user's selection. These features are defined in the Kconfig files included in the Linux kernel code. Several articles study both the source code and Kconfig files to ensure variability is correctly implemented and to detect anomalies. However, these studies ignore the Makefiles which are another important component that controls the variability of the Linux kernel. The Makefiles are responsible for specifying what actually gets compiled and built into the final kernel. With over 1,300 Makefiles, more than 35,000 source code files, and over 10,000 Kconfig features, inconsistencies and anomalies are inevitable. In this paper, we explore the Linux's Makefiles (Kbuild) to detect anomalies. We develop three rules to identify anomalies in the Makefiles. Using these rules, we detect 89 anomalies in the latest release of the Linux kernel (2.6.38.6). We also perform a longitudinal analysis to study the evolution of Kbuild anomalies over time, and the solutions implemented to correct them. Our results show that many of the anomalies we detect are eventually corrected in future releases. This work is a first attempt at exploring the consistency of the variability implemented in Kbuild with the rest of the kernel. Such work opens the door for automatic anomaly detection in build systems which can save developers time in the future.**

I. Introduction

The Linux kernel is one of the most studied open source software repositories. The Linux kernel consists of source code files, configuration files, and Makefiles [5]. However, in the past it has mostly been studied in terms of its source code (e.g., [6]). Recently, the Linux kernel has also been studied in terms of its variability and configurability [19], [21]. Variability studies on the Linux kernel have focused on the Kernel configuration (Kconfig) files and how they are related to the source code implementation. However, most of these variability studies have generally overlooked the build system despite its importance.

Studies show that build files tend to be ignored although they are important artifacts of a software system [18]. In a survey conducted by Kumfert et al. [9], the authors found that maintaining build systems accounts for 12% of the overhead time in the software development process which suggests that build systems play an important role. The build system (which usually consists of Makefiles and/or build scripts) is the ultimate controller of what ends up in the final software product. In the case of Linux, the `make` tool, originally proposed by Feldman [5], is used to build the kernel. Any mistake in the Makefiles which are read by `make` can lead to undesired behavior in the final kernel. In this sense, the build system can either *make* the system work as desired, or *break* the system. A challenge here is that some anomalies or inconsistencies do not cause the build process itself to break, but cause subtle inconsistencies in the behavior of the system that can remain undetected for a long time. Automatic discovery of anomalies in the build system is therefore important.

The Linux kernel is an interesting subject of study in terms of its Kernel build (Kbuild) system since Kbuild cooperates with the other artifacts in the system (source code and Kconfig) to implement variability in the kernel. Linux allows the user to configure the kernel such that an instance of the kernel is compiled with the options the user has selected. There are three spaces involved in this process: the *implementation space* (source code), the *configuration space* (Kconfig files), and the *compilation space* (Makefiles). In version 2.6.38.6 of the Linux kernel, the implementation space size is over 35,000 files (including .c, .h and .S files), the configuration space consists of over 10,000 features [22], and the compilation space consists of more than 1,300 Makefiles. The combination of constraints in these three spaces controls the resulting built kernel. With the large sizes of these spaces, one can almost be sure that there will be mistakes and conflicts which can result in undesired behavior in the kernel. Tartler et al. [22] study part of this problem by presenting the constraints in the implementation space and the configuration space as propositional formulas, and then using these formulas to identify code blocks which do not satisfy them. They do not, however, consider the compilation space in their analysis.

Considering the compilation space is essential because if a source code file is not referenced in the Makefiles then the feature(s) it is implementing will never be included in the final kernel. It is, therefore important to make sure that the Makefiles are correctly setup to use all the necessary source files and to only compile them if their respective feature option is chosen. In this sense, we define two families of anomalies in the Kbuild system: *syntactic* and *semantic*. Syntactic anomalies refer to incorrect setup of the Makefiles. For example, code files that are not properly used, usage of undefined Makefile variables or Kconfig features etc. Semantic

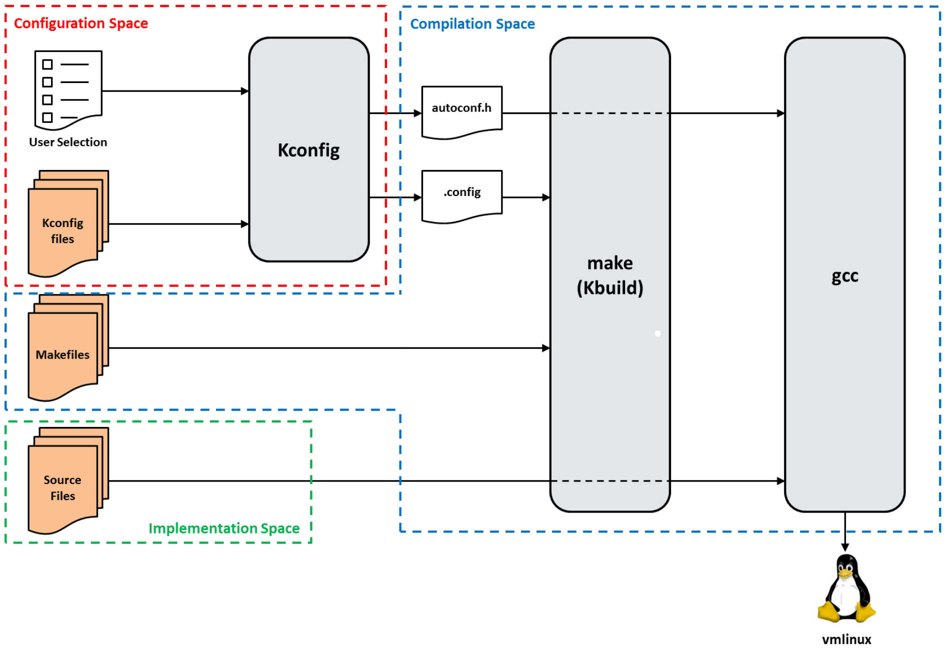

Fig. 1. Linux kernel build process

Kbuild anomalies, on the other hand, are logical contradictions with any of the other two spaces. These are more challenging to capture. In both cases, the build system does not actually break, but undesired behavior may result.

In this paper, we focus on the first part of the problem which is the syntactic anomalies. We examine the Kbuild system to check that the Linux kernel variability is correctly implemented in it, and that all the necessary source code files are correctly referenced. Based on the existing Kbuild documentation [15], as well as our examination of the Makefiles, we define three rules that must be satisfied for Kbuild to be correctly setup.

We use these rules to study the Kbuild anomalies in a snapshot of the Linux kernel (specifically release 2.6.38.6) as well as to perform a longitudinal study to analyze the evolution of the Kbuild anomalies over time, and how these anomalies get fixed. We discover a total of 89 Kbuild anomalies in release 2.6.38.6 of the Linux kernel. We use 14 releases of the Linux kernel in our longitudinal study, and find that many of the anomalies we discover get fixed in future releases. We study these fixes, and identify the most common solutions to the types of anomalies we find. The main contributions of this paper are as follows:

- We *examine* the Linux Kbuild system, in terms of its variability implementation.
- We develop *three rules* that can be used to *detect anomalies* in the Kbuild system.
- We *run* these rules on the Linux kernel code release 2.6.38.6, and *detect* 89 anomalies.
- We *explore* the *evolution* of Kbuild anomalies over time, and how they get fixed by running a longitudinal study on 14 previous kernel releases.

The rest of this paper is organized as follows. Section II explains the process of building the Linux kernel. Section III describes the variability of the Linux kernel by providing background information about Kconfig and Kbuild. Section IV describes how the three spaces of the Linux kernel must be consistent, and Section V presents the rules we define for consistency to be met with respect to the compilation space. Section VI describes the results of our snapshot study, and Section VII presents the results from our longitudinal study. Section VIII discusses possible threats to validity. In Section IX, we present related work. Section X suggests possible future work, and Section XI then concludes this paper.

II. Building the Linux Kernel

Figure 1 illustrates the process of building the Linux kernel. The Linux build process relies on three different artifacts in the Linux tree: the source code files, the Kconfig files, and the Makefiles. These three artifacts are shown on the left in the figure.

The first step in building the Linux kernel is configuring it. This is done using various tools that read the Kconfig files and display them to the user in a menu format. These tools include `menuconfig`, `xconfig`, and `qconfig`. These tools are not shown on the figure, but are part of the Kconfig box. After the user configures the kernel through Kconfig, two files are produced: the `.config` file used by Kbuild, and the `autoconf.h` file used by the gcc compiler. These files contain the user's selection of features. Although they contain the same information, they have slightly different formats since they are used in different places. Entries in the `.config` file used by Kbuild have the format shown in Listing 1. This format defines environment variables that will be used in the

Makefiles to control which files get compiled. A feature that is selected will be defined as an environment variable with the value 'y'. More details on this are given in Section III-B.

```
CONFIG_SOUND=y
CONFIG_SOUND_OSS_CORE=y
```

Listing 1. Examples of .config entries

On the other hand, the same information will be present in the `autoconf.h` file, but with the format shown in Listing 2. This is essentially a header file that defines some preprocessor directives that control selective compilation by the gcc compiler. Here, selected features have the value '1'.

```
#define CONFIG_SOUND 1
#define CONFIG_SOUND_OSS_CORE 1
```

Listing 2. Examples of autoconf.h entries

Based on the features defined in the `.config` file, the Makefiles instruct the gcc to compile and link certain files into the final kernel image, `vmlinux`, shown at the bottom of Figure 1. The Makefiles also force the header file `autconf.h` to be included in all source code compilation. Accordingly, when gcc compiles these source files, the features defined in the header file `autoconf.h` determine which parts of the code are actually compiled based on the preprocessor directives (`#ifdefs`).

Figure 1 shows how the three artifacts as well as the processes and tools fit into three spaces that ultimately control variability of the kernel. These are the configuration space (consisting of Kconfig files), the implementation space (consisting of source code files), and the compilation space (consisting of Makefiles). The following sections provide more details about Kconfig and Kbuild. For details about how variability is implemented in the source code through preprocessor directives, we refer the reader to Sincero et al. [21].

III. Variability in the Linux Kernel

A. *Kconfig*

This section provides a brief description of Kconfig that is necessary to understand the rest of this paper. More details can be found in other work [12], [13], [19]. Kconfig files are responsible for describing the various features that are part of the Linux kernel. They specify possible configuration options and their interdependencies. Each feature has a `config` entry in a Kconfig file.

Listing 3 shows examples of Kconfig entries. The first entry indicates that there is a feature called `USB` of type `bool`. This means that this feature is either selected (its value is `y`) or not selected (its value is `n`). It also shows that the `USB` feature has a default value of `y` which means it is selected by default.

```
config USB
    bool
    default y

config USB_DEVICEFS
    bool "USB device filesystem"
    depends on USB

config USB_SERIAL_CYPRESS_M8
    tristate "USB Cypress M8 USB Serial Driver"
```

Listing 3. Kconfig example

The second config entry in Listing 3 shows another feature called `USB_DEVICEFS` also of type `bool`. The quoted text after the config type is the prompt message that will appear to the user during the configuration process. Additionally, `USB_DEVICEFS` also depends on `USB`. This means that it cannot be selected unless `USB` is also selected. The last entry shows another feature `USB_SERIAL_CYPRESS_M8` of type tristate. This means that apart from the values `y` and `n`, this feature can also take on the value `m` which means it will be compiled as a loadable module.

Each CPU architecture (e.g., x86, arm etc.) has its own Kconfig file which defines architecture specific features, and also includes Kconfig files from other directories (such as drivers, memory management etc.). Note that whenever Kconfig features are referenced in the C 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`.

B. *Kbuild*

As shown in Figure 1, the Kbuild system (Kernel Build system) uses a collection of Makefiles in the Linux source code tree which are responsible for compiling and linking the source code into the final Linux image, `vmlinux`. Note that there are also files called Kbuild files that are formatted similar to Makefiles. Both files contain syntax understandable by `make`. For simplicity purposes, we will use the term Makefiles throughout the rest of the paper to refer to both types of files.

The Linux source code is stored in many directories, where each directory has a collection of source files responsible for a certain feature or a certain subsystem. There is usually a Makefile in each of those directories, and each Makefile is mainly responsible for the files in its directory [10].

There is a Makefile in the root of the Linux source tree (the top Makefile). The top Makefile is responsible for setting up all the environment variables that are needed during the build process, e.g., the CPU architecture being built, compiler options etc. The top Makefile reads the `.config` file which comes from the Linux kernel configuration process and which specifies all the features that have been selected by the user.

The top Makefile includes the Makefile of the CPU architecture being built, and then each architecture's Makefile is responsible for selecting the relevant files that should be compiled.

```
1    obj-y    +=  bar.o
2    obj-y    +=  dir1/
3    obj-$(CONFIG_FOOBAR) +=  foobar.o
4    obj-$(CONFIG_FOO)     +=  foo.o
5    foo-y    :=  foo1.o  foo2.o
```

Listing 4. Example Makefile

Listing 4 shows a sample Makefile. This Makefile shows the major entries in a Makefile that contribute to the variability of the kernel. There are certain conventions followed in Kbuild that are enforced through *implicit rules* [17] defined in the Makefiles. 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. All the files in this list (i.e., the value of the `obj-y` variable) are then compiled and built into a `built-in.o` object for that directory. At the end of the build process, all the `built-in.o` objects in the directories are linked into the final `vmlinux` product.

obj-y entries. In Line 1 of Listing 4, `bar.o` is added to the list of files that will be compiled into the `built-in.o` for that directory. Kbuild's implicit rules state that each .o file should be compiled from a corresponding .c file. In this case, `bar.c` will be compiled into `bar.o`. Directories can also be added to the `obj-y` list. For example, Line 2 adds the directory `dir1` to the this list. This means that a sub-Makefile is being invoked. This simply tells `make` to visit the `dir1` directory, but does not tell it what to do there. The Makefile found in the `dir1` directory is the one that will specify which files from that directory will be compiled. Recursive calls to make (through descending into sub directories) can share variables if the parent process exports them into the environment [17]. Thus, all the Makefiles can see the values of the entries in the `.config` file.

Feature dependent entries. The format `obj-$(CONFIG_FEATURE)` is used to allow the compilation of certain files only if their respective features are chosen. If `CONFIG_FEATURE` is set to be `y`, its value will be included as part of the `obj-y` list and will thus be built into the kernel as part of the `built-in.o` file. If it is set to be `m`, it will be part of `obj-m`, and it will be compiled as a loadable module. If this feature is not selected, then it will not be defined in the `.config` file, and thus the variable name will be `obj-` which is ignored. For example, in Line 3 of Listing 4, `foobar.o` will only be included in the list of files to be compiled if the feature FOOBAR is selected.

Composite objects. Sometimes, a combination of several source files implement one feature, and we want to group them into one list for convenience. Then, we can include this whole list if the corresponding feature is chosen. These are called composite objects. Lines 4 and 5 in Listing 4 show an example of a composite object, and its usage. If `CONFIG_FOO` is `y` or `m`, the compiler will go ahead and build the `foo.o` object. In this particular example, there is no `foo.c` file in the directory. Therefore, Kbuild checks if a `foo-y` or `foo-objs` variable is defined. This notation is enforced through Kbuild's implicit rules, and these composite objects also serve as lists of files and directories. In this example, `foo1.c` and `foo2.c` will be compiled into the `foo.o` object (Line 5), and will then be included in the `obj-y` or `obj-m` list according to the value of `CONFIG_FOO` (Line 4).

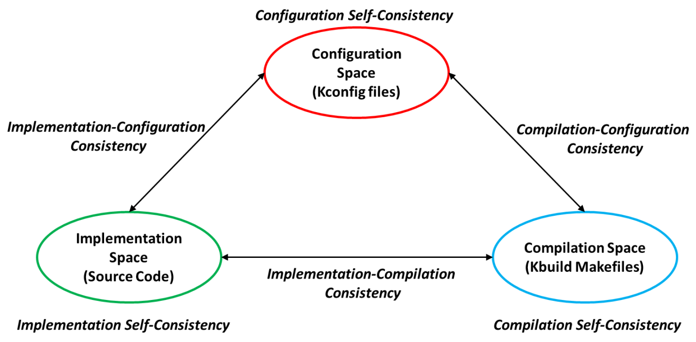
Configuration Self-Consistency
Configuration Space (Kconfig files)
Implementation-Configuration Consistency
Compilation-Configuration Consistency
Implementation Space (Source Code)
Implementation-Compilation Consistency
Compilation Space (Kbuild Makefiles)
Implementation Self-Consistency
Compilation Self-Consistency

Fig. 2. Consistency among the three spaces

Executable Files: Special cases include executable files that `make` creates on the host machine for use during compilation [15]. These are part of the `hostprogs-y` variable. In these cases, a `fileName.c` is indicated as part of the executable files if it appears in an entry such as `hostprogs-y += fileName` or `hostprogs-$(CONFIG_FEATURE) += fileName`. There are other special cases, and other lists in Kbuild (such as `head-y`, `lib-y`, etc.). However, we only describe the major parts of Kbuild here. For more information, we refer the reader to the Kbuild Documentation [15].

IV. Consistency among the Three Spaces

Figure 2 shows the three spaces responsible for implementing variability in the Linux kernel: the configuration space, the implementation space, and the compilation space. In order for variability to be correctly implemented, each space must be self-consistent, and the three spaces must be consistent with each other. This yields six consistency checks. First we have three self-consistencies of the three spaces: *configuration self-consistency*, *implementation self-consistency*, and *compilation self-consistency*. Then, we have the cross consistencies between the three spaces: *implementation-configuration consistency*, *compilation-configuration consistency*, and *implementation-compilation consistency*.

Let us consider each of these one at a time. Configuration self-consistency means that all Kconfig entries follow the Kconfig syntax, that all references to a Kconfig entry have a corresponding definition somewhere, and that dependencies within one configuration definition are not contradictory. Implementation self-consistency means that the constraints imposed by preprocessor directives in the source code are not self-contradictory. For example, a code block cannot depend on a feature being defined and undefined at the same time. Compilation self-consistency means that all composite objects defined in a Makefile are actually used.

In terms of cross consistency between the spaces, implementation-configuration consistency means that any

Anomaly	Description
File Not Used	A .c file exists in the directory but is not used in the Makefile of that directory.
Feature Not Defined	A .c file is referenced in the Makefile, and its presence is conditioned on a Kconfig feature being defined. However, this feature is not defined in any of the Kconfig files.
Variable Not Used	A .c file is referenced in the Makefile as part of a composite variable definition, but this variable is never used.

TABLE I
TYPES OF KBUILD ANOMALIES

Kconfig feature used in the source code must be defined in the Kconfig files. Additionally, the dependency constraints imposed in the Kconfig files must be consistent with those in the code. Tartler et al. [22] explore the consistency between these two spaces to identify dead and undead code blocks.

Compilation-configuration consistency requires that all Kconfig features used in the Makefiles are defined in the Kconfig files. Finally, the implementation-compilation consistency means that all source files in the compilation space should be used in the Makefiles in order to be compiled.

V. THE COMPILATION SPACE

The focus of this paper is the compilation space and the Makefiles. Accordingly, we focus on three of the consistency checks shown in Figure 2 that are relevant to the compilation space: compilation self-consistency, compilation-configuration consistency, and implementation-compilation consistency. Violating any of these three consistency checks results in a Kbuild anomaly.

In this section, we present the rules we use to perform these consistency checks, and the anomalies that result from violating any of these rules. We define three consistency rules whose violation results in three types of anomalies: *File Not Used anomaly*, *Feature Not Defined anomaly*, and *Variable Not Used anomaly*. These rules, which ensure that compilation self-consistency, compilation-configuration consistency, and implementation-compilation consistency are achieved, are as follows:

1) *Rule 1:* In each directory, every `fileName.c` file should have a corresponding `fileName.o` entry in the Makefile of that directory.
2) *Rule 2:* If `fileName.o` is dependent (directly or indirectly through a composite object) on some CONFIG feature, then there should be a corresponding config entry defined for this feature in one of the Kconfig files.
3) *Rule 3:* If `fileName.o` is part of a composite object definition, then we must make sure that this composite object gets used somewhere in that same Makefile.

File Not Used anomaly (Violation of Rule 1). We first check that every .c file in the Linux source tree is used in its corresponding directory's Makefile. If this is not the case, then this is a violation of the implementation-compilation consistency. This violation will result in a File Not Used anomaly.

Feature Not Defined anomaly (Violation of Rule 2). Suppose we find that the file we are looking for is part of an entry that is conditioned on a configuration feature. For example, `obj-$(CONFIG_FEATURE) += fileName.o`. Then, `FEATURE` must be defined as a `config` entry in one of the Kconfig files. If this is not the case, then this is a violation of the compilation-configuration consistency which leads to a Feature Not Defined anomaly.

Variable Not Used anomaly (Violation of Rule 3). Suppose we find that the file we are looking for is part of a composite object entry (see Section III-B). For example, `variableName-y += fileName.o`. In this case, `variableName` is a composite object, and `variableName.o` must be used somewhere in the Makefile. If it is not used in the Makefile, this is a violation of the compilation self-consistency, and will result in a Variable Not Used anomaly.

Table I summarizes the three types of anomalies we detect. We believe that the Variable Not Used and Feature Not Defined anomalies indicate errors. For both these types of anomalies, it seems clear that the developer intended for this file to be included in the build process, but due to some error has not setup things correctly, and forgot to use the composite variable in the first case or forgot to define the Kconfig feature in the second case. However, the File Not Referenced anomaly may not necessarily indicate an error. Instead, it can be caused by a developer intentionally ignoring to use a file because this file is no longer needed, but is kept in the directory for reference purposes.

Figure 3 illustrates the consistency rules through a simple example. In this example, there are four files in the directory: `bar.c`, `foo1.c`, `foo2.c`, and `foobar.c`, as drawn in the left part of the figure. We look at the Makefile of that directory, and check that all the .c files in the directory have corresponding .o usages in the Makefile. For example, `bar.c` has a corresponding entry `bar.o` in the Makefile, and it is part of `obj-y` so we know it will be compiled as part of the kernel (see Section III-B). In the case of `foo1.c` and `foo2.c`, we can find corresponding .o entries, but they are part of a composite object `foo-y`. We, therefore, need to look for the usage of `foo.o` somewhere, and indeed we find a usage of it in an `obj-y` entry. Finally, for `foobar.c`, its corresponding `foobar.o` entry depends on a Kconfig feature, `CONFIG_FOOBAR`. We, therefore, need to confirm that there is indeed a `config` definition for `FOOBAR` in one of the Kconfig files in the Linux source tree. If any of the arrows shown in the diagram cannot be found, then an anomaly exists.

VI. RESULTS PART 1: FINDING KBUILD ANOMALIES IN A LINUX KERNEL SNAPSHOT

In this section, we describe finding anomalies in a particular release (a snapshot) of Linux. We first describe the details of the release we used, and then present the results.

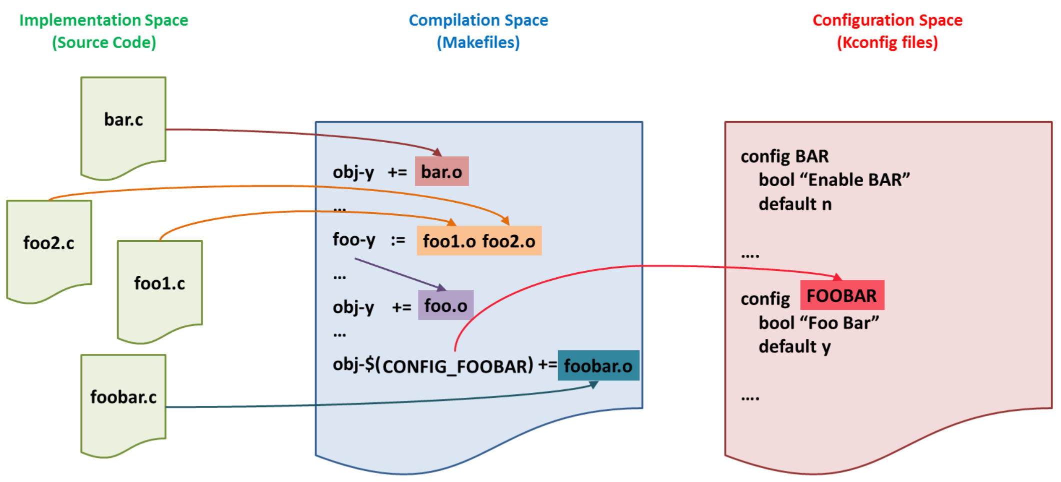

Fig. 3. Illustration of compilation space related consistency rules

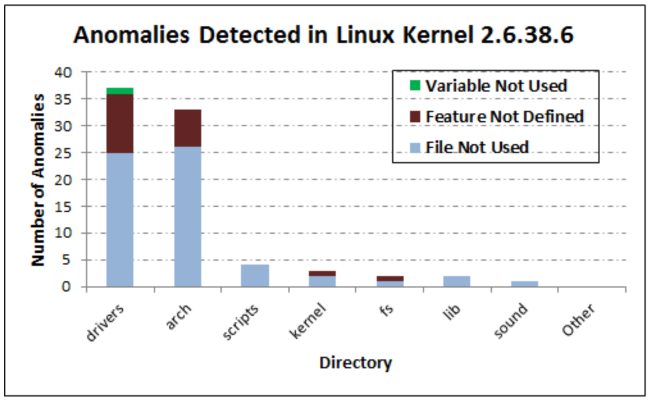

Fig. 4. Kbuild anomalies detected in release 2.6.38.6 of the Linux kernel

A. Experiment Setup

We used the three rules described in Section V to detect anomalies in the entire Linux source tree, specifically release 2.6.38.6. At the time we started this work, this was the latest kernel release. Using the latest release allows us to identify how many Kbuild anomalies currently exist in Linux. Release 2.6.38.6 consists of 2,249 directories, 15,680 .c files, 1,371 Makefiles, and over 10,000 Kconfig features. Most of the directories containing source code have a Makefile that is responsible for that directory. Directories that do not contain source code files do not contain Makefiles. In rare cases (67 directories out of 2,249), a directory which contains source code does not contain a Makefile. In these cases, the files in this directory are referenced in the Makefile of the parent directory. This does not seem like good practice, but these are cases where the directory only has a few files, and developers may feel it is easier to do it that way instead of creating another Makefile for that directory.

B. Results

The anomaly detection on the entire 2.6.38.6 Linux source tree took only 1.95 minutes on a Core i7 2.67 GHz with 8GB RAM. However, our technique can also be applied on a per directory basis. That is, a developer can detect anomalies only in the directory they are responsible for. In that case, the analysis will just run in a few seconds, and the developer does not have to wait for the full analysis of the whole source tree. This will be convenient for developers to use before committing their work.

Figure 4 summarizes the anomalies detected, grouped by the major directories in the Linux kernel. The figure shows that most of the anomalies were in the `drivers` and `arch` directory. This is consistent with various related work on the Linux kernel which find that the `drivers` directory contains many inconsistencies, errors, and clones. Examples include the work done by Tartler et al. [22] on inconsistencies between the implementation and configuration spaces, the work done by Li et al. on copy-paste bugs [11], and the work by Jiang and Hassan on source code clones [7]. We now explain the results we found in each of the three anomaly categories described in Section V, and shown in Figure 4.

File Not Used

We found a total of 61 File Not Used anomalies, i.e., 61 .c files not mentioned in any of the Makefiles. The majority of these files were in the `arch` and `drivers` directories. Detecting the right number of files that have not been used in the Makefiles was challenging at first. When we started, we discovered over 300 files in this category. We later realized that Linux does not follow the usual practice of not #include-ing .c files. Therefore, before reporting a file as not used, we had to first check if it is included in any other .c file which does not have an anomaly. If it is, then we do not report this file as having an anomaly since it will actually be indirectly included in the `built-in.o` file through the `#include` directive.

We further investigated how often this scenario (including .c files) occurs. There are a total of 563 .c files that include other .c files. The same file can be included in multiple files. On average, a .c file that is included by another .c file gets included by 3 other .c files. By considering included .c files, we avoided

Directory	Number of files
drivers/	377
sound/	100
arch/	60
lib/	12
kernel/	6
fs/	3
scripts/	1
other	4
Total	563

TABLE II
NUMBER OF .C FILES INCLUDING OTHER .C FILES BY DIRECTORY (SORTED IN DESCENDING ORDER)

erroneously reporting 198 .c files as not used. These files are not explicitly used in the Makefiles, but they are included in at least one other .c file that is used in the Makefile. Most files including other .c files include files from the same directory they are in. The included .c files usually contain code for extended functionality that may be used by more than one file. For example, the `perf_event.c` file in the `arm/kernel/` directory includes the `perf_event_v6.c` file to add power management specific implementations.

Table II shows the division of files that include other .c files among the different directories. The `drivers` directory has the most of these cases. Following that is the sound directory which is more or less similar to the drivers in nature. This indicates that the development pattern used for drivers (and sound) might be different from other parts of the kernel in that there is a lot of shared code.

While examining the files reported as not used, we also found several files that seem to be left behind for reference purposes or for ongoing maintenance. For example, some of the files reported as not referenced included `old_checksum.c`, `dummy.c`, and `test.c`. Their names indicate that they are probably there for test purposes or as a copy of some previously existing code.

Variable Not Used

We found that the Variable Not Used anomaly is rare. In release 2.6.38.6, we found one such case. This case suggests that mistakes of this nature, though rare, can happen. In the directory `arch/cris/arch-v32/mach-fs/`, the filename `vcs_hook.c` is found in the following line of the Makefile
`bj-$(CONFIG_ETRAX_VCS_SIM) += vcs_hook.o`.
According to our rules, this means that there should be a variable called `bj` used somewhere in the Makefile. However, no such occurrence was found, and so it was reported as a Variable Not Used anomaly. However, on closer inspection, this looks like a typo where the line was intended to be `obj-$(CONFIG_ETRAX_VCS_SIM) += vcs_hook.o` (i.e., the o was forgotten). Detecting this category of anomalies can help catch such spelling mistakes.

We submitted a patch for this, and it was accepted to be released in the next version of the kernel. It is surprising that this has remained undetected since 2007 (according to the developer we communicated with). This indicates that these types of anomalies may be hard to detect manually especially when they do not break the system. However, they may cause some functionality to be missing. In this particular case, the code in the file `vcs_hook.c` was only used for development purposes, and is not actually used in the "real world".

Feature Not Defined

We found a total of 27 Feature Not Defined anomalies, i.e., 27 features that were used in the Makefiles, but were not defined in any of the Kconfig files. This means that in 27 different cases, there is some code that is expected to compile when a certain feature is selected, but this code is actually never compiled because the feature is not defined

It was also interesting that one of these undefined features, `CPU_S3C2400`, appeared in a `default` and `depends on` clause in a Kconfig file, but we could find no definition for it. When there is no definition for a feature in Kconfig, then this feature can never be selected. Additionally, any other feature depending on it in any way will not be visible to the user for selection in the configuration process, and will thus, in turn, never be itself selected as well. This means that some of the intended variability for this feature as well as all features depending on it can never actually be used.

VII. RESULTS PART 2: FINDING KBUILD ANOMALIES IN LONGITUDINAL STUDY

In this section, we analyze anomalies across a sequence of releases of Linux in a longitudinal study. We first explain the releases we used in our longitudinal study, and then present the results we found.

A. Experiment Overview

The snapshot results in the previous section analyzed Kbuild anomalies in a recent release. Our next step is to study whether these anomalies persistently appear in the system, and whether they get resolved or not. We, therefore, run our analysis over a sequence of releases of Linux to observe this behavior.

In order to study the evolution of Kbuild anomalies, we perform a longitudinal study across the last 14 main releases of the Linux kernel. These releases span the period from July 17th, 2008 to May 19th, 2011 (approximately 3 years). The time between releases is approximately two to three months. Performing a longitudinal study provides us more insight into Kbuild anomalies in two ways. First, it gives an indication of the seriousness of the detected anomalies. If developers invest time in fixing the detected anomalies between releases, then this suggests that these anomalies are important. Second, it helps us understand how often these anomalies occur. If they occur frequently between releases, then this indicates the need for an automatic Kbuild anomaly detection mechanism.

B. Results

Figure 5 shows the evolution of the number of Kbuild anomalies over time. These are the main releases in the period between July 2008 and May 2011. We omit the Variable

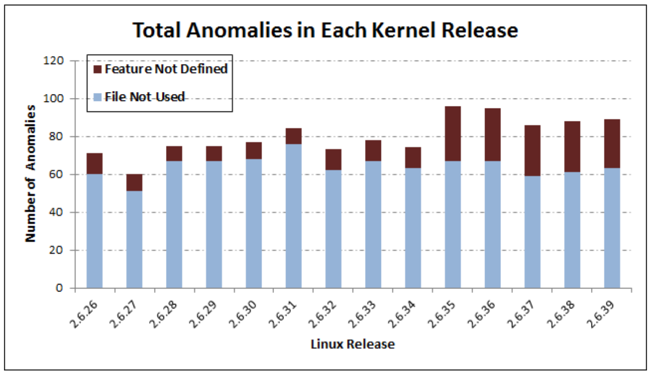

Fig. 5. Evolution of Kbuild anomalies over previous Linux releases

Not Used anomaly from the figure for better visualization since there were only two instances of this type of anomaly throughout all the releases studied. The figure shows that previous kernel releases also contained Kbuild anomalies. We can see that the File Not Used anomaly occurs more frequently than the Feature Not Defined anomaly.

Given that previous releases also contained Kbuild anomalies, we further investigate these anomalies to check the number of introduced and fixed anomalies over time. Figure 6 shows the number of anomalies fixed in each release as opposed to those introduced. Additionally, Figure 6 breaks down the number of introduced and fixed anomalies by type. The fact that several anomalies have been fixed suggests that these anomalies were producing undesired behavior and were worth fixing. The fact that new anomalies are still introduced indicates the need for automatic error detection in the Kbuild system. In order to understand the significance of these anomalies further, we look at the types of fixes in detail.

In Figure 6, we can see that the File Not Used anomaly (in light blue) has the most fixes (left columns). This makes sense since the highest number of anomalies are in that category (see Figure 5). In Figure 6, we can also see that there is a long bar showing many introduced anomalies in release 2.6.35. We can see from the breakdown that most of these introduced anomalies were of type Feature Not Defined. Looking closer at these anomalies, we found that all of the Feature Not Defined anomalies introduced in that release (2.6.35) were caused by the introduction of a new driver, `msm`, that is part of the `drivers/staging` directory. The `staging` directory contains driver code that has still not been accepted as part of the main Linux source tree. Thus, it makes sense that this directory would cause many anomalies. We believe that automatically discovering these anomalies can help developers have their drivers code accepted more quickly into the main Linux source tree.

In order to understand how developers address the anomalies we discovered, and how serious they are considered, we looked at the solution implemented for each fixed anomaly.

File Not Used anomaly. We found that the File Not Used anomaly represented on average 80% of the total anomalies in each release. However, only an average of 9% of the File Not Used anomalies in one release are fixed in the next release. This suggests that although there are many anomalies of this type, most of these anomalies are not particularly urgent since only 9% of them actually get addressed in the next release. Nonetheless, we cannot conclude that the File Not Used anomaly is insignificant. In the releases we examined, there was a total of 108 distinct File Not Used anomalies. By the last release we examined (2.6.39), 78 of these anomalies were actually addressed (i.e., approximately 74%). This shows that although this type of anomaly does not get immediately fixed, they generally eventually get addressed.

We found that there are three ways developers address the File Not Used anomaly. They (1) remove that file from the source tree (occurs 73% of the time), (2) add an entry for it in the Makefile (occurs 22% of the time), or (3) include it in another .c file that is used in the Makefile (occurs 5% of the time). Since removing the file from the source tree seems to be the most common solution, this suggests that most of the File Not Used anomalies are indeed caused by lax maintenance where code that is no longer needed is still left in the source tree.

Feature Not Defined anomaly. We found that the Feature Not Defined anomaly represented on average 16% of the total anomalies in each release. On average, 12% of the Feature Not Defined anomalies in one release were addressed in the next release. This suggests that although there are less anomalies of type Feature Not Defined, a higher percentage of them actually get fixed immediately (i.e., within one release) when compared to the File Not Used anomaly. We found that there are five ways in which the Feature Not Defined anomalies are fixed: (1) a `config` definition for that feature is added to one of the Kconfig files (occurs 37% of the time), (2) the file that depended on this feature has been removed so this feature is no longer used in the Makefile (occurs 45% of the time), (3) the file is moved to another directory which essentially moves the anomaly from one directory to another (occurs 5% of the time), (4) the file that depended on that feature no longer depends on any feature so the feature is no longer used in the Makefile (occurs 3% of the time), or (5) the file now depends on a different feature which has a proper definition (occurs 10% of the time). Since developers add a definition for the Kconfig feature being used 37% of the time, this suggests that this anomaly was indeed an error that prevented required variability from being achieved.

The results from the longitudinal study we performed confirm that developers invest time in fixing Kbuild anomalies. Additionally, they show that new anomalies are introduced in each release. This strongly suggests the need for automatic anomaly detection.

VIII. Threats to Validity

All of the anomalies detected by our tool in release 2.6.38.6 have been reviewed by hand through a grep-based search. For the File Not Referenced anomalies, all the files reported were confirmed as not used anywhere through the grep-based search. For the Feature Not Defined anomaly, we also

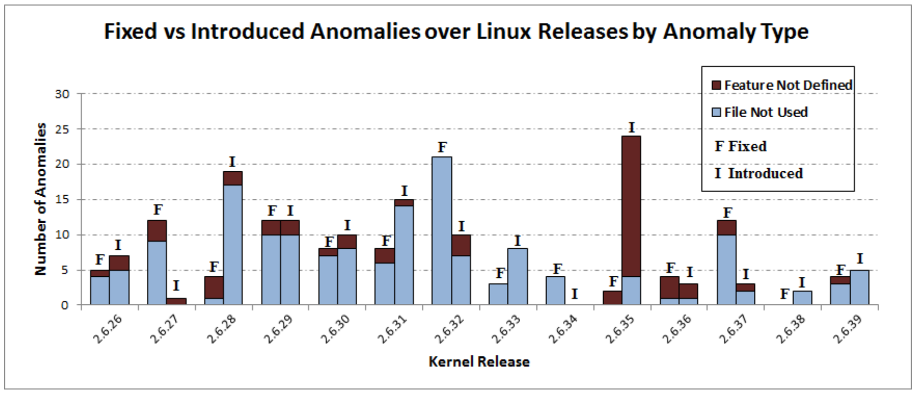

Fig. 6. Breakdown of fixed and introduced anomalies by type over Linux releases. For each release, the left column shows fixed anomalies and the right column shows introduced anomalies.

performed a grep-based search, and confirmed that there was no `config` definition for any of these features in any Kconfig file. Accordingly, we believe that our results contain a low rate of false positives (possibly zero).

On the other hand, our analysis might have missed some anomalies (i.e., false negatives). Currently, we do not consider features used in conditional blocks in the Makefiles (e.g., `ifdef CONFIG_FEATURE`). Some of the features used in such conditions may not be properly defined in the Kconfig files, but will not be reported in our current analysis.

Since this work is a first attempt at discovering anomalies in the Kbuild system, we believe that more consistency rules that take additional aspects into consideration can be developed as more research focuses on this problem. However, at this stage, we believe that a more conservative reporting is better than overwhelming the developer with many results which may contain false positives. Therefore, we do not attempt to generalize our findings in terms of the type of anomalies or how they evolve to other systems. Such generalization requires investigation of additional systems which we plan to do in the future. Once we have examined additional systems, we can achieve external validity for our results.

IX. Related Work

Miller [16] addresses the concerns about the slow performance of recursive `make` by providing guidelines of how to correctly setup the Makefiles to obtain a more efficient build. Jørgensen [8] also provides rules to ensure the correctness and completeness of Makefiles in general. The rules ensure that targets in Makefiles are correctly setup without circular dependencies such that incremental recompilation will produce the same result every time. While these papers and other work (e.g., [1], [3]) focus on the correctness, efficiency, and automatic generation of build scripts in general, our work focuses on how build systems play a role in variability, specifically in Linux's build system (Kbuild).

Adams et al. [2] study the evolution of the Linux build system, and observe that its growth is correlated to the source code growth. However, such studies only correlate the rate of evolution of the two spaces, but do not examine the details of how they actually cooperate together to provide the final product. Additionally, McIntosh et al. [14] find that although build systems account for a small percentage of the files in a project, they have a comparable churn rate to source code which suggests that they are also likely to have defects.

Day [4] presents some shell scripts that look for undefined and unused Kconfig features. However, these scripts do not find the File Not Used and Variable Not Used anomalies. Additionally, our work provides a more systematic approach to finding anomalies in the Kbuild system by defining rules for consistency among the three spaces. Apart from identifying the anomalies, our work also analyzes their frequency of occurrence, and how they get fixed.

There has been recent research examining Kconfig and the Linux variability in general. Lotufo et al. [12], [13] studies the evolution of the Linux kernel variability model. The various versions of the Kconfig are compared in terms of their complexity which is determined through three measures: size, cohesion, and depth.

Sincero et al. [20] along with Tartler et al. [23] show that there are problems that arise because parts of the kernel configurations are kept in different places. They only look at the configuration space (the Kconfig files), and the implementation space (the actual C code), but do not consider the compilation space. They define rules that check certain referential and semantic conditions to ensure consistency between Kconfig files and source code files. They use propositional logic to encode the constraints of each space [22]. Code blocks that cannot satisfy the constraints are identified as dead or undead.

The work presented in our paper is distinguished from related work in the literature in that we focus on the correctness of Kbuild from a variability perspective, and how it relates to other artifacts in the Linux kernel. We identify types of Kbuild anomalies that affect the desired variability in the kernel, and study how they evolve over time as well as how they get resolved.

X. Future Work

A build system using `make` can be quite complex due to the use of implicit rules and many conditional variables. The

Kbuild system makes full usage of these features. Therefore, there may be anomalies that arise in the Kbuild system from incorrect setup of environment variables or compilation options. Such anomalies also affect the final software product and are worth investigating. Additionally, we plan to explore other build systems which also contribute to the variability of their final software product.

In this paper, we attempted to show the significance of the anomalies we detected and how they got addressed through the longitudinal study. We plan to investigate this further by looking at additional information from the developers' commit comments when addressing these anomalies. Additionally, looking at user reported bugs that have been caused by the anomalies we detected can also provide further insight of their importance.

As an initial attempt at clarifying the internals of the Kbuild system, and how variability is implemented in it, this paper only focuses on syntax anomalies. The results of this work can later be expanded to produce a recommender system that developers can use while editing the Kbuild system to automatically detect anomalies for them, and to guide them through the corrections.

XI. Conclusions

This work is our first attempt at exploring the correctness of variability implementation in the Linux Kbuild, which has been largely ignored in previous work. Build systems in general are important to study since they control what goes into the final product. Any mistakes in them can alter the behavior of the software system, and can remain undetected for a long time. In this work, we studied the Makefiles in the Kbuild system, and developed three rules for detecting anomalies in them that may affect the desired variability of the Linux kernel. Our rules include finding code files (.c files) that have not been used in the Makefiles (File Not Used anomaly), composite variables that are never used (Variable Not Used anomaly), and Kconfig features that are used in the Makefiles but are not defined in any Kconfig file (Feature Not Defined anomaly).

We first ran our analysis on a snapshot of the Linux kernel, specifically release 2.8.38.6, and discovered 61 missing files, 1 unused composite variable (which turned out to be a typo in the Makefile and has been acknowledged by the developers), and 27 undefined Kconfig features. Additionally, we performed a longitudinal study of the Kbuild anomalies over 14 Linux releases, and evaluated the seriousness of these anomalies as well as how they get fixed. We found that many of the anomalies we detected get fixed in later releases which indicates that they previously caused erroneous behavior.

We believe that these results are a first step towards ensuring the correctness of the variability implementation in the Kbuild system. In the future, additional rules can be developed to discover more anomalies. Such work can be the basis of tools for developers to automatically detect anomalies and suggest solutions for them.

References

[1] GNU Autoconf. Available at http://www.gnu.org/software/autoconf/.

[2] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter. The evolution of the Linux build system. *Electronic Communications of the EASST*, 8(0), 2008.

[3] G. Ammons. Grexmk: speeding up scripted builds. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, WODA '06, pages 81–87, New York, NY, USA, 2006. ACM.

[4] R. Day. Kernel Cleanup. Technical report, 2009. Available at http://www.crashcourse.ca/wiki/index.php/Kernel_cleanup.

[5] S. Feldman. Make–A program for maintaining computer programs. *Software: Practice and experience*, 9(4):255–265, 1979.

[6] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceerdings of IEEE International Conference on Software Maintenance*, Los Alamitos, CA, USA, 2000.

[7] Z. M. Jiang and A. Hassan. A framework for studying clones in large software systems. In *SCAM 2007: Proceedings of Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 203 –212, 2007.

[8] N. Jørgensen. Safeness of make-based incremental recompilation. In *FME 2002:Formal Methods–Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 126–145. Springer Berlin / Heidelberg, 2002.

[9] G. Kumfert and T. Epperly. Software in the DOE: The Hidden Overhead of "The Build". Technical report, Lawrence Livermore National Lab., 2002.

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

[11] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

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

[13] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the linux kernel variability model. In *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 136–150. Springer Berlin / Heidelberg, 2010.

[14] S. McIntosh, B. Adams, T. Nguyen, Y. Kamei, and A. Hassan. An Empirical Study of Build Maintenance Effort. In *ICSE 2011: Proceedings of the 33rd Intl Conf. on Software Engineering. ACM Press*, 2011.

[15] S. R. Michael Elizabeth Chastain, Kai Germaschewski and J. Engelhardt. Linux Kernel Makefiles. Technical report, 2011. Available at /Documentation/Kbuild/makefiles.txt.

[16] P. Miller. Recursive make considered harmful. In *AUUGN 1998: Australian Unix User Group Newsletter*, page 19(1):1425. 1998.

[17] R. M. Richard M. Stallman and P. D. Smith. The GNU Make Manual. 2010. Available at http://www.gnu.org/software/make/manual/.

[18] G. Robles, J. Gonzalez-Barahona, and J. Merelo. Beyond source code: The importance of other artifacts in software development (a case study). *Journal of Systems and Software*, 79(9):1233–1248, 2006.

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

[20] J. Sincero, R. Tartler, C. Egger, W. Schröder-Preikschat, and D. Lohmann. Facing the Linux 8000 Feature Nightmare. In *EUROSYS 2010: Proceedings of the European Conference on Computer Systems.*

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

[22] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature Consistency in Compile-Time Configurable System Software. In E. C. of ACM SIGOPS, editor, *EuroSys '11: Proceedings of the EuroSys 2011 Conference* , 2011.

[23] R. Tartler, J. Sincero, W. Schroder-Preikschat, and D. Lohmann. Dead or alive: finding zombie features in the linux kernel. In *Proceedings of the First International Workshop on Feature-Oriented Software Development*, pages 81–86. ACM, 2009.