An Empirical Study of Configuration Mismatches in Linux

Sascha El-Sharkawy  
University of Hildesheim,  
Institute of Computer Science  
Universitätsplatz 1  
31141 Hildesheim, Germany  
elscha@sse.uni-hildesheim.de

Adam Krafczyk  
University of Hildesheim,  
Institute of Computer Science  
Universitätsplatz 1  
31141 Hildesheim, Germany  
adam@sse.uni-hildesheim.de

Klaus Schmid  
University of Hildesheim,  
Institute of Computer Science  
Universitätsplatz 1  
31141 Hildesheim, Germany  
schmid@sse.uni-hildesheim.de

ABSTRACT
Ideally the variability of a product line is represented completely and correctly by its variability model. However, in practice additional variability is often represented on the level of the build system or in the code. Such a situation may lead to inconsistencies, where the actually realized variability does not fully correspond to the one described by the variability model. In this paper we focus on configuration mismatches, i.e., cases where the effective variability differs from the variability as it is represented by the variability model. While previous research has already shown that these situations still exist even today in well-analyzed product lines like Linux, so far it was unclear under what circumstances such issues occur in reality. In particular, it is open what types of configuration mismatches occur and how severe they are. Here, our contribution is to close this gap by presenting a detailed manual analysis of 80 configuration mismatches in the Linux 4.4.1 kernel and assess their criticality. We identify various categories of configuration issues and show that about two-thirds of the configuration mismatches may actually lead to kernel misconfigurations.

CCS CONCEPTS
•Software and its engineering →Software verification;  
Software product lines;  
•Social and professional topics →Software maintenance;

KEYWORDS
Software product lines, variability modeling, configuration mismatches, Linux, Kconfig, static analysis, empirical software engineering

ACM Reference format:
Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2017. An Empirical Study of Configuration Mismatches in Linux. In Proceedings of SPLC ’17, Sevilla, Spain, September 25–29, 2017, 10 pages.  
DOI: http://dx.doi.org/10.1145/3106195.3106208

1 INTRODUCTION
In software product line engineering, dependency management, which is part of variability modeling, is not trivial to get right, but fundamental to identify the valid configurations. The complexity of defining the variability and their dependencies to the implementation consistently can be illustrated by the fact that, for instance, Linux version 4.4.1 has more than 15,500 configurable variables to control the conditional inclusion of over 40,000 source files with approximate 102,000 preprocessor-dependent code blocks [13]. Thus, the variability model must be carefully maintained to reflect the implementation as otherwise the configuration process yields defective products. In recent years, much effort was spent in research to detect conflicts between the variability model and implementation artifacts. In this paper, we use the technique of feature effects [22] to extract constraints from code and build artifacts and check if they are covered by the variability model. If the variability model does not cover these feature effect constraints, the configuration process becomes further restricted through these feature effects in an invisible manner. We call this divergence between modeled and implemented dependencies, configuration mismatches [12]. We present an empirical study of those divergences to better understand their sources and to assess their criticality. While the basic identification of mismatches was automated, the assessment was done manually as it required the interpretation of the situation.

The identified mismatches point to situations where the implementation of the product line introduces additional dependencies not covered by the variability model. In Linux, these dependencies do not necessarily lead to compilation errors but bear the risk of deriving the wrong product without any notification. Similar problems can be expected for other product lines using preprocessor-based instantiation, a widespread technique in industrial product lines [14]. In this study we answer the following research questions:

RQ1 What is the severity of configuration mismatches?  
RQ2 What are the technical characteristics of variables, which cause configuration mismatches?  
RQ3 How do configuration mismatches differ conceptually?  
RQ4 Are all three spaces: variability model, code artifacts, and build space always involved in a configuration mismatch? If not, what is their relation?

While earlier studies presented technical approaches for identifying mismatches, they did not provide an understanding of their causes, as they never performed a manual analysis. Here, our contribution is to provide an empirical analysis contributing to the understanding of these configuration mismatches. As a result, we identified four main types of configuration mismatches. We discuss each type based on examples taken from our Linux-analysis and identify the severity of the different mismatches. The severity ranges from rather unproblematic mismatches, e.g., in cases where variabilities were only introduced for structuring purposes to cases with mismatches of high severity, where the resulting system may
be functionally different from the expected one. We could also identify that 66% of the mismatches must be regarded as severe.

The remainder of this paper is structured as follows. In the next section we report related work. Section 3 introduces our object of investigation: Linux. We explain the technical background, which is necessary to understand our findings. In Section 4, we introduce our experimental setup. In Section 5, we present the results of our manual analysis while we discuss our findings in Section 6. Section 7 discusses threats to validity, before we conclude and outline future work in Section 8.

2 RELATED WORK

In this paper, we compare the variability information of code assets with the information in the variability model to detect mismatches in the dependency management. We then analyze the findings to present potential reasons for these configuration mismatches. This is done based on the Linux kernel as an use case for a complex real world software product line. We identified two categories of papers, which are relevant to this work: General papers about defect analysis of variability and papers analyzing the characteristics of Linux. The remainder is organized according to these two categories.

Defect analysis. In recent years, much effort was spent to detect defects related to software product line development, like (un-)dead features [1], (un-)dead code [26], configuration coverage [25], or variability type checking and liveness analysis [20]. Also Thüm et al. [27] surveyed general analysis strategies for software product lines. Here, we identify and analyze configuration mismatches contrary to the identification of permanent unconfigurable blocks or conditional parsing errors.

In [12], we presented already an efficient algorithm to identify a specific kind of configuration mismatches; missing nesting dependencies. In this paper, we use the technique of feature effects [21, 22] to allow also the detection of configuration mismatches in cross-tree constraints. While this paper focuses the manual inspection of configuration mismatches to identify real problems.

Analyzing the characteristics of Linux. The kernel maintainers developed their own solutions for variability management (Kconfig) and instantiation (Kbuild), which we introduce in the next section. These two techniques have been analyzed in many scientific studies. In Section 4, we present our approach for detecting configuration mismatches, which reuses the findings of the work presented here.

Kconfig was developed as a textual variability language to manage the variability of Linux [16]. While textual variability languages from a scientific background have already been extensively compared [9], we currently only know of a comparison of eCos and Kconfig [5]. Other work analyzes the capabilities of Kconfig in isolation. A prerequisite for many analysis is the translation of Kconfig models to a logical representation like propositional logic. This translation is not a trivial task, since the semantics are rather unclear because of many corner cases. In [10, 11] we analyzed the quality of tools regarding the translation of Kconfig models. These results formed the basis for the selection of an appropriate translation tool in Section 4.2.

Kbuild [17] is the build process of Linux and responsible for the mapping of the configuration to the code files. It makes use of more than 70 % of all Kconfig variables [13]. This makes the analysis of Kbuild an important part for any comprehensive analysis of the variability of Linux. Two strategies exist for the analysis of Kbuild, which is in principle turing-complete since it is realized with MAKE. Dietrich et al. [8] query the build system to probe which files are controlled by which variables of the variability model. They state that this concept is very robust as it works without manual intervention for all architectures on all versions of Linux, but needs approximately 90 minutes to analyze one architecture due the amount of probing steps. Nadi et al. [23, 24] as well as Berger et al. [2, 3] use text-based analysis of all build scripts. A comparison of all three implementations show that the the tools by Dietrich et al. and Berger et al. have the highest accuracy [8].

3 CONTEXT

Configurable systems, like Linux, are usually divided into problem space and solution space [7]. The problem space describes the range of supported products without any implementation details, i.e., the variability model. The solution space comprises the artifacts and their instantiation rules. Below, we briefly introduce the relevant parts of Linux related to the remainder of this paper.

3.1 Problem Space

In Linux the problem space is modeled with Kconfig [11, 16]. Kconfig is a textual language, which was developed specifically for Linux and managing its variability. Although it never became a standalone project, it was also used in several other projects [4]. We describe only the most important variability management concepts relevant to explain our findings in Section 5. This is done based on the example given in Listing 1.

The Linux kernel supports multiple CPU architectures. Version 4.4.1 of the kernel supports already 31 different architectures. Each of these architectures has its own Kconfig files, which serve as start point. These architecture-specific models specify architecture-specific variables, settings, and constraints. Most of the Kconfig model is architecture independent and included by the architecture-specific start files. However, the architecture-specific start files require that each architecture is analyzed in isolation.

Config options are the main entities of the language and can be seen as variables. These config options can be of type tristate, bool, string, hex, or int. All but the first are known from typical programming languages and have the same semantics. Tristate config options encode the following alternatives: n (the related feature will not be part of the resulting system), y (the related

| Listing 1: Example of a Kconfig file (indentation not necessary, only used for better readability). |
| --- |
| **config** BOOLEAN_VAR1 |
| bool "Visible Boolean" |
| **select** BOOLEAN_VAR2 |
| **config** TRISTATE_VAR |
| tristate "Visible Tristate" |
| depends on BOOLEAN_VAR |
| default m |
| **config** BOOLEAN_VAR2 |
| bool |
| default y if TRISTATE_VAR |
As most attributes, this attribute can not be made conditional (Line 7).

For a variability variable \( \nu \) the presence conditions it holds, that is if and only if the following formula is satisfiable:

\[
\nu \in PC \iff (\forall \nu \in PC : PC[\nu \leftarrow False] \lor PC[\nu \leftarrow True])
\]

(1)

4 EXPERIMENTAL SET-UP

This section describes our approach of identifying configuration mismatches. We define configuration mismatches and explain how they can be detected in a software product line, then we describe our toolchain for applying this concept to the Linux kernel.

4.1 Conceptual Foundation

Congruence mismatches point to a divergence between configuration information in the variability model and implemented variability of (code) assets [12]. More precisely, this means that the solution space further restricts user-defined configurations through unmodeled dependencies. For the detection of configuration mismatches, we use the approach of feature effects as described in [22], which expresses under which condition a variable has an impact on the product derivation. In their evaluation, Nadi et al. checked whether the identified feature effects are also contained in the variability model. They recognized that approximately 4% of the identified feature effects are not part of the variability model, that is what we define as a configuration mismatch [12]. However, they did not apply a manual analysis of the identified configuration mismatches. We reproduce their experimental set-up and apply a manual analysis of the identified configuration mismatches. In this section, we give an introduction to the concept of feature effect identification as defined by Nadi et al. [22].

A feature effect (FE\( \nu \)) for a variable \( \nu \) of the variability model is a combination of all presence conditions, which make use of \( \nu \). Based on [22], we use the term presence condition (PC) to refer to a propositional expression over Kconfig variables that determines when a certain code artifact is compiled. This comprises very fine-grained #ifdef-blocks surrounding a single line in a C-file, but also the definition of conditional compilation of complete C-files in Makefiles. For this reason, we use for the identification of feature effects the presence conditions from code files as well as from Kbuild.

For a variability variable \( \nu \) and \( PC_{\nu} \), the presence conditions it is used in, the feature effect is defined as [22]\(^1\):

\[
FE_\nu := \bigvee_{PC \in PC_{\nu}} PC[\nu \leftarrow True] \lor PC[\nu \leftarrow False]
\]

(1)

The approach of Formula 1 is based on propositional logic and, thus, does not support tristate logic directly. For this reason, tristate variables are translated into two variables and one constraint. The constraint is used to allow only the same configurations in the translated formulas as in the original variability model. The translation into 2 distinct variables makes it necessary to compute the feature effects for both translated variables separately.

Ideally, the variability model allows the configuration of \( \nu \) only if it affects the final product. \( \nu \rightarrow FE_\nu \) specifies under which condition \((FE_\nu)\), the selection of \( \nu \) has an effect on the final product. It can then be checked whether the variability model \((VM)\) implies (i.e. contains) this constraint:

\[
\models VM \rightarrow (\nu \rightarrow FE_\nu)
\]

(2)

A configuration mismatch is detected, if this condition does not hold, that is if and only if the following formula is satisfiable:

\[
\text{SAT}(VM \land \nu \land \neg FE_\nu)
\]

(3)

\(^1\)\(PC[\nu \leftarrow C]\) means replacing each occurrence of \( \nu \) in \( PC \) with constant \( C \)
We opted for using KbuildMiner since we had good experiences with it. This allows us to use SAT-solvers for the translation of the Kconfig model to propositional logic, as described at the beginning of this section. For each configuration mismatch found, the tool outputs the problematic variable and minimal partial configurations, under which the problem is reproducible.

In this case, every valid solution (considering the variability model) of $\neg FE_y$ is then a partial configuration in which selecting or deselecting $y$ has no effect on the final product. These partial configurations formed the basis for our manual analysis in Section 5, because they are useful for tracking down and deciding whether there is an underlying problem that needs to be fixed. The distinct calculation of feature effects for the permanent selection ($y$) and the selection as module ($m$) for tristate variables facilitates an independent analysis for both selections. However, it also doubles the number of reports in case that both assignments are problematic.

4.2 Toolchain

For our experiment, we analyzed the x86 architecture of the Linux kernel version 4.4.1. Figure 1 presents the toolchain, which we used for our analysis. For each location of variability (Kconfig, Kbuild, and C preprocessor), we used different, already available extraction tools. The outputs of the three tools are then combined and analyzed in a tool, that is specifically written for this task.

For the translation of the Kconfig model to propositional logic, we chose KconfigReader [19]. This allows us to use SAT-solvers later on. In contrast to the other available tools for the same task, KconfigReader produces the most reliable results [11]. For our analysis, we used the latest commit from 01.07.2016\(^2\).

For the extraction of presence conditions from Kbuild we chose KBuildMiner [2]. This tool computes for each file under which condition it is compiled, regarding a specific architecture. The result is then converted into a format, which is compatible with the other tools. This conversion also considers the information from Kconfig to differentiate between tristate and boolean variables. It is important to note, that KBuildMiner parses the Kbuild files only heuristically, so it is not completely reliable. However, a comparison shows that KBuildMiner is as reliable as the other available tools [8]. We opted for using KBuildMiner since we had good experiences with this tool in the past and since it has a significantly lower runtime than Golem (under a minute compared to about 90 minutes) while having comparable reliability. For our analysis, we used the latest commit from 01.07.2016\(^3\).

For extracting the variability from the C source code we chose Pilztaker [18], a tool based on Undertaker 1.6.1 [29] (commit from 02.06.2016\(^4\)). This tool extracts each conditional compilation directive of the C preprocessor, that is found inside the C source files of the Linux kernel. This hierarchical structure is then translated into presence conditions. However, Pilztaker can not handle header files, that are included inside of C source files. It also can not find any usage of variability that is not inside #ifdef-blocks (for example, in macro expansions). To bypass these limitations, we filter out each variability variable that is used in headers or outside of #ifdef blocks; these variables are not considered in any further analysis. We opted for this tool in contrast to other available tools, since including a macro-aware parsing lead to too complex constraints for the manual analysis of the identified results. Also, its limitations can be easily detected and affected variables can be avoided, in order to not produce false positives later on.

The mismatch detector [13] takes these three inputs and searches for configuration mismatches. To do that, the file presence conditions are combined with the code presence conditions. A mapping of each Kconfig variable, together with the presence condition they are used in, is created. The feature effects of each variable are calculated from this mapping, and finally a SAT-solver is applied to detect possible configuration mismatches, as described at the beginning of this section. For each configuration mismatch found, the tool outputs the problematic variable and minimal partial configurations, under which the problem is reproducible.

5 ANALYSIS

Our tooling reported 247 potential configuration mismatches for the x86 architecture of the Linux kernel version 4.4.1. Due to the SAT-based analysis, tristate variables were treated as 2 independent variables. This means that our tooling reported two distinct results, if it detected a configuration mismatch for the selection as a loadable module ($m$) and for the permanent selection ($y$). If only one of these selections lead to a mismatch or if the variable was of type bool, the tooling reported only one result.

In the first step, we excluded 150 results for the manual analysis. This was necessary for the following reasons:

- 89 of them refer to variables, which were also used in header files. We excluded these results to avoid false positive results as we could not reliably calculate feature effects for these variables.

\[^2\text{commit hash 913bd31}\]
\[^3\text{commit hash 00c5b09}\]
\[^4\text{commit hash 40fccc6d}\]
The distribution is as follows (cf. Figure 2): 42 of the results point to situations were the configuring user is able to configure Kconfig variables without any effect. 21 of the results are used to structure the variability model without affecting the final product at all. 11 results point to configuration issues with invisible Kconfig variables. The remaining 6 results are related to variables for modeling platform dependent capabilities.

Each subsection starts with a conceptual description of a specific type of configuration mismatch, followed by a meaningful example, which we took from the Linux kernel version 4.4.1. Based on the presented example, we discuss the criticality of the identified category.

### 5.1 Ineffective Configuration Options

In this section, we discuss a type of configuration mismatches, which is a prime example of the induced problems. Configuration options are defined and need to be considered by the user. Contrary to their description, the selection of those variables does not affect the final product under certain conditions. As a consequence, the configuration process takes longer than needed and may actually lead to different results than those expected as the user may choose a certain option, while the final product does not even contain the functionality. This problem arises if visible variables of the variability model are not constrained in the same manner as the associated code artifacts. In [12] we have already presented an example, which allows a detailed configuration of memory sticks during suspension, even if the user disabled support for suspension before. Below we show another meaningful example: the variable ATH5K_PCI in the wireless network drivers from Atheros.

In Listing 2, we show the relevant excerpt of the Kconfig model for Atheros wireless cards of the 5000 series. If the user selects the variable responsible for the inclusion of the complete driver (ATH5K), it automatically forces the boolean variable ATH5K_PCI to be selected as y, if the ATH25 is deselected (cf. Line 4). The later is a variable of the “MIPS” architecture and, thus, is deselected in all other architectures. However, ATH5K_PCI has also a prompt (cf. Line 8). This allows the manual configuration of this variable, if it was not selected before. As a consequence, the user can configure ATH5K_PCI only if ATH5K is deselected.

Listing 2 shows the relevant part of Kbuild for the compilation of the related C-files. The only usage of ATH5K_PCI inside the solution space is in Line 2. The variable is used to control the conditional compilation of pci.c as part of the ath5k object. The result of

```plaintext
Listing 2: Excerpt of ineffective configuration options (Kconfig).

```

```plaintext
config ATH5K
    tristate "Atheros 5xxx wireless cards support"
    depends on (PCI || ATH25) && MAC80211
    select ATH5K_PCI if !ATH25
    ...

config ATH5K_PCI
    bool "Atheros 5xxx PCI bus support"
    depends on (!ATH25 && & PCI)
    ...
```

The manual analysis of the 80 configuration mismatches revealed 4 different categories, which we describe in detail in the following subsections. These are: Ineffective configuration options, vendor-specific submenus, ignored invisible variables, and capabilities.
The usage of the \textit{(Kbuild)}.

This compilation is statically linked into the kernel or compiled as loadable module depending on the value of \texttt{ATH5K} (cf. Line 3). As a consequence, \texttt{ATH5K\_PCI} is only considered for compilation if \texttt{ATH5K} is selected.

The combination of the modeled and the implemented dependencies result in a configuration mismatch. More precisely, Kconfig facilitates the configuration of \texttt{ATH5K\_PCI} exactly if it has no impact. This is obviously a fault.

This case is obviously related to an incorrect usage of Kconfig, since the relationship between both variables is the same in the problem space and in the solution space (\texttt{ATH5K\_PCI} is used for a detailed configuration of \texttt{ATH5K}). However, without a complete knowledge about all existing \texttt{ATH5K} cards, it remains unclear how to solve this issue. There exist two possible solutions:

- The usage of a conditional \texttt{default} value would be more useful if not all but most of these cards are built for PCI (soft constraint), to facilitate overwriting the default selection.
- The usage of the \texttt{select} constraint is correct if all \texttt{ATH5K} cards are PCI based (hard constraint). However in contrast to the current situation, \texttt{ATH5K\_PCI} should be made invisible to avoid illegal configurations. This is also recommended by the Kconfig language specification, which discourages the selection of visible variables [16].

While this example is obviously related to a missing dependency in the Kconfig model, this is not necessarily the case for all instances of this category. In general, a configuration mismatch of this category may point to a missing constraint of the problem space or to an over-constrained solution space.

5.2 Vendor-Specific Submenus

Some parts of the Kconfig model use additional variables to group product-specific variables of a vendor. These variables can be identified by their name as they usually contain the name of vendor instead of a specific product or functionality. Vendor-specific variables do not directly affect the product derivation. However, these variables must be selected to facilitate the selection of the product-specific variables. Thus, the purpose of these vendor-specific variables is to form a selectable submenu. As a consequence, for each of these vendor-specific variables exists a redundant configuration: the deselected vendor-specific variable and the selected vendor-specific variables without any selected product-specific variables form the same product. The description of the vendor-specific variables points out that this behavior is actually intended as they contain the hint: “Note that the answer to this question doesn’t directly affect the kernel: saying N will just cause the configurator to skip all the questions about <VENDOR> cards. If you say Y, you will be asked for your specific card in the following questions.” Examples exist for Google, Cisco, Qualcomm, and Samsung, but also for other vendors. Below we present a representative example based on the vendor-specific submenu for Ethernet devices of Qualcomm.

Listing 3: Excerpt of ineffective configuration options (Kbuild).

Listing 4: Excerpt of a vendors-specific Submenu (Kconfig).

Listing 5: Excerpt of a vendor-specific submenu (Kbuild).
5.3 Ignored Invisible Variables

This category describes the situation where an invisible variable (without a prompt) is ignored during the instantiation process. These variables have a conditional default value or are selected (without a prompt) is ignored during the instantiation process.

According to our evaluation, these vendor-specific submenus seem not to be a problematic issue, even though they introduce redundant configurations. The description of the vendor-specific variables point out that this behavior is intended by the developer. Thus, these variables can be regarded as abstract features [28].

5.4 Capabilities

This section is similar to the previous one, but differs insofar as the identified results use variables which are used to describe platform-dependent capabilities, e.g., supported features of a CPU architecture. The names of these variables usually are in the form of "<platform>_HAVE/HAS/SUPPORTS/WANTS/...<function>". Below, we present an example of these capabilities based on the Xen hypervisor.

Listing 8 presents an excerpt of the Kconfig model related to the configuration of Xen. Lines 1–7 are specific for the x86 architecture, the lines below are from the common part. The architecture-specific part exists in a similar way also for the other architectures. For instance, the ARM architecture defines the configurable variable XEN analogous, but does not specify the select constraint of Line 5. The selected configuration option (cf. Line 15) is such an invisible variable to model the provided capabilities of the Xen platform: It specifies that the x86-specific implementation of Xen provides a virtual performance monitoring unit. Independent from the architecture-specific settings, the user is able to generally configure monitoring via the XEN_SYS_HYPERVERSOR variable (cf. Line 10).

In Kbuild, the variable XEN_SYS_HYPERVERSOR is used to select the C-file for compilation, which realizes the monitoring feature. This part is architecture independent. Inside the C-file, XEN_HAVE_VPMU is used in preprocessor statements to augment the logging output, if a virtual performance monitoring unit is available in the current platform.

In this category the configuration mismatch appears because additional variables are introduced to represent capabilities. This appears to be an explicit design decision by the people constructing

```
1 config SERIAL_8250_MID
2   tristate "Support for serial ports on Intel MID platforms"
3   depends on SERIAL_8250 & PCI
4   select HSU_DMA if SERIAL_8250_DMA
5   select HSU_DMA_PCI if X86_INTEL_MID

config HSU_DMA_PCI
6   tristate
7   select DMA_ENGINE
8   select DMA_VIRTUAL_CHANNELS

config HSU_DMA
9   tristate
10   select DMA_ENGINE
11   select DMA_VIRTUAL_CHANNELS

Listing 6: Excerpt of ignored invisible variables (Kconfig).
```

```
# drivers/dma/Hsu/Makefile
obj-$(CONFIG_HSU_DMA) += hsu/

# drivers/dma/Hsu_PCI/Makefile
obj-$(CONFIG_HSU_DMA_PCI) += hsu_dma_pci.o

Listing 7: Excerpt of ignored invisible variables (Kbuild).
```
the variability model, hence, we do not regard it as problematic. Rather, this can be seen as an evolution provision. Indeed, such a configuration mismatch need not be present in other architectures.

6 DISCUSSION

In this section we discuss our findings in the relation to our research questions. Figure 4 provides a summary.

6.1 RQ1: Severity of Findings

According to our analysis 53 of 80 (66.25%) analyzed configuration mismatches are real problems in the sense that the final product will have a different functionality than would be expected according to the configuration. This is depicted in Figure 4. Without a deep understanding of the domain knowledge it remains unclear whether these problems are related to an over-constrained solution space or an under-constrained problem space.

The identified problems can be further subdivided into two different categories according to their impact on product derivation:

- 79.25% of the problematic configuration mismatches are related to visible variables (ineffective configuration options). The user must configure options, which have absolutely no impact on the configuration result under certain conditions. Thus, the configuration process takes longer than needed and leads to different results than those expected as the user may choose a certain option, while the final product does not contain the functionality.

- 20.75% are ignored invisible variables. These variables should be used to realize technical settings depending on the current configuration. Instead of proper usage of constraints to ensure a correct selection of those variables, the developers rely on implementation techniques to ensure correct functionality of the derived kernel. As a result, these variables become useless.

6.2 RQ2: Technical Characteristics of Variables

Here, we discuss the characteristics of the variables causing the configuration mismatches. We observed the following distribution (cf. Table 1):

Type: In Kconfig, all variables are typed. While the used tools are capable of handling Strings and numerical variables also in propositional logic [11], we did not identify any configuration mismatches triggered by such variables. This is probably related to the low occurrence of those variables. A study revealed that only 6% of the variables in the variability model of Linux are of those types, while 36% are of type bool and 58% are of type tristate [5].

The 80 results of our analysis are related to 68 variables. 54 variables are of type bool (79.4%) and 14 variables are of type tristate (20.6%). Most of the reported tristate variables had a configuration mismatch for both value assignments m and y. For two of these tristate variables, only the selection as a loadable module (m) was problematic.

Visibility: Another characteristic is that not all defined variables are visible to the end-user (i.e., variables that do not have prompt). These variables can only be configured through their dependencies. Therefore, correct constraint modeling becomes even more important for those variables to be effective. This differentiation is presented in the second layer of Figure 4 as it serves as a basis of our classification of configuration mismatches.

| Type    | No. | Dist. | Visibility | No. | Dist. |
|---------|-----|-------|------------|-----|-------|
| Boolean | 54  | 79.4% | Visible    | 44  | 64.71%|
|         |     |       | Invisible  | 10  | 14.71%|
| Tristate| 14  | 20.6% | Visible    | 12  | 17.65%|
|         |     |       | Invisible  | 2   | 2.94% |
| Total   | 68  | 100%  | Total      | 68  | 100%  |

Table 1: Technical differentiation of involved variables and their distribution.

We can see that Boolean variables are significantly more involved in mismatches than tristate variables. Other types do not occur as mismatches. Also more variables with a prompt lead to configuration mismatches than invisible variables, which is the reason for the high ratio on ineffective configuration options (cf. Section 6.1).
6.3 RQ3: Conceptual Differentiation

In Section 5, we identified 4 categories of configuration mismatches. These categories form the third layer of the tree in Figure 4. Independently of their technical characterization, these 4 categories differ in their impact with respect to the product derivation:

- 27 results are categorized as unproblematic. 21 of them are used to define submenus in Kconfig and 6 are used to model capabilities, which are not needed for certain configurations.
- 53 results point to a problematic divergence between modeled and implemented dependencies, which were already discussed in Section 6.1. These two categories affect different roles during the development: Ineffective configuration options are visible variables, which need to be considered by the configuring user without any impact on the product derivation, while ignored invisible variables affect developers only.

Even if some configuration mismatches seem to be desired by the developer, we see that most of the mismatches are critical and bear the risk of a wrong product derivation.

6.4 RQ4: Involved Spaces

Our analysis revealed that all analyzed configuration mismatches are related to variables, which are either used in code artifacts or in Kbuild, but not in both. This is surprising as approximately 19% of all Kconfig variables are used to configure code and Kbuild artifacts together [13]. However, our result may be not fully comparable as we excluded all variables from our analysis, which are also used in header files. Table 2 presents the absolute and relative distribution of our findings related to code and build files. It can be seen that the distribution for problematic variables is not the same as for unproblematic ones. We do not have a strong explanation for this divergence, hence, we just report it here.

|                | Code | Kbuild | Total |
|----------------|------|--------|-------|
|                | No.  | Dist.  | No.   | Dist. |
| Problematic    | 25   | 47.17% | 28    | 52.83% | 53    | 100% |
| Unproblematic  | 7    | 25.93% | 20    | 74.07% | 27    | 100% |
| Total          | 32   | 40%    | 48    | 60%    | 80    | 100% |

Table 2: Results related to solution space.

7 THREATS TO VALIDITY

In this section, we discuss threats to validity regarding our analysis, which consists of an automatic identification of configuration mismatches and a manual assessment of the results. Beyond this discussion, we made the tool for the detection of configuration mismatches and the raw data of our analysis publicly available [13] to facilitate the reproduction of our analysis and to allow the independent verification of our findings.

Internal validity. The automatic analysis of Linux is not a trivial task. Linux consists of complex artifacts written in several different, decades-old languages [22]. The variability modeling approach consists of many undocumented corner cases, which hamper the automatic analysis [11]. To overcome these issues, we built our toolchain for the automatic identification of configuration mismatches on existing research tools, which have already shown that they perform very well. Then, we applied a manual analysis to all results to exclude false positive reports from further analysis. However, the total number of unidentified configuration mismatches remains unclear as we also had to exclude some variables from our consideration (see construct validity).

All results, which have been confirmed as real configuration mismatches, have been manually assessed by one author. Some samples were verified by a second author to ensure that the results are reproducible and correct. We discussed each of the identified categories to rate its general criticality. However, we omit a further analysis whether the results are related to an over-constrained solution space or to missing constraints of the problem space, because it would require extremely detailed domain knowledge of the implementation of Linux.

External validity. The approach presented in this paper was explicitly designed to analyze Kconfig-based software product lines with C-code. Most of the used tools have already been applied to other Kconfig-based product lines, like BusyBox or uClibc. However, the analysis of these systems require additional effort to adapt the underlying tooling, because the systems differ in how they realize Kconfig. For instance, BusyBox stores artifact-specific variability in the related code artifacts instead of separate configuration files as it is done in Linux [6].

We chose Linux as a case study, because it was already often the object of investigation in scientific research. Further, the developers spent much effort to build a huge infrastructure to test each new change. It monitors several development trees and also the mailing list. The test infrastructure tests changes on more than 100 different kernel configurations and applies statistical tests [15]. For this reason, we expect other systems to contain more configuration issues relative to their size and complexity. Our expectation is underpinned by the analysis of Nadi et al. [22], which revealed that in Linux most of the identified feature effects are also modeled as constraints in the variability model. Only 4% of the identified feature effect conditions are not covered by the variability model and, thus, are configuration mismatches. The relative number of identified configuration mismatches was much higher for the other case studies (21% – 31%).

Construct validity. We used the technique of feature effects [22] to gather constraints from the variability model. This technique requires the knowledge of all presence conditions a variable is used in to reliably calculate its dependencies. Missing presence conditions may result in the calculation of a too restrictive constraint and, thus, in a false positive report. Thus, we omitted all variables from further analysis, for which we could not compute reliable feature effect constraints. These are variables used in header files or which are involved in too complex presence conditions. Even if we excluded such variables in the first step from our analysis, there exist still enough configuration mismatches to obtain meaningful results.

Conclusion validity. The classification of configuration mismatches is a subjective process, which we did based on the manual assessment. The categorization was discussed by all three authors of the paper to make the classification as objective as possible. Independent of this categorization, we presented two conceptual different examples, which demonstrate that configuration mismatches may point to potential kernel misconfigurations.

Answers to the research questions were done based on our experience with Linux and the observations we did. However, the analysis of the involved spaces is mitigated by the fact that we...
excluded variables from our analysis, which are also used in header files. Considering these variables may change the relative distribution of the findings in Table 2.

8 CONCLUSION

In this paper, we identified configuration mismatches in Linux and applied a manual analysis to improve the understanding of their causes. We have shown that the bulk of these configuration mismatches point to real problems (RQ1), which may be the result of missing or defect constraints in the variability model or wrong dependencies in code or build artifacts (RQ4). The identified configuration mismatches can be differentiated technically based on the visibility and the type of the triggering variable (RQ2). The visibility also indicates the main stakeholders, which are affected by the configuration mismatch: Visible configuration mismatches concern the configuring users as they make the configuration process unnecessary complex and may end up in the wrong product. Invisible configuration mismatches result in superuous parts of the variability model and the developers need to encode the dependencies in the solution space. This paper supports a better understanding of the criticality of configuration mismatches in general and informs the development of automated analysis tools to identify them (RQ3).

The analysis of other Kconfig-based product lines and a macro-aware parsing remain future work. The analysis of Nadi et al. [22] has shown that other systems contain much more configuration mismatches than Linux. These systems have not been analyzed as thoroughly as Linux. For this reason, we expect more insights from these systems. In particular we expect deeper insights about their severity. A macro-aware processing makes the manual analysis more complex, which is needed to classify the impact of the configuration mismatch. However, this would also enable the inclusion of variables, which are used in header files and, thus, would make the analysis more comprehensive.

ACKNOWLEDGMENTS

This work is partially supported by the Evoline project, funded by the DFG (German Research Foundation) under the Priority Programme SPP 1593: Design For Future — Managed Software Evolution and by the ITEA3 project REVAmp2, funded by the BMBF (German Ministry of Research and Education) under grant 01IS16042H. Any opinions expressed herein are solely by the authors and not of the DFG or the BMBF.

REFERENCES

[1] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. Information Systems 35, 6 (2010), 615–636.
[2] Thorsten Berger and Christian Kästner. 2016. KBuildMiner. (2016). https://github.com/ckaestne/KBuildMiner Last visited: 03.03.2017.
[3] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Feature-to-code mapping in two large product lines. In Proceedings of the 14th International Conference on Software Product Lines (ICSE’10). Springer-Verlag, Berlin, Heidelberg, 498–499.
[4] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2012. Variability Modeling in the Systems Software Domain. Technical Report GSDLAB-TR 2012-07-06. Generative Software Development Laboratory, University of Waterloo.
[5] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. IEEE Transactions on Software Engineering 39, 12 (2013), 1611–1640.
[6] Busybox – Repository 2014. (2014). https://git.busybox.net/busybox/tree/docs/new-appllet-HOWTO.txt Last visited: 20.02.2017.
[7] Krzysztof Czarnecki and Ulrich W. Eisenacker. 2000. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, New York, NY, USA.
[8] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Peikschat, and Daniel Lohmann. 2012. A robust approach for variability extraction from the Linux build system. In Proceedings of the 16th International Software Product Line Conference (SPLC’12). Vol. 1. ACM, New York, NY, USA, 21–30.
[9] Holger Eichelberger and Klaus Schmid. 2015. Mapping the Design-Space of Textual Variability Modeling Languages: A Refined Analysis. International Journal of Software Tools for Technology Transfer 17, 5 (2015), 559–584.
[10] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the KConfig Semantics and Related Analysis Tools. Technical Report 1/2015, SSE 1/15/E. Institute of Computer Science, University of Hildesheim. Available at https://sse.uni-hildesheim.de/media/b4/informatik/AG_SSE/PDFs/publications/Kconfig/Report.pdf.
[11] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the Kconfig Semantics and Its Analysis Tools. In Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015). ACM, New York, NY, USA, 45–54.
[12] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2016. Mis-matched Configuration Information of Linux. Technical Report 1/2016, SSE 1/16/E. Institute of Computer Science, University of Hildesheim. Available at https://sse.uni-hildesheim.de/media/b4/informatik/AG_SSE/PDFs/publications/El-SharkawyKrafczykSchmid16.pdf.
[13] SSE Group. 2017. Configuration Mismatches. (2017). https://sse.uni-hildesheim.de/en/configuration-mismatches Last visited: 16.05.2017.
[14] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Lefebvre, Martin Becker, and Sven Apel. 2016. Preprocessor-based variability in open-source and industrial software systems: An empirical study. Empirical Software Engineering 21, 2 (2016), 449–482.
[15] Intel. 2017. Linux Kernel Performance. (2017). https://01.org/lkp/documentation/0-day-test-service Last visited: 01.03.2017.
[16] Kconfig Language Specification 2016. (2016). https://www.kernel.org/doc/documentation/kbuild/kconfig-language.txt Last visited: 13.02.2017.
[17] Kconfig Language Specification 2016. (2016). https://www.kernel.org/doc/documentation/kbuild/makefiles.txt Last visited: 14.02.2017.
[18] Adam Krafczyk. 2017. Pilztaeker. (2017). https://github.com/SSE-LinuxAnalysis/pilztaeker Last visited: 01.03.2017.
[19] Christian Kästner. 2016. KConfigReader. (2016). https://github.com/ckaestne/kconfigreader Last visited: 01.03.2017.
[20] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Diirre, and Christian Lengauer. 2013. Scalable analysis of variable software. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, New York, NY, USA, 81–91.
[21] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In Proceedings of the 16th International Conference on Software Engineering (ICSE 2014). ACM, New York, NY, USA, 140–151.
[22] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. IEEE Transactions on Software Engineering, 41, 8 (2015), 820–841.
[23] Sarah Nadi and Ric Holt. 2011. Make it or break it: Mining anomalies from Linux kbuild. In 18th Working Conference on Reverse Engineering (WCRE 2011). IEEE Computer Society, Washington, DC, USA, 315–324.
[24] Sarah Nadi and Ric Holt. 2012. Mining Kbuild to detect variability anomalies in Linux. In 16th European Conference on Software Maintenance and Reengineering (CSMR 2012). IEEE Computer Society, Washington, DC, USA, 107–116.
[25] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Peikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90,000 ifdef Issue. In Proceedings of the 2014 USENIX Annual Technical Conference (USENIX 2014). USENIX Association, Berkeley, CA, USA, 421–432.
[26] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Peikschat. 2011. Feature Consistency in Compile-Time—Configurable System Software: Facing the Linux 10,000 Feature Problem. In Proceedings of the 6th Conference on Computer systems. ACM, New York, NY, USA, 47–60.
[27] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. ACM Computing Surveys (CSUR) 47, Article 4 (2014), 45 pages. Issue 1.
[28] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract features in feature modeling. In Proceedings of the 16th International Software Product Line Conference (SPLC’11). IEEE, Washington, DC, USA, 191–200.
[29] Undertaker 2015. (2015). https://vamos.informatik.uni-erlangen.de/trac/undertaker Last visited: 01.03.2017.

Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid