Understanding Linux Feature Distribution *

Christian Dietrich Reinhard Tartler Wolfgang Schröder-Preikschat Daniel Lohmann
Friedrich-Alexander University Erlangen-Nuremberg, Germany
{dietrich, tartler, wosch, lohmann}@cs.fau.de

Abstract

Managing variability is hard. This applies both to feature modeling itself as well as the maintenance of the corresponding feature implementations which poses additional challenges. Especially in embedded systems and system software that are developed using the tools CPP, GCC and MAKE, feature realizations happen on different levels of abstractions, concepts and implementation languages.

This particularly applies to Linux, which exposes over 11,000 features on over two dozen different architectures. While features are modeled centrally with the KCONFIG tool, feature-code is realized in various source-files and managed by the KBUILD build-system. In this article, we identify and relate levels of variability on which feature-code is implemented. The quantification of variability on the different levels in Linux disproves two common beliefs about the amount of implemented variability.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design; D.2.9 [*Management*]: Software configuration management

General Terms Design, Experimentation, Management

Keywords Configurability, Maintenance, Linux, Kbuild, Static Analysis, VAMOS

1. Introduction

System software typically employs compile-time configuration to tailor the system with respect to a broad range of supported hardware architectures and application domains. A prominent example is the Linux kernel, which provides more than 11,000 configurable features [18].

Technically, static configurability is generally perceived as implemented by means of the C Preprocessor (CPP) [10, 16]; the prospective disadvantages of this approach ("#ifdef hell") have often been criticized [15]. In the case of Linux (2.6.35) they seems to be true: more than 84,000 #ifdef-blocks, spread over 28,000+ source code artifacts, account for this – and these numbers are subject to constant growth; they have practically doubled over the last five years and have already caused hundreds of bugs [18].

Problem Statement

In order to provide a practical means to control more than 11,000 features, the Linux KCONFIG tools allows the user to *configure* his selection with various front-ends. While this provides a central interface for the user for configuring the kernel (the *configuration space*), the feature *implementations* for the selected features are scattered over the whole codebase (*implementation space*).

Due to its sheer size, importance, and source-code availability, Linux has been a first-class evaluation subject for approaches and tools for static analyses in the systems as well as the software engineering communities. The general believe is that "if they got Linux through it, it scales". However, in fact researchers almost never evaluate their tools and approaches with the whole Linux tree. For technical and practical feasibility, they often restrict the scope to a subset of it (such as: #ifdef-induced variability on arch-x86 or arch-x86 with allyesconfig). There is a common belief that they have chosen the most significant part.

In this paper we quantitatively analyze some properties of the implementation of variability in Linux with respect to *common beliefs*:

1. **Most variability is expressed by boolean (or tristate) switches.** The common assumption is that it is sufficient to concentrate on them and to not consider value-type features.
2. **arch-x86 is the largest and allyesconfig selects most features.** Here, the conclusion is often that arch-x86/allyesconfig covers most of the Linux code base.
3. **Variability is mostly implemented with the CPP.** The conclusion is that metrics and analyses regarding Linux variability can be considered as sound even if they focus only on CPP-induced variability.

The remainder of the paper is structured as follows: Section 2 describes how variability is technically implemented in Linux,

* This work was supported by the German Research Council (DFG) under grant no SCHR 603/7-1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MISS'12, March 27, 2012, Potsdam, Germany.
Copyright © 2012 ACM 978-1-4503-1217-2/12/03…$10.00

that is, how a feature selection in KCONFIG eventually results in a compiled kernel image. Section 3 describes and discusses our analysis results regarding the mentioned common beliefs. Section 4 gives an overview on the related work. Section 5 summarizes our work and concludes the paper.

2. Implementation of Static Variability in Linux

This work analyzes the different characteristics of feature selections and implementations that are used for compiling the Linux kernel. While variation points for the same feature may occur in different source files and languages, they remain interlinked because they derive from a common source: the user selection. Therefore, this user selection influences both the fine-grained variability within the source files using the `CPP`, as well as the coarse-grained variability in the build system. A sound analysis of feature implementations therefore requires the correlation of variations points in different languages, which is non-trivial.

2.1 The Hierarchy of Variability

We observe variability in software projects in several forms, locations and granularity. In fact, it can be described as a hierarchy like in Figure 1. At the top-most layer l_0, features are described and their constraints are modeled. This can happen either programmatically (like in GNU/autoconf [3]), or in a declarative way using domain specific languages (DSLs). For instance, the operating-systems eCos [12] and Linux provide sophisticated languages, the configuration description language (CDL) and KCONFIG [5], and configurators, which expose all user-configurable features with various front-ends.

The realization of features and their selection happen for systems written in C and MAKE heterogeneously at different levels. In Linux, a sophisticated build system KBUILD integrates KCONFIG seamlessly with the compilation process. This represents level l_1. Here, the decision what source files to include and what compilation and linker flags to use is heavily influenced by the feature selection from level l_0.

For each file that the build system selects for compilation, the compiler uses the `CPP` to compose header and implementation source files and to select what parts to include in a compilation unit. This fine-grained selection of feature implementations on level l_2 is dominated by the decisions on level l_1: Only features in source files that are actually compiled will end up in the resulting kernel.

This dominance hierarchy continues on the language level l_3 and below: In the resulting compilation unit, `if()` statements decide what feature implementations to use both at compilation-time (by constructs such as `if (HAVE_FEATURE)`, with `HAVE_FEATURE` being a configuration generated `CPP` macro[1] on level l_0) and at run-time. On the linking level l_4, further variation points can be realized in linker scripts; again only for actually compiled compilation units. This hierarchy can even be extended to run-time variability with loadable kernel modules (LKMs) (level l_5). The levels l_4 and l_5 are not (directly) KCONFIG-driven, because the respective tools (e.g., linker, INSMOD) and languages (e.g., linker-scripts) cannot (directly) reference KCONFIG-derived configuration variables (i.e., the `CONFIG_` variables).

[1] This example doesn't work in Linux for purely technical reasons, but is not uncommon at all in other systems such as busybox.

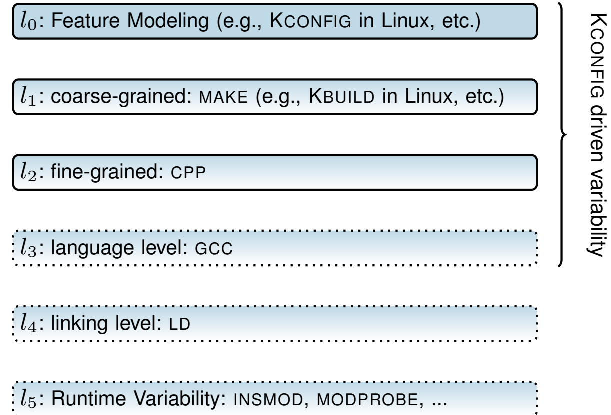

Figure 1. Abstract overview over the dominance hierarchy of variability implementations

In this article, we compare the variability declaration on level l_0 with the implementation levels l_1 and l_2. Higher levels will be covered in future work.

2.2 Fine- and Coarse-Grained Variability

Figure 2 shows how coarse-grained and fine-grained variability points relate to each other and how they are controlled by the used toolchain. In step ❶, the feature selection which is created by the user with KCONFIG, is saved to a file (`.config`) that serves several purposes. First, it is used for storing, loading and interchanging feature selections. Second, the Linux build system KBUILD converts this feature selection into two representations: In Step ❷ the selection is stored in MAKE syntax to the file `auto.conf`. And in Step ❸ it is stored in `CPP` syntax to the file `autoconf.h`. These generated artifacts control the compilation process on different levels.

The representations for feature selections in the KCONFIG definitions lack the `CONFIG_` prefix; it is added to all other representations which creates a (more or less strictly enforced) namespace for KCONFIG controlled symbols. For the `CPP` representation ❸ an additional normalization step is applied for tristate features: In order to represent the ternary logic *compiled into kernel*, *compiled as loadable kernel module* and *disabled*, KCONFIG creates an additional `CPP` variable with an `_MODULE` suffix in `autoconf.h` for each tristate feature. In earlier work, we have analyzed the logical constraints between the code and the KCONFIG definitions for logical and referential defects [18].

The MAKE representation manages the **coarse-grained variability** on a per-file basis. It selects a subset of all source

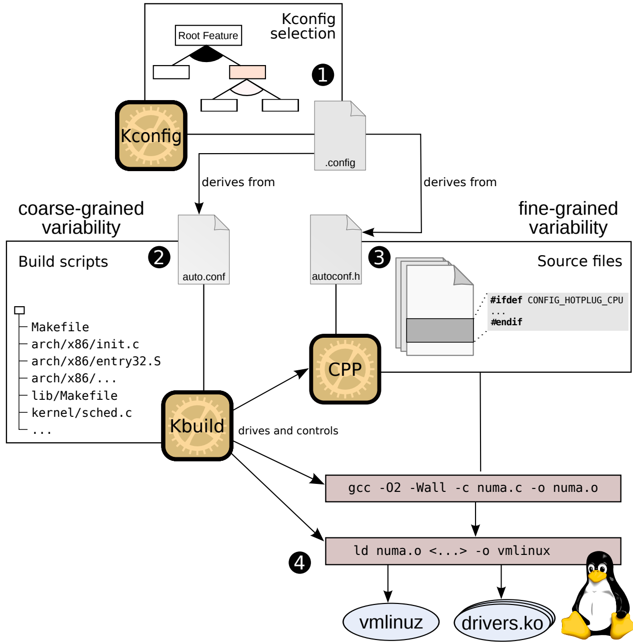

Figure 2. In Linux coarse-grained varibility dominates fine-grained varibility

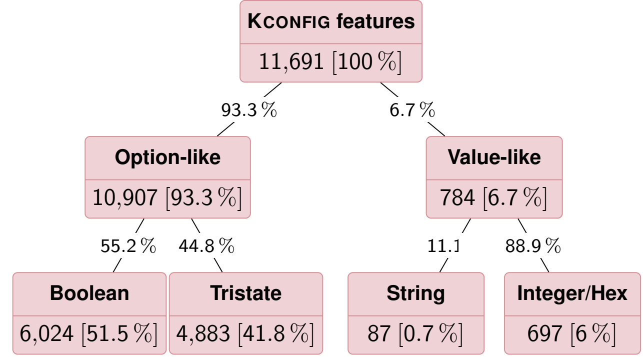

Figure 3. Distribution of features by type

files as compilation units in Step ❷. In Step ❹ compiler options and binding units are defined by this representation and used during the compilation and linking process. The exact mechanisms are fairly technical and have already been discussed elsewhere, (e.g. [13]). The **fine-grained variability** is implemented in `CPP` blocks (`#ifdef`) and their conditional expressions. These are controlled by the representation of the feature selection in `autoconf.h`. The file is forced included (with the `-include CPP` option) during the preprocessing Step ❸.

3. Feature Distribution in Linux

As part of the VAMOS[2] project we have been developing tool support [3] to quantify and analyze the specified (l_0) and implemented variability (currently l_1 and l_2) in Linux, that is, *features*, their *constraints* and the mapping between specified variability and implemented variability. In the following, we provide some results from our analyses with Linux v3.1, which defines a total of 11,691 feature symbols on the feature modeling level (l_0).

3.1 By Type

The KCONFIG configuration tool supports features of different types. Their distribution is depicted in Figure 3. Most of the features (> 90%) are declared as *option-like* features (either `boolean` or `tristate`). Their selection decides on the presence of a feature and how it is compiled (either linked statically or as a LKM). Additionally, we see that the 784 *value-like* (`string`, `integer` or `hex`) declared features are clearly dominated by *option-like* features. *Value-like* features are used for statically parametrising aspects of the Linux kernel (e.g. the fequency of the timer interrupt).

A sound and complete analysis of the variability in Linux requires both *option-* and *value-like* feature types as well as their constraints. However, we observe that *value-like* features generally don't enable feature code in practice, but (at least in Linux) are used to configure non-functional features such as timer frequencies, page sizes or the kernel version. We therefore conclude that tools for extraction of feature implementations that extract variability based on *option-like* features, (potentially) catch a reasonable subset of all available variation points.

3.2 By Architecture

Linux version v3.1 supports a total of 26 architectures, plus a number of additional sub-architectures. For each architecture Linux essentially defines a distinct feature model which share a large number of features. In total we count 7,226 (61.8%) different features that are shared by all main[4] architectures. The high number of distinct `tristate` features (4,657 or 64.4%) which are shared across all architectures, goes along with the fact that most drivers are modeled this way.

Many great papers (e.g., [4, 7–9, 13, 18]) have been published about applying static bug-finding approaches to Linux and other pieces of system software. In all cases the authors could find (and eventually fix) a significant number of bugs. It is, however, remarkable, that most papers do not even state the configuration(s?) they have analyzed, which, as [14] shows, can be a serious issue for reconstructing published experiments. We assume that they have used a single configuration that selects fewer features than the configuration preset `allyesconfig`, which raises the question how many feature implementations are actually analyzed.

[2] VAriability Management in Operating Systems

[3] Our tools will be made freely available at `http://www4.cs.fau.de/Research/VAMOS/tools.shtml`

[4] excluding `h8300`, `cris` and `um`

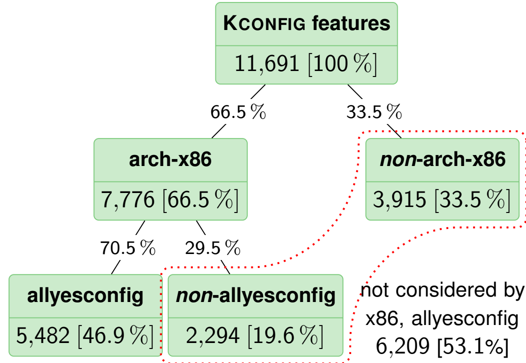

Figure 4. Coverarge of arch-x86 / allyesconfig

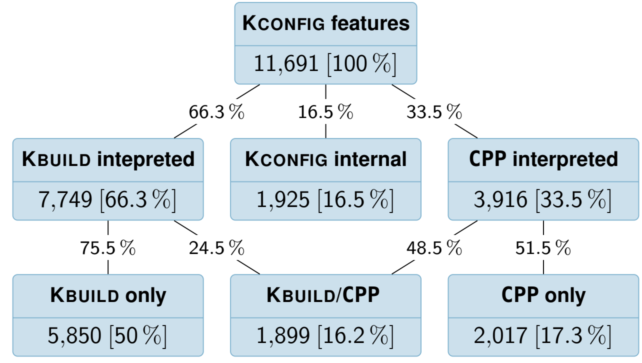

Figure 5. Distribution of features by granularity

Figure 4 depicts the feature distribution between the most often analyzed architecture `x86` and all others. In the first level, we see that restricting an analysis on arch-x86 already results in 3,915 unselected (and unselectable!) features. Additionally, the configuration preset `allyesconfig` selects only roughly half of all available features. These numbers show that more than 50% of the implemented features, and therefore a large amount of source-code, remain unconsidered when restricting an analysis to arch-x86/allyesconfig.

3.3 Coarse- Versus Fine-Grained Variability

Figure 5 shows the distribution of features by *implementation granularity*: Despite the fact that existing studies (including our own) have mostly focussed on the `CPP` as a means to implement features in Linux [10, 17, 18] only a third (33.5%) of all features do actualy affect the work of the `CPP`. These features have an effect on the sub-file level. On the other hand, two third (66.3%) of all features are referenced in the build system. These features have an effect on the selection of whole files into the build process as well as on general build options.

But the features that influence KBUILD and `CPP` aren't disjunct. Only 17.3% influence just the variability points within the source files. Half of all features (50%) are exclusivly used within the makefiles. Moreover, only 16.2% of all features influence both coarse-grained as well as fine-grained variability. The remaining 16.5% features are used only internally in KCONFIG as meta-variables for interconnecting features. They are exported in the different representations, but are not used outside of KCONFIG.

These numbers show that the amount of coarse-grained variability implemented in KBUILD is much larger than the fine-grained variability implemented with the `CPP` within the source files. Keeping in mind that variability points from KBUILD (level l_1 in Figure 1) dominates the `CPP` induced variability (level l_2), we find it surprising that only 16.2% of all features in KCONFIG get referenced by both the `CPP` and KBUILD. We therefore conclude that in order to get a holistic view on feature implementations in Linux, both coarse-grained and the fine-grained variation points have to be taken into account.

4. Related Work

In [5] Berger et al. investigate the configuration languages and tools KCONFIG and CDL. While the work shows that variability management tools are employed successfully in open-source operating-systems, it covers only the feature specification and modeling, that is, Step ❶ or level l_0 in Figure 2.

Adams demonstrates that analysis, visualization and in essence, re-engineering of the Linux build-system [2] is feasible, which corresponds to level l_1 (Step ❷). The framework Makao [1] infers modularity in KBUILD by analyzing build traces. However, the amount of variation points that we identify in KBUILD with this article indicates that the full re-engineering of build-system variability remains an unsolved problem.

Liebig et al. present a wide-ranged analysis of feature implementations in system software [11]. This work focuses on how the `CPP` is employed to implement (fine-grained) variability, which corresponds to level l_2 (Step ❸). Focusing on Linux, our analysis [18] of feature implementations in `CPP`-blocks reveals a number of configuration defects. Nadi et al. extend this work by including constraints from KBUILD [13], revealing additional configuration defects (Step ❷ in Figure 2). Because their approach uses parsing to extract variability from KBUILD, it exhibits similar problems as the appraoch in [4].

Kästner et al. propose a technique coined *variability aware parsing* [8], which basically integrates the `CPP` variability into tools for static analysis. This allows variability aware type-checking. Mainly because of implementation challenges, TypeChef focuses on `arch-x86` and requires assistance in form of additional constraints by tools like [4] or [18]. Even with this, the approach covers only variability from level l_3 – the build-system derived variability from level l_2 remains out-of-scope.

In contrast, [17] proposes to employ existing tools more effectively by running them multiple times. The idea is to calculate (partial) configurations that combined maximize the so called *configuration coverage*, which indicates the covered

amount of code. The results of Section 3 suggest promising benefits from better integration of build-system variability.

In [14], Palix et al. analyze the evolution of faults in Linux over the last decade. Already the setup of the experiment and reconstruction of the original analysis from [6] turned out as a challenge, because a considerable amount of work for finding the correct configuration, and therefore the exact set of files to analyze, had to be done.

5. Summary and Conclusion

This article continues our line-of-argument from earlier articles that variability has to be seen as source of bugs on its own respect. We identify a hierarchy of variability that shows how variability points, which are managed by different tools, concepts and languages related to and dominate each other.

In order to scope and relate the amount of induced variability by different tools, we use this hierarchy to quantify variability points in Linux v3.1 on the levels l_0 to l_2. By this, we provide numbers that support the relevance of *option-like* configuration. Moreover, we disprove the common belief that most variability was induced by the `CPP`. Quite the contrary, the presented numbers show that build system variability is much larger than generally expected. We see this as a call for further research on (proper) extraction of build-system induced variability to improve tools that analyze variability and feature implementations in software systems.

References

[1] ADAMS, B., DE SCHUTTER, K., TROMP, H., AND MEUTER, W. D. Design recovery and maintenance of build systems. In *23st IEEE Int. Conf. on Software Maintainance (ICSM'07)* (October 2007), IEEE, pp. 114–123.

[2] ADAMS, B., SCHUTTER, K. D., TROMP, H., AND MEUTER, W. D. The evolution of the Linux build system. *Electronic Communications of the EASST* (2007).

[3] Autoconf – GNU project – Free Software Foundation (FSF). `http://www.gnu.org/software/autoconf`, visited 2011-11-12.

[4] BERGER, T., SHE, S., CZARNECKI, K., AND WASOWSKI, A. Feature-to-code mapping in two large product lines. Tech. rep., University of Leipzig (Germany), University of Waterloo (Canada), IT University of Copenhagen (Denmark), 2010.

[5] BERGER, T., SHE, S., LOTUFO, R., AND UND KRZYSZTOF CZARNECKI, A. W. Variability modeling in the real: A perspective from the operating systems domain. In *25th IEEE Int. Conf. on Automated Software Engineering (ASE '10)* (2010), pp. 73–82.

[6] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating systems errors. In *18th ACM Symp. on OS Principles (SOSP '01)* (2001), ACM, pp. 73–88.

[7] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *18th ACM Symp. on OS Principles (SOSP '01)* (2001), ACM, pp. 57–72.

[8] KÄSTNER, C., GIARRUSSO, P. G., RENDEL, T., ERDWEG, S., OSTERMANN, K., AND BERGER, T. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *26th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '11)* (Oct. 2011), ACM.

[9] KREMENEK, T., TWOHEY, P., BACK, G., NG, A., AND ENGLER, D. From uncertainty to belief: inferring the specification within. In *7th Symp. on OS Design and Implementation (OSDI '06)* (2006), USENIX, pp. 161–176.

[10] LIEBIG, J., APEL, S., LENGAUER, C., KÄSTNER, C., AND SCHULZE, M. An analysis of the variability in forty preprocessor-based software product lines. In *32nd Int. Conf. on Software Engineering (ICSE '10)* (2010), ACM.

[11] LIEBIG, J., KÄSTNER, C., AND APEL, S. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *10th Int. Conf. on Aspect-Oriented Software Development (AOSD '11)* (2011), S. Chiba, Ed., ACM, pp. 191–202.

[12] MASSA, A. *Embedded Software Development with eCos*. New Riders, 2002.

[13] NADI, S., AND HOLT, R. C. Mining Kbuild to detect variability anomalies in Linux. In *16th Eur. Conf. on Software Maintenance and Reengineering (CSMR '12)* (2012), T. Mens, Y. Kanellopoulos, and A. Winter, Eds., IEEE. To appear.

[14] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J. L., AND MULLER, G. Faults in Linux: Ten years later. In *16th Int. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS '11)* (2011), ACM, pp. 305–318.

[15] SPENCER, H., AND COLLYER, G. #ifdef considered harmful, or portability experience with C News. In *1992 USENIX ATC* (June 1992), USENIX.

[16] SPINELLIS, D. A tale of four kernels. In *30th Int. Conf. on Software Engineering (ICSE '08)* (May 2008), W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds., ACM, pp. 381–390.

[17] TARTLER, R., LOHMANN, D., DIETRICH, C., EGGER, C., AND SINCERO, J. Configuration Coverage in the Analysis of Large-Scale System Software. In *6th W'shop on Progr. Lang. and OSes (PLOS '11)* (2011), A. SIGOPS, Ed., ACM.

[18] TARTLER, R., LOHMANN, D., SINCERO, J., AND SCHRÖDER-PREIKSCHAT, W. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2011 (EuroSys '11)* (Apr. 2011), ACM, pp. 47–60.