Towards Scalable Configuration Testing in Variable Software

Valentin Rothberg, Christian Dietrich, Andreas Ziegler, and Daniel Lohmann
Friedrich-Alexander University Erlangen-Nürnberg, Germany
{rothberg, dietrich, ziegler, lohmann}@cs.fau.de

Abstract

Testing a software product line such as Linux implies building the source with different configurations. Manual approaches to generate configurations that enable code of interest are doomed to fail due to the high amount of variation points distributed over the feature model, the build system and the source code. Research has proposed various approaches to generate covering configurations, but the algorithms show many drawbacks related to run-time, exhaustiveness and the amount of generated configurations. Hence, analyzing an entire Linux source can yield more than 30 thousand configurations and thereby exceeds the limited budget and resources for build testing.

In this paper, we present an approach to fill the gap between a systematic generation of configurations and the necessity to fully build software in order to test it. By merging previously generated configurations, we reduce the number of necessary builds and enable global variability-aware testing. We reduce the problem of merging configurations to finding maximum cliques in a graph. We evaluate the approach on the Linux kernel, compare the results to common practices in industry, and show that our implementation scales even when facing graphs with millions of edges.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing tools; D.2.9 [*Software Engineering*]: Software configuration management

General Terms Algorithms, Experimentation, Reliability

Keywords Software Testing, Configurability, Sampling, Software Product Lines, Linux

GPCE '16, October 31-November 01, 2016, Amsterdam, Netherlands
 ISBN 978-1-4503-4446-3/16/10$15.00
DOI: http://dx.doi.org/10.1145/2993236.2993252

"In fact our kernel configuration UI and workflow is still so bad that it's an effort to stay current even with a standalone and working .config, even for experienced kernel developers."

Ingo Molnar (Linux kernel developer)

1. Introduction

Testing a software product line (SPL) at the scale of the Linux kernel [1] is challenging. Kernel developers as well as advanced continuous-integration frameworks such as Intel's 0-day infrastructure [2] run various kinds of quality assurances, ranging from static analysis with Coccinelle [3, 4] to build and run-time tests including fuzzing [5]. The latter are particularly challenging, as it takes considerable time to build a complete kernel. The high variability with more than 15,000 configuration options and the fast evolution [6] of the Linux kernel increases testing complexity considerably. Even minor changes to variability points in the feature model (FM) [7, 8], the build system [9, 10] or the source code [11] can impact dozens to hundreds of different product variants [12]. Thus, testing as many potentially impacted variants as possible is crucial for the effectiveness of detecting bugs and regressions in general. In case of Linux, testing different variants implies compiling the kernel with different configurations, but generating such configurations is non-trivial. Manual approaches to assemble a set of configurations that build different variants of even one source file are barely possible, even for experienced developers [13].

Problem Statement and Contributions Research has proposed various solutions to automatically generate configurations for specified source files or configuration options, which is also referred to as *sampling*. Although a considerable amount of variability-related issues can be found by sampling [11, 14, 15], it is practically unusable in *realistic* testing scenarios for Linux due to several drawbacks. First, most algorithms show bad run-times when including constraints from the feature model and the build system, which is inherently important to avoid generating invalid configurations. Second, analyzing the entire source code at once (i.e., global analysis) is either not possible due to exponential nature of some algorithms, or it generates too few or too many configurations for exhaustive testing. Finally, per-

forming local analysis on many files simply yields too many configurations and thus exceeds the limited testing resources.

In this paper we present an approach to fill the gap between systematic approaches to generate configurations for testing purposes, and the necessary requirement to fully build software in order to test it. Instead of considering *global analysis* to generate configurations for many source files at once, we generate configurations on a file-local level and *merge* them in a subsequent process. We abstract the problem to finding maximum cliques in a graph and explain in detail our algorithms to build the graph. By using our tool, Troll, we can reduce the number of sampled configurations by more than 90 percent compared to traditional sampling approaches and thereby make file-based sampling tools usable for daily industrial development and testing. In summary, we claim the following contributions:

1. We present an approach to do *global analysis* using local sampling of software at the size and scale of Linux.
2. We explain in detail how our tool, Troll, can *scale* even when building graphs with hundreds of millions of edges on simple workstations.
3. We evaluate our approach in a qualitative study on the x86 architecture and the USB subsystem in Linux v4.5 using two well known and widely used sampling algorithms, *statement coverage* and *pairwise* sampling.
4. We show that our approach *improves the state-of-the-art* in (build) testing the Linux kernel.

Our approach can be applied to other software projects than the Linux kernel as well. However, the Linux kernel is a prominent example of highly configurable and variable open source software, so that we take it as a case study to explain and evaluate our approach in greater detail.

The remainder of the paper is structured as follows. In Section 2, we describe how the Linux kernel implements configurability, the importance of configurations for testing and the state-of-the-art in industry and research to generate configurations for testing purposes. In Section 3, we explain our approach, the problem abstraction to find maximum cliques, and how we designed the algorithms to scale even when facing graphs with hundreds of millions of edges. In Section 4, we evaluate our implementation on realistic testing scenarios and compare our results with commonly used approaches in industry. Before concluding in Section 7, we discuss our results in Section 5 and the threats to the validity of our approach in Section 6.

2. Background

In this section we briefly describe the build process of the Linux kernel and show how a user-specified configuration dominates this process from the coarse-grained selection of compilation units to the fine-grained decomposition of C source files via the C preprocessor. We further explain the different stages of testing in the development model of the Linux kernel, and discuss current approaches from industry and research to generate configurations for testing purposes.

2.1 Configurability in Linux

The feature model of the Linux kernel is specified in the *Kconfig* language. Kconfig offers various constructs to describe a feature model, most importantly configuration options (i.e., features). A Kconfig configuration option has a specified type (e.g., boolean, string, integer) and optional dependencies to describe constraints among options. Furthermore, the Linux-specific *tristate* type denotes features that can be compiled as run-time loadable kernel modules. The following Kconfig code-snippet shows the configuration option `FUTEX`, which can be selected by a user if she desires to support fast userspace mutexes. Note that the option is only visible to the user if its dependencies (i.e., `EXPERT`) are met. A detailed description of the semantics of the Kconfig language can be found in [16].

```
config FUTEX
  bool "Enable futex support" if EXPERT
  default y
  select RT_MUTEXES
```

Figure 1 illustrates the build process of the Linux kernel. After having parsed and evaluated the user-specified configuration ❶, the build system *Kbuild* includes and excludes entire source files from the build process ❷ and invokes the translation process. After preprocessing ❸, the GNU C compiler (GCC) compiles the set of source files ❹, which are finally linked ❺ into a bootable kernel image.

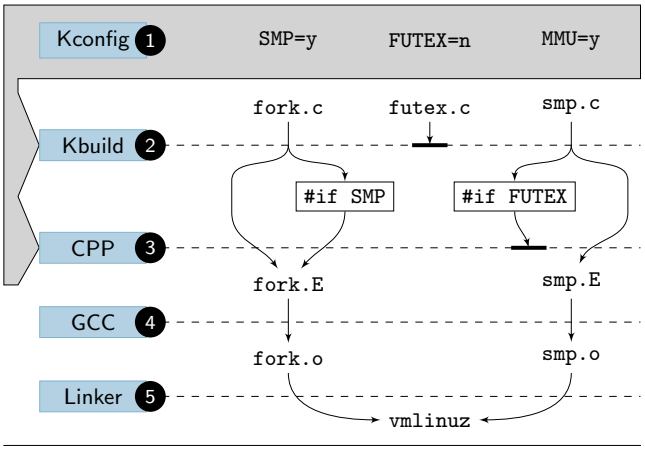

Figure 1. The build process of Linux in a nutshell.

The *Kbuild* build system of Linux requires a user-specified configuration file as input in order to decide which source files will be compiled into a specific product variant, allowing a coarse-grained decomposition of modules [17] into compilation units. Before compilation, Kbuild parses the configuration file and further translates it into Make syntax such that configuration options are accessible as Make variables to the build system. The Makefile code snippet below shows a build

rule to conditionally compile the `futex.c` module. Depending on the boolean configuration option `FUTEX`, the object file will be added to an internal list, which either includes or excludes its elements from the following compilation process. Please refer to Nadi et al. [10] for a detailed description of the Linux build process and conditional build rules.

```
obj-$(CONFIG_FUTEX) += futex.o
```

Among other software product lines [18, 19], Linux makes further use of the C preprocessor to allow a fine-granular decomposition of source code in the form of C preprocessor (CPP) #ifdef blocks. Source code guarded by such CPP blocks will only be part of the compilation unit if the block's condition (e.g., the value of a configuration option) evaluates to true [20]. Notice that Kbuild also translates the configuration into a C header, which is forcefully included into compilation to make configuration options accessible during preprocessing. The following simplified code snippet in the C language shows such conditional kernel code.

```
struct task_struct *copy_process(...) {
#ifdef CONFIG_FUTEX
    p->robust_list = NULL;
  #ifdef CONFIG_COMPAT
    p->compat_robust_list = NULL;
  #endif
    INIT_LIST_HEAD(&p->pi_state_list);
    p->pi_state_cache = NULL;
#endif
    return p;
}
```

2.2 Testing the Linux Kernel

The set of tests being performed highly depends on the role and position of the tester in the development process. In general, a *developer* proposes code changes in the form of a patch being sent via email to a development mailing list. After review, the responsible *maintainer* integrates the patch into her dedicated repository. Before such changes are merged into the *mainline* repository of Linus Torvalds, the changes have to pass a repository for continuous integration: *linux-next*. New versions of the kernel are released in a rather strict interval of two months [21]. After release of a new version, the so-called *merge window* opens; in a period of around two weeks, changes are pulled mostly from linux-next to the main repository [22]. After the merge window is being closed, the stabilization phase of this new kernel version starts, which lasts around six weeks. All in all, a patch will pass at least four instances before making it into a new version of the kernel: developer, maintainer, linux-next and mainline.[1]

At each step in the path of a patch, starting from writing the code changes to merging it into mainline, we need to make sure that the patch (a) does not introduce any build warnings or errors, that (b) the patch is semantically correct and that (c) it does not introduce other kinds of regressions and bugs such as unintended run-time behavior or negative impacts on performance – a *global* view on the changed source is of utmost importance. In general, the testing complexity increases the closer a patch is to being merged into the mainline repository. Besides the potential that merges of patches from different developers and different development repositories lead to build warnings and errors, new features interact for the first time with numerous impacts on semantics and performance [23]. Thorough tests of such feature interactions [24] are important to detect issues early in the development process and before shipping the product to users and customers.

In the past years, the Linux community has developed various testing frameworks. Some of those frameworks are directly shipped with the source code of Linux. Such is the case of *kselftest*, a developer-focused test framework which targets relatively short-running unit tests that are supposed to terminate in a timely fashion of 20 minutes. For the purpose of static analysis, there are currently almost 60 Coccinelle scripts which are also used to enable collateral evolution of the source code [25]. In addition to such kinds of tests and to tests that are performed by the developers and maintainers themselves, there is also an infrastructure for continuous integration (CI). The most prominent one is the Intel's 0-day kernel infrastructure, also called the 0-day robot. The robot monitors more than 600 development repositories and runs tests on newly integrated changes. On an average day, the 0-day robot performs more than 36,000 build tests, 20,000 boot tests and 12,000 performance tests. All those tests require a configuration in order to build the kernel.

2.3 Configurations for Testing

The Linux community employs mainly two approaches to tackle the problem of generating kernel configurations for testing. First, it is common to use a set of previously defined configurations that are considered to cover most configuration options of a specific architecture, hardware platform, driver or use case. For instance, there are more than 100 different so-called *default configurations* for the ARM architecture shipped with the Linux kernel source. Those default configurations can be used to build and test the Linux kernel on a specific ARM platform without redundantly configuring the kernel by each developer. Nonetheless, a developer who is interested in a specific driver or subsystem may maintain a set of kernel configurations on her own to cover more variants. As previous research stated [11], it also common to use the build system of Linux directly to generate configurations, for instance via '`make allyesconfig`' which is considered to achieve a high code coverage [11, 26].

The build system of Linux can further be used to generate random configurations, which is the second main approach to generate testing configurations in the Linux community. Depending on the selected architecture, the build system

[1] Notice that this process describes the path of an average patch. Security fixes in Linux may use a fast path and can be integrated without showing up on public mailing lists.

will assign random values to yet unselected configuration options. Using random configurations for testing purposes is considered to be comparably effective, since it generates combinations of configuration options that a user may not intuitively select. Melo et al. [27] confirmed the usefulness of random configurations in a quantitative study using random kernel configurations to catch and analyze compiler warnings. The authors found that most random configurations (i.e., 99 percent) indeed lead to compiler warnings. Such warnings can stem for instance from mismatches between the feature model and the implementation [28], or complex feature interactions spanning multiple files [14].

Modern continuous-integration frameworks such as Intel's 0-day infrastructure use both approaches, a pre-defined set of configurations for well known use-cases as well as randomized configurations. However, those approaches have several drawbacks. First, using a pre-defined set of configurations entails to continuously evolve and maintain those files, which is prone to errors and demands a high expertise and hence costs. Second, both approaches are not systematic. Especially using randomized configurations does not give any guarantee whether the code that is subject of testing will actually be compiled. Moreover, Melo et al. [27] found that the random-configuration generator of the Linux build system does not provide a uniform distribution over the entire configuration space.

2.4 State of Sampling Approaches

Previous research [11, 14, 26, 29, 30, 31, 32, 33] proposed various sampling [30] algorithms to automate the task of generating configurations that systematically enable or disable different variants of source code, or a given set of configuration options.

The *one-disabled* sampling algorithm has been proposed by Abal et al. [14] in a qualitative study of variability bugs in Linux. The authors found that disabling exactly *one* of *n* options per configuration helped to detect 40 out of 42 bugs while still being comparatively cheap. In [11], Tartler et al. presented a tool, Vampyr, which automates the task of generating configurations for a specified source file and invokes static analyzers (e.g., GCC or Coccinelle[2]) in a following step. The underlying sampling strategy *statement coverage* is implemented in the Undertaker[3] tool. Statement coverage tries to enable each C preprocessor #if block at least once – each statement will be compiled at least once – but it does not aim for enabling different combinations of blocks.

Medeiros et al. [15] compared a set of ten sampling algorithms with respect to effectiveness in terms of fault detection and how well the algorithms scale when considering constraints, when performing global analysis and when including header files. Although most algorithms show satisfying fault-detection capabilities, the authors made several observations that disqualify certain algorithms for our motivated goal: a systematic and scalable mechanism to generate configurations for testing purposes.

Many algorithms disqualify since they do not scale when considering constraints from the feature model or from the build system of Linux. Ignoring such constraints will yield a high rate of invalid configurations, which we consider to be inacceptable. Only non-combinatorial algorithms, such as one-disabled, one-enabled, statement coverage and most-enabled-disabled have shown promising results. An optimal solution to generate configurations for testing is so-called *global sampling*, which means to sample the entirety of variability (e.g., FM, #ifdef blocks, build system) at once. Global pairwise sampling, for instance, builds pairs among *all* available configuration options. However, global analysis is likely to cause an exponential explosion in the number of considered configuration options, even when sampling only few files at once. Only the one-enabled, one-disabled, most-enabled-disabled and randomized algorithms seemed to scale when performing global analysis. However, the one-enabled and one-disabled algorithms generate exactly n configurations for n available options and thereby exceed the testing resources for Linux with thousands of different configuration options. We exclude randomized sampling and most-enabled-disabled from our approach since it is already state-of-the-art in the Linux community (i.e., randconfig, allyesconfig and allnoconfig), which we seek to improve.

Similar to global analysis, exponential problems occur when including header files, since the amount of distinct configuration options increases dramatically; the average Linux source file includes three distinct options, whereas headers transitively add 238 distinct configuration options on average [15]. Considering variability information from Linux headers is desirable since developers are urged to move #ifdefs source files to headers [34] to increase readability of the code and avoid CPP related bugs. Nonetheless, especially global analysis is becoming increasingly important since configuration-related issues are likely to span over multiple files [35]. Some kinds of statically detectable bugs, such as linker errors, can only be revealed with build testing and thus requiring global analysis.

Our approach avoids the aforementioned exponential problems. Instead of sampling all variability (e.g., configuration options, #ifdef blocks) at once, we perform local sampling and merge the generated configurations afterwards by means of graph theory. Using such an approach, we can exploit the qualities of local sampling strategies, most importantly the fault-detection capabilities, while reducing the amount of initial configurations that need to be considered for testing purposes. In summary, we identify the following algorithms as candidates for our approach: one-enabled, one-disabled, pairwise sampling and statement coverage.

[2] http://coccinelle.lip6.fr

[3] https://undertaker.cs.fau.de

3. Approach

Our approach consists of two main steps. First, we make use of an efficient bit-parallel algorithm to identify compatible configurations and to build an undirected compatibility graph. In the second step, we find and select merge candidates by finding large cliques in the compatibility graph.

3.1 Configuration Compatibility Graph

After we have applied traditional sampling strategies (e.g., statement coverage) locally, we are left with a large number of configurations. In this first step, the sampling algorithm incorporates constraints from the feature model, the build system and the C preprocessor.

In the second step, we merge *compatible* configurations to reduce their total number, and build the *configuration compatibility graph* (CCG): an undirected graph $G = (V, E)$ with one vertex v for each locally-sampled configuration. The graph includes an edge $e \in E$ between two vertices $v_1, v_2 \in V$ if the corresponding configurations are compatible and do not conflict. v_1 and v_2 conflict if at least one common configuration option has differently assigned values, otherwise they are compatible. The algorithm to construct the CCG looks as follows:

for all $v_1 \in V$ **do**
  for all $v_2 \in V, v_1 \neq v_2$ **do**
    if $compatibleConfigs(v_1, v_2)$ **then**
      $graph.addEdge(v_1, v_2)$
    end if
  end for
end for

The resulting configuration compatibility graphs of Linux can grow up to more than 30 thousand vertices and 340 million edges. Since our implementation needs to scale on a single workstation in order to be useful for testing purposes, we developed a fast and efficient algorithm to check for configuration compatibility.

Fast and Efficient Comparison of Configurations In the following algorithm, we mainly exploit the fact that configuration options can only ship one in four values [11]: unset (0), enabled (1), disabled (2), and run-time loadable module (3). We consider two options compatible, if they have the same value or at least one value is *unset*. Since only four values are possible, a configuration option can be represented by two bits, such that the compatibility check of two options a and b, transformed into *algebraic normal form*, looks as follows:

$$compatible(a, b) = \neg((a_{bit0} \wedge b_{bit1}) \textbf{ xor } (a_{bit1} \wedge b_{bit0}))$$

Since this operation is easily expressible with bit operations, we can pack multiple options, with padding zero bits in between, into a single machine word and check them all at once. Then, the check only succeeds if all options in the machine word are compatible.

```
bool compatible(word a, word b) {
  return !((a & (b<<1)) ^ ((a<<1) & b);
}
```

Using such encoding, configurations can be represented as equally sized arrays of machine words, whereas a configuration option needs to be placed at the same index and the same offset within the indexed machine word in all arrays. This packing does not only reduce the memory consumption for the many thousand configurations, but also speeds up the compatibility check.

In our implementation, we packed 21 configuration options into one 64-bit machine word to subsequently check their compatibility in one pass. With this dense encoding and the bit parallelism, we could reduce the single-threaded execution time to build the graph (32,060 configurations, 15,069 options) from 30 minutes down to only three minutes. Note that we parallelized the process of building the CCG in our final implementation to gain further speedups.

3.2 Merging Vertices in the Graph

The overall goal of our approach is to reduce the number of configurations for testing without sacrificing the desired coverage in terms of C preprocessor #ifdef blocks. Therefore, we find groups of compatible configurations in the CCG and merge their configuration options into a single configuration. In terms of graph theory, we can only merge a set of vertices if they build a complete subgraph (i.e., a clique). Since we want to merge as many configurations as possible, we abstract the problem of merging configurations to finding maximum cliques in the CCG graph. The underlying algorithm is to iteratively find the maximum clique, remove its vertices from the graph and repeat until the graph is empty.

while $graph.notEmpty()$ **do**
  $clique \leftarrow graph.findClique()$
  $mergeConfigurations(clique)$
  $graph.removeVertices(clique)$
end while

Since we aim for a practical application of our tool even on a single workstation and since finding maximum cliques in an undirected graph is a NP-complete problem, we only search for *big* cliques within the CCG. In our implementation, we use the parallel maximum clique (PMC) library from Rossi et al. [36], which includes heuristics to approximate solutions of the maximum clique problem. We abort the search for a larger clique, after PMC reports the first heuristically large clique. In our experience, the approximated cliques reach 90+ percent of the optimum.

After we have identified a clique of compatible configurations, the actual merge operation is to concatenate the configurations and eliminate duplicate option assignments. The clique members are further removed from the CCG and the search for the next large clique is started.

3.2.1 Requirements for Merging

In order to successfully apply the presented approach – the merging process in particular – the generated configurations must meet the following requirements:

(a) The configurations must include all direct and transitive dependencies (i.e., a *slice* of the FM) to avoid merging invalid configurations. Finding a valid configuration for the entire slice implies using a SAT solver.

(b) For file-based approaches, such as statement coverage, we must further consider constraints from the build system to avoid generating and merging invalid configurations, since the build condition adds further constraints. Notice that around 50 percent of configuration options in Linux are mentioned exclusively in the build system [9]. Hence, we argue that file-based sampling without considering such variability is rather useless as it cannot reflect real dependencies and constraints.

(c) The sampled configurations must assign values only to configuration options in the slice of the variability model (i.e., feature model and build system) and leave all other options *unset*. This way, the compatibility of two configurations is purely determined by the two slices; all other options are unset and thus remain compatible. Such configurations are also referred to as *partial configurations* since only parts of the available configuration options are set. Nonetheless, a partial configuration must eventually be *expanded* (i.e., all unset options need to be assigned) in order to start the build process; we use Kconfig[4] directly for expansion.

A natural consequence of requirement (a) is that our approach only works on software projects exposing a feature model, such as Kconfig in Linux. The key for merging is the FM slice, which can be computed for any boolean expression (e.g., presence condition of an #ifdef block or a pair of configuration options) and basically consists of the transitive hull around referenced configuration options. Generating the slice can thus be done after sampling, making the presented approach independent from the used sampling algorithm.

3.3 Workflow Overview

Our tool implementation, Troll, is just one step in a bigger workflow depicted in Figure 2. The first step is to sample, for instance source files, and to generate *partial* configurations. Secondly, Troll parses the generated configurations and builds the configuration compatibility graph. As aforementioned, we developed a fast and efficient configuration-compatibility algorithm in order to let Troll scale even on a single workstation. The third step is to iteratively select cliques in the graph and merge the corresponding configurations. By merging the previously sampled *partial* configurations we seek to reduce the amount of configurations that need to be considered for testing purposes, such as build testing or run-time performance tests. The implementation of Troll is hosted on github[5]. All source code is licensed under the terms of the GPLv3, including scripts, compiler wrappers and a detailed README.

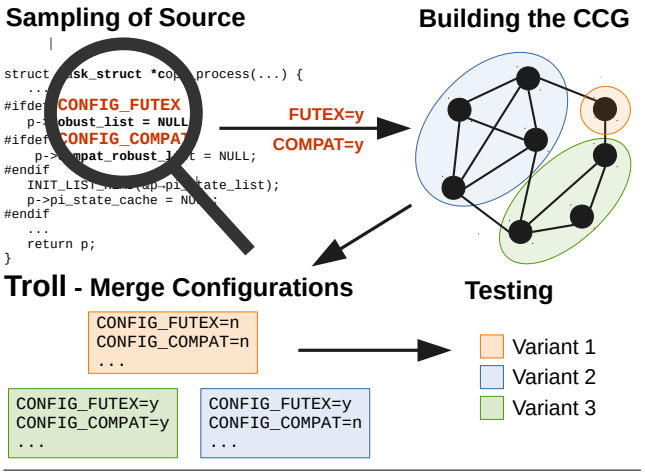

Figure 2. From sampling to merging configurations.

4. Evaluation

In the following evaluation we want to examine multiple aspects of our approach, which ultimately aims at producing a small set of testing configurations as fast as possible. Thus, we want to investigate (a) which sampling strategies we can use in a timely manner for differently sized samples of interest, (b) which compression rates Troll can achieve depending on the used sampling strategy, (c) how useful the merged configurations are. As shown by previous research, the fault-detection capabilities varies among the sampling strategies [15].

4.1 Study Design

In a first qualitative study we evaluate Troll from the perspective of a Linux maintainer. In this context, we contacted the maintainer of the Linux USB subsystem, Greg Kroah-Hartman (Linux Foundation), who uses *one* pre-defined kernel configuration for build testing his code base. In a second scenario we evaluate Troll on an entire Linux kernel of the x86 architecture. In both scenarios we use two sampling strategies. The first strategy is the *statement-coverage* algorithm, which is known to be fast while being weaker than others at detecting faults. The second strategy is *pairwise testing*. We chose this algorithm since combinatorial algorithms, although being slow, allow to systematically test interactions among configuration options (i.e., features) and perform best at detecting faults.

Since resources for testing are usually limited, we want to examine how useful subsets of varying sizes of the merged configurations are. Therefore we sort the merged configurations (top n) by the size of the clique

[4] `KCONFIG_ALLCONFIG=$PARTIAL_CONFIG make ...`

[5] https://github.com/vrothberg/troll

(the biggest clique is top 1). We further compare the sampled configurations with 121 randomly generated configurations, generated with '`make randconfig`'. The randomized configurations are sorted by the amount of enabled configuration options. Moreover, we use the following two metrics to compare the configurations:

1. *Block Coverage*
   The amount of C preprocessor #ifdef blocks being compiled by a specified configuration. Block coverage is a widely used metric to evaluate how much variable code can be enabled with a given configuration. Notice that this metric cannot necessarily be correlated with the fault-detection capabilities of the used sampling strategy.
2. *Sparse Warnings*
   Sparse[6] is a static analysis tool, initially developed by Linus Torvalds to find coding faults in the Linux kernel such as mixing pointers to user and kernel address spaces or unexpected acquiring and releases of locks. Previous work has shown that combinatorial sampling (e.g., t-wise sampling) performs best at detecting coding faults [15].

To determine the block coverage of a given kernel configuration, we parse *all* source files and headers for C preprocessor #ifdef blocks and insert a unique CPP #warning at the beginning of each block. Since the preprocessor will print a unique warning for each parsed block during the build process, we can determine exactly which #ifdef block is compiled by a given kernel configuration. To speed up the computation of block coverage, we implemented a wrapper for the GNU C compiler. Instead of performing expensive compilation such as source-code optimizations, we instrument the GCC to only perform preprocessing and generate empty dummy object-files afterwards. Using such a wrapper can speed up the computation of block coverage by a factor of nine.

In both evaluation scenarios we use a workstation with an Intel Core i7-4770HQ processor, 16 GiB of RAM and a solid state drive. For the sake of reproducibility and the ease of evaluation, a set of analysis scripts is shipped with the source of Troll. We use the Undertaker tool-suite in version 1.6 to extract the variability model and to perform the statement-coverage sampling. The pairwise sampling is performed with a Python script, which uses Undertaker as an interface for SAT solving and is included in the Troll repository. We further use Sparse in version 0.4.5-rc1.

4.2 Scenario 1: USB Subsystem

In version 4.5 of the Linux kernel, there are 703 source files and headers[7] directly related to the USB subsystem, and 596 distinct configuration options mentioned in Kconfig files, Makefiles and the source code.

[6] https://sparse.wiki.kernel.org/index.php/Main_Page

[7] ./drivers/usb/*, ./include/linux/usb/* and ./include/linux/usb.h

Amount of Sampled Configurations Sampling the 703 USB-related source files and headers yields 776 partial configurations with the statement coverage algorithm and took around 8 seconds. The pairwise sampling script generated 1,779 partial configurations in around 2.5 minutes. Thus, statement coverage generates 1.1 partial configurations per file, pairwise sampling generates 2.53 partial configurations per file.

Merging Configurations Using Troll we can merge the 776 partial configurations from the statement coverage to 121 configurations, and thereby reach a compression rate of around 85 percent. Figure 3 shows how many cliques of which size have been merged by Troll. We can see that there are many cliques of small sizes and one big clique of size 424. Hence, the biggest clique includes nearly 55 percent of the initial 776 partial configurations that we merged with Troll.

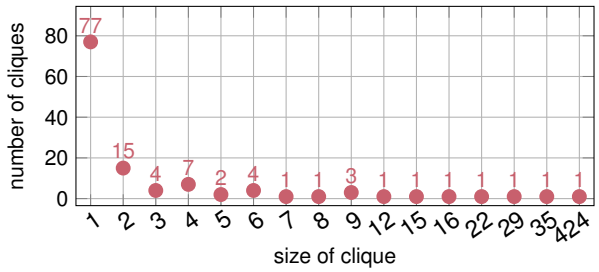

Figure 3. Size of merged configurations (sampled with statement coverage) in the USB subsystem of Linux v4.5.

The 1,779 pairwise sampled partial configuration can be merged to 186 configurations with Troll, showing a compression rate of 90.6 percent. Similar to the previous results of statement coverage, there are many cliques of small sizes and few big ones (see Figure 4). The biggest clique has a size of 458 and thus includes around 25 percent of the initial 1,779 partial configurations from pairwise sampling.

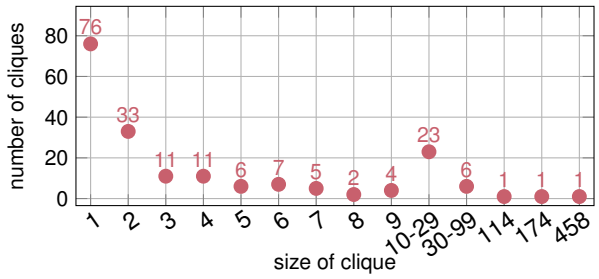

Figure 4. Size of merged configurations (pairwise sampled) in the USB subsystem of Linux v4.5.

Coverage and Sparse Warnings Table 1 compares the merged configurations of Troll with the 121 automatically randomized ones, which we generated using the built-in Linux make target '`randconfig`'. We further take the pre-defined kernel configuration of USB maintainer Greg Kroah-Hartman as a baseline that we seek to improve. We can see that the

top 1 clique from statement-coverage (stmt), as well as from pairwise sampling (pair), cover considerably more blocks than the top 1 randomized configuration (rand). The top 1 configurations from Troll further yield considerably more Sparse warnings (35 and 42) than the first random configuration (15) and instantly outperform the baseline (23) as indicated with the bold font. The top 2 and top 3 configurations from Troll beat the baseline in terms of covered blocks. Notice that there are 186 merged configurations from pairwise sampling, but only the top 121 are displayed in Table 1. However, the numbers in terms of covered blocks and Sparse warnings do not change between using the top 121 or top 186 configurations.

Config	Covered Blocks			Sparse Warnings			Builds
	stmt	pair	rand	stmt	pair	rand	(const.)
top 1	1,161	1,207	35	**35**	**42**	15	$1n$
top 2	1,163	**1,307**	437	35	43	**30**	$2n$
top 3	**1,426**	1,421	448	47	50	30	$3n$
top 10	1,438	1,441	1,140	47	50	36	$10n$
All (121)	1,495	1,486	**1,516**	47	50	55	$121n$
Base	**1,238**			**23**			$1n$

Table 1. Evaluation of `drivers/usb/`.

4.3 Scenario 2: x86 Kernel

In this evaluation scenario we use Troll on a bigger scale and sample configurations for all source files and headers related to an entire x86 kernel.[8] This is a realistic use case for continuous-integration systems, which face commits that potentially change huge parts of the source code.

Amount of Sampled Configurations The *statement-coverage* sampling generated 30,327 partial configurations for 29,887 files related to x86 (i.e., 1.01 configurations per file) in 15 minutes. Sampling the 29,887 files *pairwise* took around two hours and yielded 134,173 partial configurations (i.e., 4.49 configurations per file).

Merging Configurations Using Troll we merged the initial 30,327 partial configurations from *statement coverage* to 1,973 configurations within three minutes, reaching a compression of 93 percent. The resulting graph has more than 340 million edges and a total size of 2.3 GiB in the matrix market exchange format (MTX).[9] As previously observed in the USB subsystem, there are many small cliques and few big ones. In fact, 1,473 cliques have a size of one which means that those configurations could not be merged. On the contrary, the biggest clique has a size of 18,727 – 62 percent of the initial graph.

Merging the 134,173 *pairwise sampled* partial configurations took 740 minutes and yielded 4,997 configurations with a compression rate of 96.3 percent. The resulting graph has 5.5 *billion* edges and a size of 17 GiB in the MTX format. Due to the pure size of the graph we needed another machine with 128 GiB of RAM for merging. Again, there are many small cliques and few big ones, whereas each of the first two cliques includes nearly 20 percent of the initial partial configurations.

Coverage and Sparse Warnings Table 2 compares the top 10 merged configurations from statement coverage and pairwise sampling with the top 10 randomized configurations. The baseline is the x86 *allyesconfig*, which is widely considered to achieve a high coverage in terms of code and #ifdef blocks and is commonly used for bug and fault-detection purposes in the Linux community. We can read from the table that randomized configurations perform worse, both in terms of coverage and fault detection (i.e., Sparse warnings). We can see that the merged configurations from both sampling algorithms can beat the baseline (i.e., allyesconfig) with the top 1 to top 4 configurations, depending on the algorithm and metric. The table also emphasizes the differences between and the strengths of both algorithms. In terms of coverage, both algorithms show comparable results with few merged configurations but differ strongly in the top 10, where statement coverage covers 1.17 times more blocks than pairwise sampling. Regarding Sparse, pairwise sampling yields consistently more warnings than statement coverage; the top 10 yield 1.26 more warnings.

Config	Covered Blocks			Sparse Warnings			Builds
	stmt	pair	rand	stmt	pair	rand	(const.)
top 1	57,489	51,212	23,069	4,779	**5,306**	2,123	$1n$
top 2	**59,713**	56,368	34,004	4,813	6,014	2,861	$2n$
top 3	60,393	**60,407**	40,206	4,887	6,733	3,358	$3n$
top 4	60,632	61,119	40,886	**5,332**	6,758	3,394	$4n$
top 10	75,065	64,274	49,973	5,548	7,007	4,485	$10n$
Base	**59,042**			**5,126**			$1n$

Table 2. Evaluation of an entire x86 Linux kernel.

5. Discussion

In the following section we will reflect on our results, discuss the usefulness of our tool and analyze weaknesses of our approach.

Suitability of Sampling Strategies When it comes to testing in the context of continuous integration we face systems with limited resources in terms of time and throughput (i.e., builds per commit). Thus, a key criteria to a successful adaption of our approach is the time it takes to generate configurations for testing. We deduce from the results of our evaluation that *statement-coverage* sampling can be used even on an entire x86 Linux kernel with nearly 30 thousand source files and headers. However, *pairwise* sampling should be preferred whenever possible. As previous studies have shown, such combinatorial sampling strategies show better fault-detection capabilities, which aligns with our results. But due the combinatorial nature and the quadratic amount of SAT calls, pairwise sampling should only be used on rather small inputs. The overall run-time of pairwise sampling is deter-

[8] ./arch/x86/ and all remaining subsystems including drivers

[9] http://math.nist.gov/MatrixMarket/formats.html#MMformat

mined (a) by the amount files to parse, and (b) the amount of pairs to check with the SAT solver. We consider 2.5 minutes for pairwise sampling the USB subsystem to be acceptable with around 700 files and 2,718 pairs (18.12 pairs per second).

Merging Big Graphs As we have shown in the evaluation, Troll can build and merge graphs with more than 30 thousand vertices and 430 million edges in around three minutes, what we consider to be reasonably fast. However, when facing bigger graphs of more than 100 thousands vertices (e.g., pairwise sampling on x86 Linux) the NP-hard problem of finding cliques yields unacceptable run-times of over 700 minutes and increases the hardware requirements to more than 120 GiB of RAM. When facing such graphs, it is possible to partition the graph by grouping configurations regarding their subsystem and iteratively merge them in a subsequent step.

Global Analysis and Header Inclusion As stated before, global analysis is too time consuming [15] when sampling multiple source files at once, if even possible. Our approach avoids such scalability issues by doing local sampling first, and merging the generated configurations in a following process with Troll. If a resulting configuration subsumes more than one local result, it provides a more global view on the analyzed system. In the evaluation of the x86 kernel, we could merge 62 percent of the initial partial configurations into one big, global configuration.

Including headers when sampling a source file is also known to fail due to the high number of (transitively) included headers and the entailed exponentially higher variability. Similarly to global analysis, we avoid such scalability impairments by doing local analysis on each header, and adding the sampled configurations of all headers to the configuration compatibility graph, such that they can subsequently be merged. Hence, we include header-induced variability in our configurations. However, our current approach does not consider #includes, which would be possible using a directed configuration compatibility graph and merge configurations following the actual include graph.

Comparison to Randomized Configurations Our data confirms one of our initial claims that randomized configurations cannot give any guarantee if the code of interest will actually be compiled and hence tested. In contrast to `randconfig`, our approach is able to give certain guarantees. Since the process of merging does not alter any option–value pair, all file-local results are to be found in the merged configurations; our results are neither better nor worse than the sampled configurations. In other words, the merged configurations compile exactly the same source files and achieve the same block coverage than the input configurations. For example, using Undertaker's statement-coverage sampling we can achieve around 90 percent block coverage of an x86 Linux kernel [11]. However, we cannot make a statement about the *top n* merged configurations.

Detecting Bugs in the FM Much to our surprise, we could not detect any compiler warning with our merged configurations in the USB subsystem, which indicates a high quality of USB-related source code. However, we found 16 warnings from Kconfig complaining about the selection of configuration options with unmet dependencies. The problem behind this warning lies in the semantics of the '`select`' statement of the Kconfig language, which allows to forcefully enable a configuration option even when its dependencies are not satisfied. The select issue is a natural consequence of the absence of a SAT checker in Kconfig and developers consider such cases as *illegal* and are advised to use the select statement only for options without prompts and for options with no dependencies. We further investigated the 16 Kconfig warnings and found that they all relate to the configuration option `SND_SST_IPC_ACPI` whose dependencies were poorly designed, and even lead to build errors. This issue has now been fixed[10] by changing the constraints in the feature model. We deduce that our approach can help to detect such bugs in the FM. Notice that Linux developers and fellow researches are currently working to add SAT-checking capabilities to Kconfig in order to solve the select issue and to define clear and meaningful semantics in the Kconfig language.[11]

Reducing the Number of Builds By using Troll, we could reduce the amount of configurations and hence the number of builds by up to 96 percent compared to local sampling. We argue that using our approach can significantly reduce the number of builds, which either gives more time to perform additional tests or further reduces the high requirements regarding resources (e.g., number and performance of machines, energy consumption, etc.). Nonetheless, testing more than a hundred configurations per commit may exceed today's systems. In build-restricted environments we recommend using the *top n* merged configurations. As we have shown in the evaluation, the top 1 to top 4 merged configurations are sufficient to increase the number of covered C preprocessor blocks as well as the number of Sparse warnings in both evaluation scenarios compared to the baseline.

Compression Rates We expected a much higher compression rate than 85 percent (statement coverage) in case of the USB subsystem, since configuration options of drivers can usually be selected without constraining others, but 77 of 121 cliques are of size 1 and thus can not be merged. We blame the model slicing in combination with SAT solving to generate rather "exotic" configurations in some cases, resulting in incompatible configurations. We believe that avoiding such "exotic" configurations can yield higher compression rates, and we plan to address the issue by coming up with a topology of model slices, such that common sub-slices are equally assigned.

[10] https://patchwork.kernel.org/patch/9168611/

[11] http://kernelnewbies.org/KernelProjects/kconfig-sat

6. Threats To Validity

Extracted Variability Model Errors or inaccuracies in the extracted models could lead to inaccurate or even invalid configurations with respect to the actual constraints, which further affects the computed feature model slices. El-Sharkawy et al. [16] have shown that the feature models extracted by Undertaker's dumpconf, among other tools, do not accurately reflect the semantics of Kconfig. However, at the current state the impact of the models' inaccuracies on analysis as such performed in this study is unclear; some unsupported cases affect only certain versions of the Linux kernel or do not practically impact its models (e.g., the `option module` case [16]). In case of Linux, we can extract more than 95 percent of all build-system constraints [9].

Feature Model Slice Computing the feature model slice is not trivial, since an underapproximation can lead to invalid merged configurations, an overapproximation can lead to bad compression rates. We have evidence that the configurations used in our study are not invalid, since we tested all merged configurations without encountering any errors from Kconfig. The absence of invalid configurations makes us confident that (a) the quality of the extracted variability models is sufficient for our approach, and that (b) the slicing algorithm to find transitive dependencies implemented in the Undertaker generates reliable results.

Detecting Cliques As described in Section 3, we make use of the PMC library from Rossi et al. [36]. Bugs in the PMC library can lead to mistakenly merged configurations, which further lead to invalid configurations. However, the PMC library has been successfully applied to graphs representing various problems in different domains, from biological to information networks.[12] The absence of invalid merged configurations is further indicating reliable results from PMC.

Compression Rate Since finding the maximum clique in a graph is a NP-complete problem, our implementation heavily relies on the heuristics [36] implemented in the PMC library. In fact, we do not compute the optimum but approximate with potentially negative impact on the achieved compression rate. Figure 5 shows the time to compute cliques of increasing sizes up to the maximum clique in the configuration compatibility graph of the USB subsystem of Linux v4.5 with around 16 thousand vertices. In total, the computation of the maximum clique with 11,050 vertices takes nearly 6,000 seconds. However, Figure 5 also shows that the default maximum-clique heuristics implemented in PMC work remarkably well. After 0.33 seconds a clique of size 10,850 is found. Thus, finding the actual maximum clique takes more than 18,000 times longer than the first approximation, while increasing the size of the clique only by a factor of 1.07. We consider the negative impact of our approximation to be acceptable when comparing it to the decrease of execution time.

[12] http://ryanrossi.com/pmc/download.php

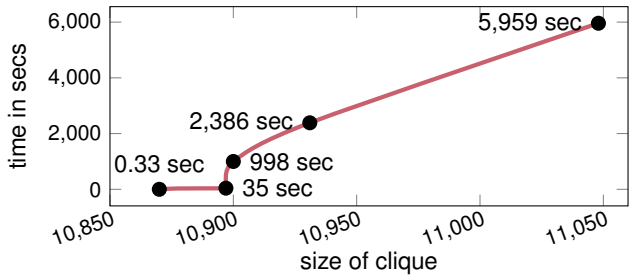

Figure 5. The computation time to find cliques of increasing sizes with the PMC solver.

Generalizability of the Approach In this paper, we used the Linux kernel in version 4.5 (March 2016) as a case study. Although Linux might be the biggest available highly variable software project, our approach can easily be ported to other software projects exposing a feature model (e.g., via Kconfig) and build-system constraints. Previous work has already successfully applied sampling to the L4/FIASCO µ-kernel, the Busybox coreutils generator for embedded systems [11], and many more software projects of varying domains [15]. Our approach is further independent from the employed sampling algorithm. However, having a variability model at hand is a key precondition for the sampling algorithm to generate partial configurations that meet the requirements formulated in Section 3.2.1.

7. Conclusion and Future Work

In this paper we presented an approach that can lift file-local sampling to global analysis of variable software of the size and complexity of the Linux kernel. Instead of considering global analysis as sampling multiple source files at once, we use local sampling and merge the generated configurations in a subsequent step. Our results show compression rates of more than 90 percent and that the top 1 to 5 configurations suffice to considerably improve the state-of-the-art in Linux in terms of the amount covered #ifdef blocks and Sparse warnings. We believe that investing into further research on achieving scalable sampling approaches will pay off. We are especially interested in the problem of including headers when sampling, which also faces exponential explosions. By using information from a variability-aware include graph [37], we can transform the configuration compatibility graph into a directed graph and consequently merge configurations of transitively included headers.

Acknowledgments

We want to thank Greg Kroah-Hartman for kindly providing his build-testing configuration for the USB subsystem, and the reviewers for their remarkably detailed and very constructive feedback. This work has been supported by the German Research Council (DFG) under grant no. LO1719/3-1.

References

[1] Julio Sincero et al. “Is The Linux Kernel a Software Product Line?” In: *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*. 2007.

[2] Michael Kerrisk. *Kernel build/boot testing*. 2012. URL: https://lwn.net/Articles/514278/.

[3] Henrik Stuart et al. “Towards Easing the Diagnosis of Bugs in OS Code”. In: *Proceedings of the 4th Workshop on Programming Languages and Operating Systems*. 2007, 2:1–2:5.

[4] Julia L Lawall et al. “WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code”. In: *IEEE/IFIP International Conference on Dependable Systems & Networks*. 2009, pp. 43–52.

[5] Dave Jones. *Trinity: A Linux system call fuzzer*. URL: https://github.com/kernelslacker/trinity.

[6] Rafael Lotufo et al. “Evolution of the Linux Kernel Variability Model”. In: *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*. 2010, pp. 136–150.

[7] Steven She et al. “Variability Model of the Linux Kernel”. In: *Fourth International Workshop on Variability Modeling of Software-intensive Systems (VaMoS 2010)*.

[8] Julio Sincero et al. “Facing the Linux 8000 Feature Nightmare”. In: *Proceedings of ACM European Conference on Computer Systems (EuroSys 2010), Best Posters and Demos Session*.

[9] Christian Dietrich et al. “A Robust Approach for Variability Extraction from the Linux Build System”. In: *Proceedings of the 16th International Software Product Line Conference*. Vol. 1. 2012, pp. 21–30.

[10] Sarah Nadi and Ric Holt. “Mining Kbuild to detect variability anomalies in Linux”. In: *16th European Conference on Software Maintenance and Reengineering (CSMR)*. 2012, pp. 107–116.

[11] Reinhard Tartler et al. “Static analysis of variability in system software: The 90,000# ifdefs issue”. In: *Proc. USENIX Conf* (2014), pp. 421–432.

[12] Andreas Ziegler, Bernhard Heinloth, and Daniel Lohmann. “Automatic Feature Selection in Large-Scale System-Software Product Lines”. In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. 2014, pp. 39–48.

[13] Sarah Nadi et al. “Mining Configuration Constraints: Static Analyses and Empirical Results”. In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 140–151.

[14] Iago Abal, Claus Brabrand, and Andrzej Wasowski. “42 variability bugs in the linux kernel: a qualitative analysis”. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 421–432.

[15] Flávio Medeiros et al. “A Comparison of 10 Sampling Algorithms for Configurable Systems”. In: *Proceedings of the 38th International Conference on Software Engineering - Volume 2*. ICSE ’16. 2016.

[16] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. “Analysing the Kconfig Semantics and Its Analysis Tools”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 2015, pp. 45–54.

[17] D. L. Parnas. “On the Criteria to Be Used in Decomposing Systems into Modules”. In: (1972), pp. 1053–1058.

[18] Jörg Liebig et al. “An analysis of the variability in forty preprocessor-based software product lines”. In: *ACM/IEEE 32nd International Conference on Software Engineering*. 2010, pp. 105–114.

[19] Michael D. Ernst, Greg J. Badros, and David Notkin. “An Empirical Analysis of C Preprocessor Use”. In: *IEEE Trans. Softw. Eng.* (2002), pp. 1146–1170.

[20] Alejandra Garrido and Ralph Johnson. “Analyzing multiple configurations of a C program”. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance*. 2005, pp. 379–388.

[21] Jonathan Corbet and Greg Kroah-Hartman. *Linux Kernel Development Report 2016*. 2016.

[22] Jonathan Corbet. *A day in the life of linux-next*. URL: https : / / lwn . net / Articles / 287155/ (visited on 09/06/2016).

[23] Norbert Siegmund et al. “Predicting Performance via Automated Feature-interaction Detection”. In: *Proceedings of the 34th International Conference on Software Engineering*. 2012, pp. 167–177.

[24] Brady J Garvin and Myra B Cohen. “Feature interaction faults revisited: An exploratory study”. In: *2011 IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE)*. 2011, pp. 90–99.

[25] Yoann Padioleau et al. “Documenting and Automating Collateral Evolutions in Linux Device Drivers”. In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. 2008, pp. 247–260.

[26] Reinhard Tartler et al. “Configuration Coverage in the Analysis of Large-Scale System Software”. In: *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*. 2011, p. 2.

[27] Jean Melo et al. “A Quantitative Analysis of Variability Warnings in Linux”. In: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. 2016.

[28] Reinhard Tartler et al. “Feature Consistency in Compile-Time Configurable System Software”. In: *Proceedings of the EuroSys 2011 Conference (EuroSys ’11)*. 2011, pp. 47–60.

[29] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. “An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models”. In: *Proceedings of the 16th International Software Product Line Conference-Volume 1*. 2012, pp. 46–55.

[30] Jörg Liebig et al. “Scalable analysis of variable software”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013, pp. 81–91.

[31] Malte Lochau et al. “Model-based pairwise testing for feature interaction coverage in software product line engineering”. In: *Software Quality Journal* 20 (2011), pp. 567–604.

[32] Gilles Perrouin et al. “Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines”. In: *Third International Conference on Software Testing, Verification and Validation (ICST)*. 2010, pp. 459–468.

[33] Sebastian Oster, Florian Markert, and Philipp Ritter. “Automated Incremental Pairwise Testing of Software Product Lines”. In: *Software Product Lines: Going Beyond*. 2010, pp. 196–210.

[34] *Linux Coding Style Guidelines*. URL: https://www.kernel.org/doc/Documentation/CodingStyle (visited on 09/06/2016).

[35] Flávio Medeiros et al. “An Empirical Study on Configuration-Related Issues: Investigating Undeclared and Unused Identifiers”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 2015, pp. 35–44.

[36] Ryan A Rossi et al. “A fast parallel maximum clique algorithm for large sparse graphs and temporal strong components”. In: *CoRR, abs/1302.6256* (2013).

[37] Andreas Ziegler, Valentin Rothberg, and Daniel Lohmann. “Analyzing the Impact of Feature Changes in Linux”. In: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. 2016, pp. 25–32.