Static Analysis of Variability in System Software: The 90,000 `#ifdef`s Issue*

Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, Daniel Lohmann
{tartler, dietrich, sincero, wosch, lohmann}@cs.fau.de
FAU Erlangen-Nürnberg

Abstract

System software can be configured at compile time to tailor it with respect to a broad range of supported hardware architectures and application domains. The Linux v3.2 kernel, for instance, provides more than 12,000 configurable features, which control the configuration-dependent inclusion of 31,000 source files with 89,000 `#ifdef` blocks.

Tools for static analyses can greatly assist with ensuring the quality of code-bases of this size. Unfortunately, static configurability limits the success of automated software testing and bug hunting. For proper type checking, the tools need to be invoked on a concrete configuration, so programmers have to manually derive many configurations to ensure that the configuration-conditional parts of their code are checked. This tedious and error-prone process leaves many easy to find bugs undetected.

We propose an approach and tooling to systematically increase the *configuration coverage* (CC) in compile-time configurable system software. Our VAMPYR tool derives the required configurations and can be combined with existing static checkers to improve their results. With GCC as static checker, we thereby have found hundreds of issues in Linux v3.2, BUSYBOX, and L4/FIASCO, many of which went unnoticed for several years and have to be classified as serious bugs. Our resulting patches were accepted by the respective upstream developers.

1 Introduction

System software typically employs compile-time configuration as a means to tailor it with respect to a broad range of supported hardware architectures and application domains. A prominent example is Linux, which in v3.2 provides more than 12,000 configurable features (KCONFIG options) that control the inclusion of 31,000 source files and 89,000 `#ifdef` blocks when building the Linux kernel. The huge and growing amount of configurability in modern system software implies quite some challenges with respect to testing and maintenance.

Static configurability is mostly implemented by source-code transformations [16, 17]: The build system and textual preprocessors, such as the *C Preprocessor* (`CPP`), interpret configuration flags to (a) filter the set of compilation units and (b) transform their actual content before passing them to the compiler. Consider this example of a *variation point* implemented with `CPP` in Linux:

```
#ifdef CONFIG_NUMA
  Block1
#else
  Block2
#endif
```

For any given configuration, depending on the configuration switch `CONFIG_NUMA`, *either* $Block_1$ *or* $Block_2$ is passed to the compiler (or any other static checker that drops in as a compiler replacement). This means that the responsible maintainer has to derive at least two configurations to validate that each line of code does even *compile*. This is not trivial: `CONFIG_NUMA` and the containing translation unit are constrained by further rules and configuration switches in the make files (KBUILD) and the feature model (KCONFIG) that all have to be set to the right values.

It is not hard to imagine that doing this manually does not work in practice. Nevertheless, this is the state of the art: Point 8 from the *Linux Kernel patch submission checklist*[1] requires, that all submitted code

> *has been carefully reviewed with respect to relevant* KCONFIG *combinations. This is very hard to get right with testing – brainpower pays off here.*

Our approach **replaces brainpower by tools**:

```
$ git am bugfix.diff  # Apply patch
$ vampyr -C gcc --commit HEAD  # Examine
```

*This work was partly supprted by the German Research Council (DFG) under grant no. LO 1719/3-1

[1] cf. `Documentation/SubmitChecklist` in the source tree

VAMPYR maximizes the *configuration coverage* (CC) by automatically deriving a set of configurations that together cover all variation points (`#ifdef` blocks and configuration-conditional files) in all files modified by the patch. It then invokes the build-system for each configuration, in this case with GCC as static checker.

1.1 Problem: Configuration Coverage

Many papers (e.g., [2, 4, 7, 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 a significant number of bugs. It is remarkable that the issues of configuration-conditional code and CC is not mentioned at all in these papers – the authors do not even state which configuration(s) they have analyzed. This does not only raise strong issues with respect to scientific reproducibility,[2] but also potentially limits their success: We have to assume that only a single configuration and architecture was analyzed (as also reported by Palix and colleagues [15]) – so how many bugs were missed that could have been found with full coverage?

There also is a more practical side of CC: A Linux developer, for instance, who has modified a bunch of files for some maintaining task would probably want to make sure that every edited line of code does actually compile and has been tested before submitting a patch. However, how to derive a set of configurations that covers all configuration-conditional pieces?

Deriving a configuration that reliably selects a particular configuration-conditional part of the code is not trivial. The problem is that for proper type checking the configuration needs to be sound and complete, as only then all header include paths are available and all types are properly resolved. To derive such a configuration, not only the C source code, but also the build system, feature model, and architecture have to be examined.

In practice, this leads to the situation that only a single architecture and configuration is checked. In the case of Linux, this typically is Linux/`x86` and – in the best case – the predefined `allyesconfig` configuration, which is supposed to be a maximum configuration. However, current versions of Linux support more than twenty architectures and `allyesconfig` is by far not a full configuration: Depending on the architecture, it covers only 42–83 percent of all configuration-conditional parts of the code. The result is that – despite extensive code reviews and other quality measures performed by the community – Linux contains quite some code that does not even compile.

[2] In their "Ten years later" paper, Palix and colleagues describe the enormous difficulties to figure out the Linux v2.4.1 configuration used by Chou *et al.* in [2] in order to reproduce the results. Eventually, they had to apply source-code statistics to figure out the configuration "that is closest to that of Chou *et al.*" [15].

1.2 Our Contributions

Our variability-aware driver VAMPYR mitigates these problems. It maximizes the CC by automatically deriving a set of configurations. By just employing the *compiler* (GCC) as a static checker, we thereby already can find hundreds of issues in Linux, L4/FIASCO, and BUSYBOX. In particular, we claim the following contributions:

(a) We analyze the conceptual and technical issues of configuration-dependent bugs (Section 2) and quantify how many variation points of the Linux source base are missed by the current state of the art (Section 4).

(b) We present an approach and tool implementation to systematically increase the CC in compile-time configurable system software (Section 3). Our approach and the resulting VAMPYR tool provide an easy and noninvasive integration into existing build systems and combination with existing code checkers, such as CLANG, GCC, SPARSE or COCCINELLE [14].[3] Besides Linux, we also have applied our approach to the L4/FIASCO μ-kernel and the BUSYBOX coreutils generator for embedded systems.

(c) Our experimental studies with GCC 4.7 as a static checker have revealed hundreds of issues (Section 5). For Linux/`arm` VAMPYR increases the CC (compared to `allyesconfig`) from 59.9 to 84.4 percent, which results in 199 additionally reported configuration-conditional issues (compiler warnings and errors). 91 of these issues have to be classified as serious bugs. We proposed patches for seven bugs in Linux and one in L4/FIASCO and BUSYBOX. All patches got accepted and the responsible developers have confirmed the found bugs. Some bugs went unnoticed for up to six years – just because they do not show up in a standard configuration.

1.3 Previous Work

This paper builds on previous work, especially the open-sourced variability extractors for KCONFIG, KBUILD, and `CPP` we have presented in [3, 20]. Our work on *variability defects* in Linux [20] reveals bugs in `#ifdef` expressions or KCONFIG constraints, such as typos in the feature identifiers or presence conditions that are a tautology/contradiction, so that the `#ifdef` statement or the complete block can be removed. In essence [20] finds faulty `#ifdef` statements, but does not look *inside* the thereby constrained blocks of code. This is what VAMPYR does by maximizing the CC of existing static checkers, so in this paper we look for a very different kind of configurability-related bugs.

In a previous workshop paper [19], we have sketched the issue of CC, the coverage of `allyesconfig` (re-

[3] VAMPYR and all related tools presented this paper are available for download under GPLv3 at `http://vamos.cs.fau.de/trac/undertaker`.

stricted to Linux/x86), and the idea to improve the CC by employing multiple configurations. However, we did not find any bug; our results regarding CC later turned out to be *way* too optimistic: The generated configurations were not sound, as we did not consider the coarse-grained variability implemented by the build system. This led to the development of our variability extractor for KBUILD [3], which we have integrated into our VAMPYR tool for this work. Only thereby, VAMPYR has become a practically usable driver for static checkers that has already helped to identify hundreds of issues in Linux, L4/FIASCO, and BUSYBOX.

2 Configuration-Dependent Bugs

The goal of our approach is to increase the effectiveness of existing static analysis tools so that they reveal also *configuration-dependent bugs*. Those are bugs that manifest only in configuration-conditional parts of the code, such as `#ifdef` blocks or configuration-conditional source files.

We classify a bug as a configuration-dependent bug, iff there exists any configuration in which the bug is not observed. In the following, we present some examples of configuration-dependent bugs we have found by maximizing the CC of GCC with VAMPYR:

(1) Consider the following situation in the HAL for the ARM architecture in Linux. In the file `arch/arm/mach-bcmring/core.c`, the timer frequency depends on the configured derivate:

```
#if defined(CONFIG_ARCH_FPGA11107)
/* fpga cpu/bus are currently 30 times slower so
     scale frequency as well to slow down Linux's
     sense of time */
[...]
#define TIMER3_FREQUENCY_KHZ (tmrHw_HIGH_FREQUENCY_HZ
     /1000 * 30)
#else
[...]
#define TIMER3_FREQUENCY_KHZ (tmrHw_HIGH_FREQUENCY_HZ
     /1000)
#endif
```

The variable `tmrHw_HIGH_FREQUENCY_MHZ` is defined in the header file `tmrHw_reg.h` with the value 150,000,000 to denote a frequency of 150 MHz. These timer frequencies are used in the static C99 initialization of the timer `sp804_timer3_clk`:

```
static struct clk sp804_timer3_clk = {
        .name = "sp804-timer-3",
        .type = CLK_TYPE_PRIMARY,
        .mode = CLK_MODE_XTAL,
        .rate_hz = TIMER3_FREQUENCY_KHZ * 1000,
};
```

The problem is that the member `rate_hz`, which has the type `unsigned long` (i.e., 32 bits on this platform), is too small to contain the resulting value of 30 times 150 Mhz in Hertz. We have reported this issue (unnoticed for three years and detected by our tool) to the responsible maintainer, who promptly acknowledged it as a new bug.[4] The point, however, is: This is a configuration-conditional bug that is *easy to detect* at compile time! The GCC compiler correctly reports it (with an integer overflow warning) iff (a) Linux is compiled for a 32-bit platform *and* (b) the Linux configuration happens to include the `#ifdef` block, which, however, is inserted by the CPP only if the `CONFIG_ARCH_FPGA11107` feature flag is set – which in turn depends on several other features.

(2) BUSYBOX is a compile-time tailorable implementation of the UNIX core utilities for memory-constrained environments. It is employed in many wireless routers and DSL modems, but also in several Linux distributions. With VAMPYR, we found several configurations of BUSYBOX, for which the GCC compiler warns about format-string security problems in `coreutils/stat.c`. The upstream developers have confirmed this issue as a new security-relevant bug.[5]

(3) In the L4/FIASCO μ-kernel, the file `ux/main-ux.cpp` revealed a compilation error, as the instantiation of a `Spin_lock` type lacks a type parameter. Again, this is a configuration-dependent bug, which is reported by GCC iff the feature flags `CONFIG_UX` (for choosing Linux user-mode as target architecture) *and* `CONFIG_MP` (for multi-processor support) are both enabled. We have reported this issue (detected by our tool) to the L4/FIASCO developers, who confirmed it as a new bug.

Summary All of the above bugs were caused by subtle issues that are difficult to spot with code reviewing, but easy to detect by a static checker (even the compiler in our case) – if the respective lines of code *are* getting compiled. However, the problematic lines of code are not covered by a standard configuration, which probably is the reason these bugs went unnoticed for up to three years. By increasing the CC, we have found tens of such issues in BUSYBOX and L4/FIASCO and hundreds in Linux (see Section 5).

3 Our Approach

The goal of our approach is, ultimately, to find configuration-conditional bugs by the systematic and automatic increasing of the *configuration coverage* (CC) of static code checkers and other quality measures, such as unit tests. Technically, this is achieved by automatically deriving a (reasonably small) set of configurations that together provide coverage of all configuration-conditional parts of the code. The static checker or unit test driver is then invoked individually for each element of this set.

[4] https://lkml.org/lkml/2012/4/23/229

[5] http://lists.busybox.net/pipermail/busybox/2012-September/078360.html

Hence, existing bug-hunting tools become *configurability-aware* without changing them. For now, we aim for *statement coverage*, that is, we want to make sure that every line of code is checked at least once by the employed static checker. We consider statement coverage as a significant first step to increase CC with modest computational complexity. While algorithms for higher coverage criteria are technically easy to integrate with our approach, we discuss their feasibility in Section 6.2.

The key idea is to extract configuration constraints and use a SAT Checker to construct sets of configurations. The challenge is that static variability is not only implemented by means of the `CPP`, but by a multitude of languages and tools that introduce and constrain variation points at different stages of the build process. In the following, we illustrate this on the example of the Linux generation process.

3.1 Static Variability in Linux

Linux is configured and generated in a sequence of steps that is depicted in Figure 1. Each of these steps effectively constraints the set of static variation points in the subsequent steps:

❶ The first decision is to choose a target architecture. Technically, this is done by setting an environment variable, which, if omitted, leads to native compilation; otherwise KBUILD uses a cross-compiler to produce a kernel for a nonnative architecture.

❷ Depending on the selected architecture, the KCONFIG configuration tool loads a set of `Kconfig` files that together define the configuration space (features and constraints) of the chosen architecture. On Linux/`x86`, for instance, the user can choose from more than 7,700 features. KCONFIG saves the resulting feature selection to a file (`.config`) that is used for storing, loading and interchanging feature selections. The generated artifacts (`.config` and `auto.conf`) control the compilation process in the subsequent steps.

❸ MAKE is used to implement **coarse-grained variability** on a per-file basis. Depending on the configured features, the KBUILD tool selects the subset of all source files that are actually passed to the compiler and linker.

❹ In this subset, the `CPP` representation is used to implement **fine-grained variability** on a sub-file basis. Depending on the configured features (`configure.h`), `#ifdef` blocks are included or excluded by the `CPP` from the token stream passed to the compiler.

❺ Finally, MAKE is also used to derive, depending on the selected features, compiler options and binding units, and thus, to drive the compilation and linking process. The result is a bootable kernel image and the associated *loadable kernel modules* (LKMs) for the chosen architecture and KCONFIG selection.

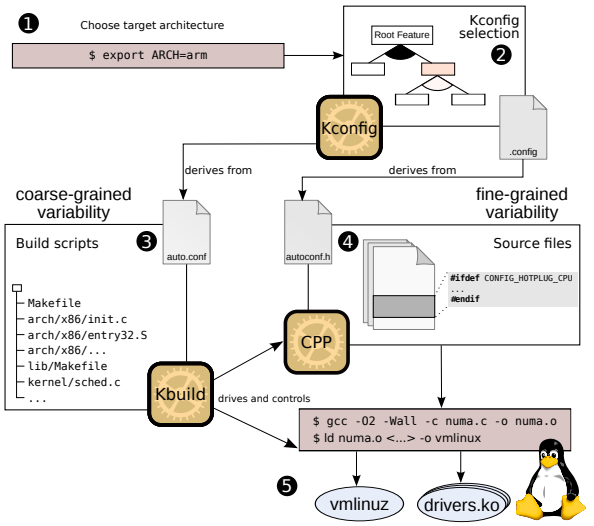

Figure 1: Fine-Grained and Coarse-Grained Variability Implementation in Linux.

Variation points are not only specified on the `CPP` level, but on different levels and in different languages. In fact, each build step (❶–❹) in Figure 1 also constitutes a distinct level of variability implementation, which constrains the effective number of variation points on subsequent levels: The chosen architecture ❶ constrains the possible KCONFIG selection ❷, which in turn constrains the inclusion and exclusion of complete source files (coarse-grained variability) ❸, which further constrains the inclusion and exclusion of `#ifdef` blocks (fine-grained variability) ❹. To derive a concrete configuration that selects a particular `#ifdef` block in some translation unit, the developer basically has to go back all steps of this hierarchy to make sure that all dependencies of the translation unit and block are fulfilled.

The general lesson to be learned is that CC cannot be achieved by looking at the source code alone – all levels of static variability, including the constraints specified in the build system (KBUILD), feature model (KCONFIG) and architecture selection have to be taken into account. This makes it so challenging for developers to manually derive configurations that cover all parts of their code.

3.2 Maximizing Configuration Coverage

The goal of the approach is to find a set of configurations for each source file that, when accumulated, selects all configuration-conditional parts of the code. Analyzing all configurations then maximizes the CC with respect to statement coverage.

Configuration-conditional parts of the code are given as complete files (level ❸, coarse-grained variability) and `#ifdef` blocks (level ❹, fine-grained variability). The resulting set of configurations has to cover both, but we

conceptually operate on the most fine-grained level only: variation points that represent the selection (or deselection) of `#ifdef` blocks. Conditionally compiled files (level ❸) are treated as a single `#ifdef` block that (conceptually) spans the whole file content. Without loss of generality, in the following we therefore use *block* or *`#ifdef` block* as a collective noun for any kind of configuration-conditional variation point.

The reason why a single configuration is not the solution to this problem arises from the fact that blocks and whole source files may be in conflict to each other and can therefore not be enabled by the same configuration. Such conflicts can stem from *all* variability levels ❶–❹ in Figure 1, including the configuration model (KCONFIG) and architecture.

The `CPP`-statements of a C program describe a meta-program that is executed by the C Preprocessor before the actual compilation by the C compiler takes place. In this meta-program, the `CPP` expressions (such as `#ifdef`– `#else`– `#endif`) correspond to the conditions in the edges of a loop-free[6] *control flow graph* (CFG); the thereby controlled `#ifdef` blocks are the statement nodes. On this CFG, established metrics, such as *statement coverage* or *path coverage*, can be applied.

The structure of `CPP` blocks and the identifiers used in their expressions translate into a propositional formula such that each `CPP` identifier and each `#ifdef` block is represented as a propositional variable. For the sake of a uniform treatment of source files with `CPP` blocks, we introduce an artificial `CPP` expression for the top-level block to express the constraints that are imposed by the build-system. For calculating the configurations, the actual block contents, in this case C code, can be ignored. We calculate the *Presence Condition* ($\mathcal{PC}$) for each `#ifdef` block, which here is influenced by three factors (Figure 2): Firstly, by φ_{CPP}, which encodes the structure and semantics of the `CPP` language (level ❹ variability). Secondly, by φ_{KBUILD}, the constraints that are imposed in build-system rules in KBUILD (level ❸ variability). Thirdly, by φ_{KCONFIG}, the constraints that arise from the feature dependencies declared in KCONFIG and by the selected architecture (level ❷ and ❶ variability).

The algorithm to calculate the configurations basically iterates over all blocks and employs a SAT solver to find a configuration that selects the current block. To reduce the number of SAT queries, blocks that are covered in already found configurations are skipped. As a further optimization, the algorithm tries to enable as many blocks as possible simultaneously, which reduces the number of resulting configurations. Our algorithm has a worst-case complexity of n^2 SAT calls for n blocks; however, in practice the number of SAT calls remains in the order of n for the vast majority of files. We discuss this algorithm in earlier work [19] with more detail.

[6]Leaving aside "insane" `CPP` meta-programming techniques based on recursive `#include`, which are not used within Linux.

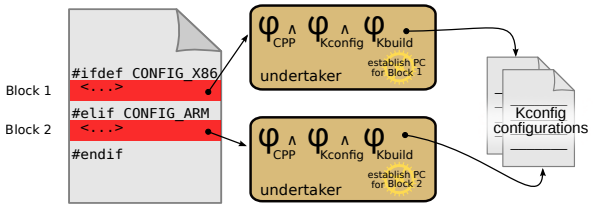

Figure 2: Deriving configurations: For each configuration-conditional block, we establish their $\mathcal{PC}$ to derive a set of configurations that maximize the CC.

3.3 Implementation: The VAMPYR

We provide the VAMPYR tool as an easy-to-use variability-aware driver that orchestrates the concepts and tools outlined in the previous sections. The general interaction between the individual tools is depicted in Figure 3. First, VAMPYR ensures that all variability constraints from `CPP`, KBUILD and KCONFIG are available. This formula is loaded (UNDERTAKER in Figure 3) and used to produce the configurations, on which the tools for static analysis are applied.

Note that the resulting configurations only cover variation points that are included in the source file, which means that they cannot be loaded directly into the KCONFIG configuration tool. Conceptually, we can understand this *partial configuration* as a set of variation points on level ❹ in Figure 1, for which we need to find a sound configuration on level ❷. This means that a produced configuration does not constrain the selection of the remaining thousands of configuration options that need to be set in order to establish a full KCONFIG configuration file that can be shared among developers. These remaining unspecified configuration options can be set to any value as long as they do not conflict with the constraints imposed by the partial configuration.

To derive configurations that can be loaded by KCONFIG, we reuse the KCONFIG tool itself to set the remaining unconstrained configuration options to values that are not in conflict with φ_{KCONFIG}. With these full configurations, we use the KBUILD build system to apply the tools for static analysis on the examined file. So far, we have integrated three different tools for static analysis: GCC, SPARSE, and SPATCH from the COCCINELLE tool-suite [14, 15].

3.4 Application Scenarios

(a) The Linux maintainer for the `bcmring` ARM development board has received a contributed patch via email. She first applies the patch to her local git tree, and

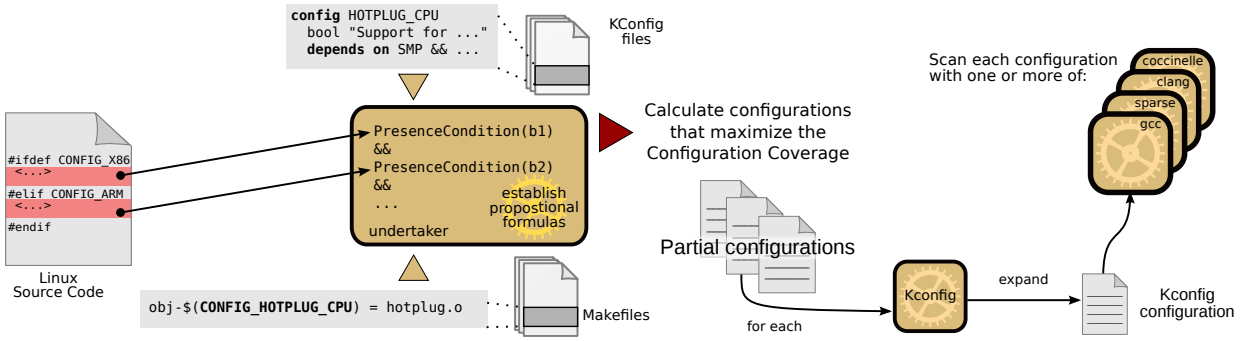

Figure 3: Workflow of the VAMPYR configuration-aware static-analysis tool driver

then runs the VAMPYR tool on all files that the proposed change modifies:

```
$ git am bugfix.diff           # Integrate the patch
     as new commit
$ vampyr -C gcc --commit HEAD  # Examine files of
     latest commit
```

VAMPYR derives a CC-maximizing set for each file modified by the patch. The resulting configurations are plain text files in a syntax that is familiar to Linux developers, but only cover those variation points that are actually related to the $\mathcal{PC}$s contained in the affected source files. VAMPYR utilizes the KCONFIG configuration tool to set all remaining unspecified items to the default values. This expanded configuration is activated in the source tree (i.e., KBUILD updates the force-included `autoconf.h` and `auto.make` files), and GCC, or another static checker, is called on the examined file.

The issued warnings and errors are collected and presented to the developer after VAMPYR has inspected all configurations. The whole process takes less than a minute to complete on a modest quadcore development machine. In this case, VAMPYR reveals the integer overflow bug that has been presented in Section 2.

(b) The same maintainer implements a nightly quality-assurance run. After having integrated the submissions of various contributors, she calls it a day and lets VAMPYR check the complete source code base on Linux/`arm` (a worklist with 11,593 translation units) in a cronjob:

```
$ vampyr -C gcc -b worklist
```

In this case, VAMPYR calculates 14,222 configurations (~1.2 per file) in less than 4 minutes. The actual analysis, which includes extracting the variability from KCONFIG and KBUILD (cf. Section 3.1 and 3.2) and running the compiler and Linux makefiles, takes about 4.5 hours. We present and discuss the summarized findings of such a run in the evaluation (Section 5, Table 2 and 3).

Both application examples show that the approach results in a straight-forward and easy to use tool that unburdens developers from the tedious task of finding (and testing) the relevant configurations manually.

4 Configuration Coverage

As also reported by Palix and colleagues [15], we assume that static checkers and bug-finding tools are generally applied to a single configuration only. This raises the question of how many conditional blocks are commonly left uncovered. To be able to answer this question (and eventually quantify in Section 5 how much VAMPYR can improve on the situation), we first establish in Section 4.1 a metric for the effective CC achieved by a configuration or a set of configurations. We then use this metric in Section 4.2 to calculate the CC of the `allyesconfig` standard configuration in Linux. This synthetic configuration enables as many features as possible and is supposed to cover the maximum amount of code. We use it here to get an *upper bound* of the CC that can be achieved by testing a single configuration only.

4.1 Calculating Configuration Coverage

The definition of the CC depends on the chosen coverage criteria, including *statement coverage* (every block is included at least once), *decision coverage* (every block is included at least once and excluded at least once), and *path coverage* (every possible combination of blocks is included at least once). In this work, we go for *statement coverage* and define the CC of a given configuration as the fraction of the thereby *selected blocks* divided by the number of *available blocks*:

$$CC_S := \frac{\text{selected blocks}}{\text{available blocks}} \quad (1)$$

To calculate CC_S, we need to determine the number of *selected blocks* and the number of *available blocks* given by a configuration. To determine the set of selected blocks, we calculate the $\mathcal{PC}$ for each block b (as described in Section 3.2) to check if the respective block gets enabled by the configuration. For unconditional parts of the code (such as the file `fork.c`, which is included in every Linux configuration), the $\mathcal{PC}$ is a tautology.

Determining the set of available blocks is more complex: Depending on (a) the taken perspective (such as

a concrete architecture) and (b) the constraints specified on each configuration level (❶–❹), many blocks are only *seemingly* configurable. To illustrate this effect, consider the following excerpt from `drivers/net/ethernet/broadcom/tg3.c`:

```
static u32 __devinit tg3_calc_dma_bndry(struct tg3 *
     tp, u32 val)
{
        int goal;
[...]
#if defined(CONFIG_PPC64) || defined(CONFIG_IA64) ||
     defined(CONFIG_PARISC)
        goal = BOUNDARY_MULTI_CACHELINE;
#else
[...]
#endif
```

This code configures architecture-dependent parts of the Broadcom TG3 network device driver with `#ifdef` blocks. Given any concrete architecture (which is the common perspective in Linux development), these blocks are not variable: The $\mathcal{PC}$s of the `#ifdef` blocks will either result in a tautology (on Linux/`IA64`, respectively Linux/`PARISC`) or a contradiction (on all other architectures) for any KCONFIG selection.

So, on level ❹ for Linux/`ia64` and Linux/`parisc` the `#if` part is always selected, whereas for all other architectures the `#else` part is chosen – but this only holds under the assumption, that on level ❸ the `tg3.c` file itself is selected. On Linux/`s390`, for instance, this file is singled out by a MAKE-file constraint; hence, the $\mathcal{PC}$ of *each* block is a contradiction. In line with our terminology from [20], we call a block with a $\mathcal{PC}$ that is a contradiction on the taken perspective, a *dead* block; a block with a $\mathcal{PC}$ that is a tautology is called an *undead* block, respectively. Both play an important role with respect to CC: A dead block cannot be selected by any configuration, whereas an undead block is implicitly selected by every configuration.

The Linux/`s390` example shows that it generally is not obvious from the code if a block is dead/undead: Most of the 12,000 Linux features (`CONFIG_` flags) become a tautology or contradiction because of the constraints expressed on level ❷ or ❸. On Linux/`x86`, for instance, 17 percent of all blocks are only seemingly variable, whereas on Linux/`s390`, this holds for 67 percent. One reason for this high rate is that `s390` hardware does not feature the PCI family of buses, so all PCI-related features, including many device drivers, become contradictions.

The practical consequence is that dead and undead blocks have to be singled out for calculating the configuration coverage. We call this refined metric the *normalized configuration coverage* (CC_N):

$$CC_N := \frac{\text{selected blocks} - \text{undead blocks}}{\text{all blocks} - \text{undead blocks} - \text{dead blocks}} \quad (2)$$

In Table 1, the CC_N is normalized with respect to the selected architecture in Linux, which impacts what blocks have to be considered as dead or undead. For a platform maintainer, for instance, ignoring blocks of a "foreign" architecture makes perfect sense: Linux is generally compiled natively and code parts for other architectures are likely to not compile anyway.

Architecture	Total kLOC	in CPP blocks	# variation points (dead/undead rate)	allyes $\mathcal{CC}_S$	allyes $\mathcal{CC}_N$
x86	8,391	4.5%	30,368 (17%)	65.2%	78.6%
hardware	6,417	3.6%	22,152 (22%)	59.7%	76.8%
software	1,974	7.6%	8,216 (4%)	80.2%	82.7%
arm	8,568	4.6%	33,356 (18%)	49.2%	59.9%
hardware	6,629	3.9%	25,140 (20%)	40.8%	51.2%
software	1,938	6.9%	8,216 (11%)	74.8%	83.6%
mips	7,848	4.3%	30,094 (23%)	42.3%	54.5%
hardware	5,896	3.3%	21,878 (29%)	30.1%	42.1%
software	1,952	7.4%	8,216 (7%)	74.8%	79.8%
s390	2,783	5.7%	28,756 (67%)	24.2%	72.1%
hardware	1,034	2.8%	20,540 (86%)	5.1%	37.2%
software	1,748	7.3%	8,216 (18%)	71.8%	86.8%
...	< *19 further architectures* >				
Mean μ	6,447	3.8%	29,180 (39%)	43.9%	71.9%
Std. Dev. σ	±1,652		±1,159	±11.8%	±12.2%

Table 1: Quantification over variation points across selected architectures in Linux v3.2 and the corresponding $\mathcal{CC}_S$ and $\mathcal{CC}_N$ of `allyesconfig`.

4.2 The Configuration Coverage of 'allyesconfig' in Linux

Table 1 lists configurability-related source code metrics together with the resulting CC_S and CC_N of the `allyesconfig` standard configuration. We have examined these metrics for 24 out of the 27 Linux architectures. Table 1 lists an excerpt of this analysis: selected "typical" architectures (for PCs, embedded systems, mainframes), together with the mean μ and standard deviation σ over all 24 analyzed architectures.[7] We further discriminate the numbers between "hardware related" (originated from the subdirectories `drivers`, `arch`, and `sound`) and "software related" (all others, especially `kernel`, `mm`, `net`).

The average Linux architecture consists of 6,447 kLOC distributed over 8,231 source files, with 3.8% of all code lines in (real) `#ifdef` or `#else` blocks and 29,180 total variation points (`#ifdef` blocks, `#else` blocks, and configuration-dependent files). There is relatively little variance between Linux architectures with respect to these

[7] We could not retrieve results for `um`, `c6x`, and `tile`, which seem to be fundamentally broken. The complete data tables are available as an online appendix: `http://vamos.cs.fau.de/usenix2014-annex.pdf`

simple source-code metrics, as all architectures share a large amount of the source base (including the kernel and device drivers).

For the three rightmost columns, however, the variance is remarkable. The rate of dead/undead variation points varies from 17 percent on Linux/`x86` to up to 67 percent on Linux/`s390` ($\mu = 39\%$). It furthermore is reciprocally correlated to the CC_S of `allyesconfig`: These numbers underline the necessity to normalize the CC_S with respect to the actual viewpoint (here: the architecture). The normalized configuration coverage CC_N is generally much higher ($\mu = 71.9$, $\sigma = 12.2$) – here Linux/`s390` (72.1%) is even above the average and close to Linux/`x86` (78.6%).

The situation is different on typical embedded platforms, where the CC_N of `allyesconfig` is significantly lower: On Linux/`arm`, the largest and most quickly growing platform, only 59.9 percent are covered. These numbers are especially influenced by the relatively low CC_N (51.2%) achieved in the hardware-related parts. We find a similar situation for Linux/`mips`, for which only 54.5 percent are covered. We take this as an indicator for the larger hardware variability typically found on embedded platforms, which manifest in many alternative or conflicting features on the KCONFIG level and in `#else` blocks on the `CPP` level.

5 Evaluation

In the following, we evaluate the benefit of increasing the CC_N with VAMPYR. The working hypothesis is that especially subsystems and architectures with a relatively low CC_N of `allyesconfig` are prone to bugs that in principle are easy to find (reported by the compiler), but remain undetected for several years – like the integer overflow issue from Section 2.

We analyze this hypothesis (i.e., how many additional bugs can be found with our approach) on two operating systems, namely Linux version v3.2 and L4/FIASCO, as well as on BUSYBOX, a versatile user-space implementation of important system level utilities targeted at embedded systems. They all use sufficiently similar versions of KCONFIG, which allows reusing the variability extractor for φ_{KCONFIG} for all projects.

In all cases, we calculate a set of configurations for each file as described in Section 3.3, and apply GCC 4.7 as static checker for each configuration on all files individually. As an optimization, the initial starting set contains the standard configuration `allyesconfig`.

5.1 Application on Linux

On Linux, we use VAMPYR to generate the configurations for the 24 architectures examined in Section 4.2. In comparison to `allyesconfig`, VAMPYR increases the CC_N for every architecture, on average from $\mu = 71.5$ to $\mu = 84.6$ percent. Our Quad-core workstation calculates all partial configurations for 9,300 source files in less than 4 minutes. For the sake of comprehensibility, we limit in the following the in-depth analysis with GCC as static checker to three architectures. We choose, based on the observations in Section 4.2, `x86`, `arm`, and `mips`: We assume Linux/`x86` to be the best tested architecture – because of its maturity, wide-spread application, and the fact that it has the lowest rate of dead/undead blocks (see Table 1). Hence, we expect to find relatively fewer configuration-dependent bugs than in Linux/`arm`, which is the largest (in terms of files and code lines) and, driven by Android, most quickly growing Linux architecture, with a relatively low CC_N. We analyze Linux/`mips` as another embedded platform that is less in a state of flux than Linux/`arm`, but shows an even lower CC_N.

Table 2 depicts the results: Again we list the CC_N and found issues for both the hardware- and software-related parts of Linux. For the three architectures, we notice an increase of the CC_N by 10 to 36 percent, paid by about 20 percent more GCC invocations compared to a regular compilation with a single configuration. So on average, a Linux translation unit requires 1.2 invocations of a static checker to achieve CC with respect to statement coverage.

However, even though VAMPYR does increase the CC_N, it is not increased to 100 percent. We achieve the best result for Linux/`mips` with 91 percent coverage (`allyesconfig`: 55%); for the other two architectures the results are slightly lower. This is caused by (a) deficiencies in the current VAMPYR implementation as well as (b) bugs in the Linux KCONFIG models. We further discuss these issues in Section 6.1.

Nevertheless, VAMPYR reveals a high number of additional GCC messages that are not found with the `allyesconfig` configuration (last column of Table 2): 26 additional messages on Linux/`x86`, 199 on Linux/`arm`, and 91 on Linux/`mips`.

We take the number of `#ifdef` blocks per reported issue (bpi) as a normalization metric for code quality. This confirms our working hypothesis from Section 5: The bpi of Linux/`x86` is 110 , which is the lowest among the examined architectures. It is about 2.4 times better than the bpi of 46 revealed for Linux/`arm`, which is the highest. Linux/`mips` is with an bpi of 85 in between. On all architectures, the hardware-related parts of the source code (`arch`, `drivers`, `sound`) contain significantly more issues than the software-related parts (everything else, especially `kernel`, `mm`, and `net`) – we can confirm the frequent observation that hardware-related code contains significantly more bugs than software-related code [2, 15] also in the context of variability. This holds in particular for the quickly growing `arm` architecture, where the hardware-related parts show 5.6 times more issues than

Software Project	allyesconf CC_N	VAMPYR CC_N	Overhead: increase of GCC Invocations	GCC #warnings VAMPYR (allyesconfig)	GCC #errors VAMPYR (allyesconfig)	Σ Issues	#ifdef blocks per reported issue (bpi)	**Result**: increase of GCC messages
Linux/x86	78.6%	88.4%	21.5%	201 (176)	1 (0)	202	110	26 (+15%)
hardware	76.8%	86.5%	21.0%	180 (155)	1 (0)	181	82	26 (+17%)
software	82.7%	92.4%	22.7%	21 (21)	0 (0)	21	351	0 (+0%)
Linux/arm	59.9%	84.4%	22.7%	417 (294)	92 (15)	508	46	199 (+64%)
hardware	51.2%	80.1%	23.7%	380 (262)	92 (15)	471	34	194 (+70%)
software	83.6%	96.3%	19.5%	37 (32)	0 (0)	37	192	5 (+16%)
Linux/mips	54.5%	90.9%	22.0%	220 (157)	29 (1)	249	85	91 (+58%)
hardware	42.1%	88.2%	21.5%	174 (121)	17 (1)	191	72	69 (+57%)
software	79.8%	96.3%	23.2%	46 (36)	12 (0)	58	128	22 (+61%)
L4/FIASCO	99.1%	99.8%	see text	20 (5)	1 (0)	21	see text	16 (+320%)
Busybox	74.2%	97.3%	60.3%	44 (35)	0 (0)	44	72	9 (+26%)

Table 2: Results of our VAMPYR tool with GCC 4.7 (`-fno-inline-functions-called-once -Wno-unused`) as static checker on Linux v3.2, L4/FIASCO and BUSYBOX.

the software-related parts.

In Table 2 the reported issues are differentiated between errors and warnings. However, this classification by the *compiler* is only seemingly related to the severity of the issue. While many developers consider warnings as more-or-less cosmetic issues, they often point to critical bugs, such as the integer overflow from Section 2. Nevertheless, by the fact that a high number of warnings is also revealed by `allyesconfig`, we have to conclude that at least some warnings are considered as "false positives".

To quantify the actual benefit of our approach in this respect, we have reviewed all messages on Linux/`arm` manually to discriminate less critical messages from real bugs. The results of a conservative classification[8] are depicted in Table 3: 91 out of the 508 reported issues for Linux/`arm` have to be considered as real bugs. For seven bugs, including the issues from Section 2, we have proposed a patch[9] to the upstream developers, which all got immediately confirmed or accepted. Six of these seven bugs had been unnoticed for several years.

Less critical GCC messages	warnings	errors
Σ Less critical messages	347 (223)	16 (0)
Manually validated bugs		
Undeclared types/identifiers		46 (4)
Access to possibly uninitialized data	22 (20)	
Out of bounds array accesses	11 (7)	2 (0)
Bad pointer casts	8 (0)	
Format string warnings	1 (0)	
Integer overflows	1 (0)	
Σ Bugs found	**43 (27)**	**48 (4)**
Σ All reported issues	390 (250)	64 (4)

Table 3: Classification of GCC warnings and errors revealed by the VAMPYR tool on Linux/`arm`. The numbers in parentheses indicate messages that are also found when compiling the configuration `allyesconfig`

5.2 Application on L4/Fiasco

In order to show the general applicability, we apply the VAMPYR tool also to the code base of the L4/FIASCO μ-kernel. Compared to Linux, L4/FIASCO is relatively small: It encompasses about 112 kLOC in 755 files (only counting the core kernel, that is, without user-space packages). Nevertheless, we identify 1,255 variation points (1,228 conditional code blocks and 16 conditionally compiled source files) in the code base.

L4/FIASCO employs the KCONFIG infrastructure to configure 157 features on 4 architectures. Unlike Linux, the architectures are user-selectable KCONFIG options. Also, L4/FIASCO does not only use the `CPP`, but also uses a transformation process that allows programmers to declare interface and implementation in the same source file. This additional processing step produces traditional header and implementation files, which are preprocessed by `CPP` and compiled with GCC. We cope with this by processing the resulting `CPP #ifdef` blocks for calculating the configurations. However, because of the additional preprocessing step, the metrics of GCC invocations per source file and bpi do not relate to the results of Linux and BUSYBOX, so we leave them out in Table 2.

For L4/FIASCO, the VAMPYR tool produces 9 different configurations that in total cover 1,228 out of 1,239 `#ifdef` blocks, which maps to a CC_N of 99.8 percent. Compared to `allyesconfig`, the number of compiler messages thereby increased from 5 to 21, among them

[8] Only messages for which the manual source-code review provides *strong* evidence of an actual bug are counted as such. Everything else is considered to be less critical. We also count some errors (caused by `#error` statements) that point to issues in the KCONFIG model and not in the code as less critical.

[9] `http://vamos.cs.fau.de/usenix2014-annex.pdf`

the compilation error in `ux/main-ux.cpp` we have illustrated in Section 2. We have reported this issue to the L4/FIASCO developers, who confirmed it as a bug.

5.3 Application on Busybox

Another popular software project that makes use of KCONFIG is the BUSYBOX tool suite. The analyzed version `1.20.1` exposes 879 features that allow users to select exactly the amount of functionality that is necessary for a given use case, implemented by 3,316 `#ifdef` blocks and conditionally compiled source files.

For BUSYBOX, VAMPYR increases the number of reported issues from 35 to 44; we have proposed a fix for one of them to the upstream developers, who have confirmed it as a bug and accepted our patch.

6 Discussion

6.1 Threats to Validity – Quality of Results

Is it fair to compare to `allyesconfig`? In this paper we argue that checking a single configuration is not enough and evaluate this by comparing our configuration-aware VAMPYR results against the `allyesconfig` standard configuration, which we assume to achieve the best possible CC for a single configuration.

However, this is not guaranteed, as `allyesconfig` is a synthetic configuration generated by the KCONFIG tool with a simple algorithm: Traverse the feature tree and select each feature that is not in conflict to an already selected feature. The outcome is sensitive to the order of features in the feature tree, hence, does not necessarily include the possible maximum number of features. Also, even if we assume a maximum number of features as the outcome, this does not necessarily imply the largest possible CC, as we might have missed a feature with a highly crosscutting implementation (i.e., a feature that contributes many `#ifdef` blocks) in favor of another feature that contributes just a single variation point. However, features in Linux are generally not very cross cutting: 58 percent of all features in Linux v3.2 are implemented by a single variation point; only 9 percent contribute more than 10 variation points, most of which are architecture-related features that anyway cannot be modified on the KCONFIG level. So, despite these limitations, we consider `allyesconfig` as a realistic upper bound of the CC that could be achieved with a single configuration in practice.

Why not hundred percent coverage? Even though VAMPYR increases the CC significantly, we do not get full coverage. In some cases, the expansion of a partial configuration by KCONFIG results in a full configuration that contradicts the partial configuration. In order to achieve correct results (and in contrast to our in retrospect naïve attempt in [19]), VAMPYR validates the soundness of each configuration after the expansion process: Configurations that no longer contain the relevant features of interest are skipped; the thereby induced `#ifdef` blocks or files are considered as not covered in Table 2, which results in rates that are below hundred percent. We see three major causes for this effect:

(1) One reason for failed expansions are bugs in the Linux variability descriptions (KCONFIG models). Feature dependencies are notoriously hard to get right with KCONFIG, as the KCONFIG tool does not validate the soundness of the models: In feature dependency expressions, the KCONFIG language provides (via the `SELECT` statement) the option to select arbitrary other features; in this case it is in the responsibility of the *developer* to ensure that thereby the configuration remains valid. In practice, this is not always the case and leads to user-selectable configurations that are formally invalid (contain a contradicting feature), but nevertheless "work" for the user. However, as element of a partial configuration derived by VAMPYR, such a contradicting feature usually causes an incorrect expansion.

(2) Bugs in the KCONFIG descriptions can furthermore cause missing of some dead/undead blocks. This directly leads to a lower CC_N, as dead/undead blocks are substracted in denominator of the CC_N (see Equation 2).

(3) Another potential cause for expansion issues is that the VAMPYR implementation, in particular the model extractor for φ_{KCONFIG}, is not yet feature complete. We currently do not correctly handle the situation when some feature depends on the *value* (rather than the mere *selection*) of some other string or integer feature. Luckily, the number of features that employ value tests in their $\mathcal{PC}$ is low in Linux (`mips`: 0.26%, `arm`: 0.28% `x86`: 0.31%; $\mu = 0.4\%$, $\sigma = 0.22$). Nevertheless, this can make the expansion fail, and thus, impact the achieved CC.

Are there false positives/negatives? The expansion issues imply that there is a high probability of false negatives – bugs we miss, because we do not achieve full CC. Nevertheless, our results show that our approach helps to discover a significant number of long-time overlooked bugs in Linux, L4/FIASCO, and BUSYBOX. The point is that VAMPYR is easy to use and does, by construction, not produce any false positives. Hence, the "annoyance factor" is low, which increases the chance of acceptance by system software developers.

6.2 Higher Coverage Criteria

The chosen coverage criterion also implies the existence of false negatives (i.e., undetected issues): The current implementation of VAMPYR achieves *statement coverage*, that is, every configuration-conditional block is included at least once – at the price of 20 percent additional compiler invocations. Would using a higher coverage criterion would reveal more issues?

We are currently experimenting with a VAMPYR prototype that provides *decision coverage*, that is, every block is included and excluded at least once. Technically, this is realized by virtually adding an empty `#else` block to every `#if` block without an `#else` part. Over the three analyzed architectures (Linux/`x86`, Linux/`arm`, Linux/`mips`), the shift from statement to decision coverage increases the number of reported issues by an additional nine percent at the price of fifteen percent more compiler invocations. We consider this as still acceptable for most use-cases.

The next step will be *path coverage*: every possible combination of blocks is included at least once. This has exponential overhead: a source file with n `CONFIG` flags requires up to 2^n configurations (compiler invocations) to achieve path coverage. However, we expect the constraints from the `CPP`, KBUILD, and KCONFIG levels to considerably restrict the number of actually possible configuration. Furthermore, more than 98 percent of all Linux compilation units employ five or less `CONFIG` flags.[10] So in practice, also path coverage may be feasible for large parts of Linux – at least for tasks such as checking a contributed patch (application scenario (a) from Section 3.4), which we consider as the major use case for VAMPYR.

Checking of a complete architecture with path coverage is probably infeasible on the average developer's machine. Heuristic approaches, such as *pairwise testing* [13], may be more promising candidates to achieve higher coverage at acceptable run times. This is a topic of further research. The point here is that we intend VAMPYR to be a tool that developers can use in their regular development cycles, so the run-time caused by algorithmic complexity (another "annoyance factor") has to be limited.

6.3 General Applicability of the Approach

We have evaluated our approach with Linux, L4/FIASCO, and BUSYBOX and GCC as a static checker. However, the approach is as well applicable to other software families and static checkers. To apply VAMPYR to some piece of configurable software, one basically needs extractors for the configuration points and constraints specified on all employed implementation levels of variability. These levels are generally project-specific: In Linux, we have the `ARCH` environment variable, KCONFIG, KBUILD, and `CPP`; in L4/FIASCO we additionally have the custom `preprocess` level. Our approach, however, makes it easy to integrate such custom variability extractors. We also expect a significant amount of reusability: Almost every system software project employs `CPP` to implement variability and the KCONFIG language is adopted by more and more projects as a means to describe the intended configurability.

Static checkers are not always warmly welcomed – many developers are (initially) reluctant to accept their findings [1]. To convincingly illustrate the issue of configuration-dependent bugs, we therefore have chosen GCC as our static checker: The compiler is a "least common denominator" of a bug-finding tool that *has* to be accepted by developers. Nevertheless, VAMPYR can be employed as variability-aware driver for any static checker that drops in as a compiler replacement. We have also tested it successfully with COCCINELLE [14, 15] and SPARSE. With SPARSE, for instance, VAMPYR more than doubles the issues reported for Linux/`arm`: from 9,484 (`allyesconfig`) to 23,964 (VAMPYR). The high number of issues already reported for `allyesconfig`, however, is a strong indicator that the "annoyance factor" of SPARSE is too high to be accepted as a helpful static checker by the Linux developers – even though "Check cleanly with sparse" is an explicit requirement on the Kernel patch submission checklist.

7 Related Work

Despite these apparent acceptance problems by kernel developers, automated bug detection by examining the source code has a long tradition in the systems community. Many approaches have been suggested to extract rules, invariants, specifications, or even misleading source-code comments from the source code or execution traces from Linux [2, 4, 7, 8, 11, 18]. However, it is remarkable that all of these papers seem agnostic to the used configuration and do not even mention what configuration has been analyzed – even though the wide-ranged analysis of feature implementations in system software by Liebig, Kästner, and Apel [9] underlines the impressive amount of `CPP`-based configurability in today's system software. The issue of CC is largely underestimated.

In the verification and validation community, the notion of CC has been defined in a very similar way to this work by Maximoff et al. [12] in the context of NASA spacecrafts. Unlike our work, this work does not create the configurations from the implementation. Instead, the CC assesses the quality of a given set of test cases.

Liebig et al. [10] present a good overview over the current state of the art for variability-aware analysis of software systems, in which the authors identify two major approaches to the problem: Either by making all tools for static analysis configurability-aware, or by improving the effectiveness of the existing tools by a configurability-aware driver by applying the tools using a sample set, that is, a subset of all possible configuration. Our VAMPYR tool is the latter; there are, however, several research groups that attempt the first approach: Kästner et al. [6] propose a technique coined *variability aware parsing*, which basically integrates the `CPP` variability into tools for static analysis and allows variability aware type-checking across all configurations. Gazzillo and Grimm

[10] The extreme corner-cases of the remaining two percent are `sysctl.c` with 59 flags and `sched.c` with 47 flags.

[5] propose a generalized and much better performing configurability-aware parser for C *with* CPP. Their SUPERC basically treats C and CPP together as a single language and thereby could be used as front-end for the implementation of variability-aware checkers. However, for any practical use, both approaches also need further assistance by a model of all variability constraints from other implementation levels, otherwise, type-checking invalid configurations would cause a very high run time and result in a tremendous number of false positives. The big advantage of these approaches is that they achieve path coverage over the CPP meta program. The disadvantages are that they only work on the CPP level and that they cannot be combined with other static checkers. Hence, we consider our VAMPYR approach to be more flexible.

8 Conclusions

System software is typically configured at compile-time to tailor it with respect to the supported application or hardware platform. Linux, for instance, provides in v3.2 more than 12,000 configurable features. This enormous variability imposes great challenges for software testing with respect to *configuration coverage* (CC).

Existing tools for static analyses are configurability-agnostic: Programmers have to manually derive concrete configurations to ensure CC. For this, they do not only have to consider the structure of `#ifdef` blocks in the code, but also the thousands of constraints specified in the build system and feature model. Hence, many easy-to-find bugs are missed, just because they happen to be not revealed by a standard configuration – Linux contains surprisingly much source code that does not even compile.

With our VAMPYR approach and implementation, the necessary configurations can be derived automatically. VAMPYR is easy to integrate into existing tool chains and provides configurability-awareness for arbitrary static checkers. With GCC as a static checker, we have revealed hundreds of configuration-dependent issues in Linux, L4/FIASCO, and BUSYBOX. For Linux/`arm`, we have found 60 new bugs, some of which went unnoticed for six years. We have also found one bug in L4/FIASCO and nine issues in BUSYBOX. For all three projects, the upstream developers have confirmed the reported bugs and accepted our resulting patches.

The tools VAMPYR and UNDERTAKER are available under GPLv3 at `http://vamos.cs.fau.de/`. All raw data is generated by automated experiments to support scientific reproducibility, and can be examined at `http://vamos.cs.fau.de/jenkins` (login: `public`/`i4guest`). The Linux data and a detailed description of our patches is furthermore available as an online-appendix at `http://vamos.cs.fau.de/usenix2014-annex.pdf`.

References

[1] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. “A few billion lines of code later: using static analysis to find bugs in the real world”. In: *CACM* 53 (2).

[2] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. “An empirical study of operating systems errors”. In: *SOSP ’01*.

[3] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “A Robust Approach for Variability Extraction from the Linux Build System”. In: *SPLC ’12*.

[4] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. “Bugs as deviant behavior: a general approach to inferring errors in systems code”. In: *SOSP ’01*.

[5] Paul Gazzillo and Robert Grimm. “SuperC: parsing all of C by taming the preprocessor”. In: *PLDI ’12*.

[6] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. “Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation”. In: *OOPSLA ’11*.

[7] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. “From uncertainty to belief: inferring the specification within”. In: *OSDI ’06*.

[8] Zhenmin Li and Yuanyuan Zhou. “PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code”. In: *ESEC/FSE ’00*.

[9] Jörg Liebig, Christian Kästner, and Sven Apel. “Analyzing the discipline of preprocessor annotations in 30 million lines of C code”. In: *AOSD ’11*.

[10] Jörg Liebig, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. “Scalable Analysis of Variable Software”. In: *ESEC/FSE ’13*.

[11] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics”. In: *ASPLOS ’08*.

[12] J.R. Maximoff, M.D. Trela, D.R. Kuhn, and R. Kacker. “A method for analyzing system state-space coverage within a t-wise testing framework”. In: *4th anual IEEE Systems Conference*.

[13] Sebastian Oster, Florian Markert, and Philipp Ritter. “Automated Incremental Pairwise Testing of Software Product Lines”. In: *SPLC ’10*. Vol. 6287.

[14] Yoann Padioleau, Julia L. Lawall, Gilles Muller, and René Rydhof Hansen. “Documenting and Automating Collateral Evolutions in Linux Device Drivers”. In: *EuroSys ’08*.

[15] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. “Faults in Linux: Ten years later”. In: *ASPLOS ’11*.

[16] Henry Spencer and Gehoff Collyer. “#ifdef Considered Harmful, or Portability Experience With C News”. In: *USENEX ATC ’92*.

[17] Diomidis Spinellis. “A Tale of Four Kernels”. In: *ICSE ’08*.

[18] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. “/*iComment: Bugs or Bad Comments?*/”. In: *SOSP ’07*.

[19] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. “Configuration Coverage in the Analysis of Large-Scale System Software”. In: *PLOS ’11*. DOI: 10.1145/2039239.2039242.

[20] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. “Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem”. In: *EuroSys ’11*. DOI: 10.1145/1966445.1966451.