These numbers are subject to constant growth; they have practically doubled over the last five years [10].

In the case of Linux (2.6.35), the result are more than 84,000 #ifdef-blocks spread over 28,000+ source code artefacts. These numbers are subject to constant growth; they have practically doubled over the last five years [10].

From the maintenance point of view, compile-time configurability imposes big challenges with respect to configuration coverage (CC). With this term we denote the coverage of software quality measures (such as unit tests or the use of static bug-finding tools) with respect to the configuration-conditional parts of the source code.

1.1 Problem Statement

One often hears statements like “we have tested this with Linux, it works.” (Yes, but which configuration?) “Our tool has found 47 bugs in Linux!” (That is impressive, but how much of Linux did you actually analyze?). Even though static configurability by conditional compilation is omnipresent in system software, the community seems to be somewhat agnostic of the CC issue:

1. There is a plethora of literature and tool support (such as gcov) to implement, for instance, unit tests that reach a certain coverage criteria (such as decision coverage: every edge in the control flow graph is taken at least once). However, the available methods and tools generally understand coverage from the compiler’s perspective – they (implicitly) assume that preprocessing (thus, static configuration) has already happened. The problem of CC is completely ignored.

2. Many great papers (e.g., [2, 4, 9]) have been published about applying static bug-finding approaches to Linux and other pieces of system software. In all cases the authors could find (and eventually fix) a significant number of bugs. It is, however, remarkable, that the issue of CC is not mentioned at all in these papers; in most cases the authors do not even state the configuration(s?) they have analyzed.

(1.) has practical implications on the every-day work of software developers: 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, deducing a set of configurations that covers all relevant compilation-conditional parts of the source is sometimes difficult, but always tedious and error-prone task. Our initial findings show that some files (such as kernel/sched_fair.c) require seven different configurations to cover all conditional parts – and this is probably just a lower bound, as we do not yet include header files in our analysis.

With respect to (2.): Private communication with some of the authors revealed that with Linux the common approach is to use a standard configuration (x86 with either defconfig or allyesconfig). This is perfectly acceptable – their goal was to find bugs and they have found bugs [1]. However,
how many additional bugs could possibly be found with full coverage? Given that `allyesconfig` is the de-facto standard for testing things with Linux: What is its actual coverage?

1.2 About this Paper

We think that configurability as a system property is a significant (and widely underestimated) concern for the complete software development cycle. In the VAMOS\(^1\) project, we aim to mitigate this situation by developing new methods and tools to deal with configurability-induced complexity. In a recent paper [10] we have shown that by better tool support, many configurability-related source-code defects (such as dead `#ifdef`-code) and bugs can be found upfront. Our findings led to fixes for 20 new bugs and the removal of 5,000+ superfluous lines of `#ifdef`-code in Linux 2.6.36. The work presented in this paper continues this line of research: We describe an approach to measure the effective CC of Linux configurations. We have extended our undertaker tool to automatically generate Linux configurations for a given source file in order to achieve full CC. In our preliminary results, we thereby achieve a CC of 94 percent, whereas `allyesconfig` reaches a CC of only 78 percent.

2. HOW TO MEASURE CONFIGURATION COVERAGE IN LINUX

Technically, 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 `#ifdef` expressions (such as `#ifdef` – `#else` – `#endif`) correspond to the conditions in the edges of a loop-free\(^2\) control flow graph (CFG); the thereby controlled fragments of C-code (i.e., the bodies of `#ifdef`-blocks) are the statement nodes (Figure 2). On this CFG, established metrics, such as statement coverage or path coverage, can be applied. In this paper, we go for statement coverage and define configuration coverage (CC) as the fraction of selected configuration-conditional blocks divided by the number of available configuration-conditional blocks. (In the longer term, we aim for path coverage; this is more thoroughly discussed in Section 5.)

Not every CPP identifier that is used in some `#ifdef`-expression does actually describe a configuration-conditional block. For instance, the identifiers used for `#include` guards do not contribute to configurability. In order to get a sound analysis of the configuration-controlled CPP-based variability in Linux, we restrict the analysis to those flags that can be controlled by `KCONFIG` (the configuration front-end of Linux). Fortunately, this is relatively easy, as it is a convention in Linux to prefix all configuration-related CPP identifiers with the string `CONFIG_`. We furthermore restrict our initial analysis to one architecture, namely arch-x86. Figure 1 shows the progression of configuration-conditional blocks over the last 6 years of Linux development for this architecture.

Linux ships with a number of predefined default configurations in the source tree. As pointed out in Section 1.1, we assume that tools for static analysis have generally been applied on one of these reference configurations of Linux.

\(^1\)Variability Management in Operating Systems
\(^2\)Leaving aside “insane” CPP meta-programming techniques based on recursive `#include`

Figure 1: Progression of user controlled variability in Linux (arch-x86) in the last six years

(allyesconfig in the best case), ignoring any user-induced variability.

Figure 1 presents the CC of `allyesconfig` over the last six years of Linux kernel development. Because of alternative features (which manifest as `#else-blocks` in the code) generally there is no configuration that selects all configuration-conditional blocks. However, depending on the kernel version, the CC of `allyesconfig` has always been in the range of 60 to 70 percent: This is not too bad – with more and more features being added to Linux, we had expected this number to decrease over time. We also had expected it to be somewhat lower in general. Still, when assuming that most Linux kernels are tested with a configuration based on `allyesconfig`, code that is not covered by this configuration is generally less tested and therefore is more likely to contain bugs.

The numbers presented so far, however, do not represent the full picture in two respects:

Firstly, as we have shown in previous work [10], not every seemingly configuration-conditional block is actually configuration-conditional. The Linux source code contains many (thousands!) `dead blocks` – conditional blocks that are never selected, as their presence condition contains a contradiction – either within the source code itself (such as an `#ifdef CONFIG_FOO` block nested into an `#ifdef CONFIG_FOO` block) or in conjunction with the constraints defined by the `KCONFIG`-models. These “technically unreachable” blocks should not be considered with respect to CC. Luckily, our undertaker tool [10] can be employed to detect global and architecture-specific dead blocks. When applying our tools to Linux 2.6.35, we observe that the actual CC of `allyesconfig` increases to nearly 78 percent.

Secondly, we do not yet analyze the variability brought in by the header files. The effects of this limitation are difficult to judge (we are working on this), however, it might be significant. There is a coding guideline that variability should be implemented as far as possible in the headers. Instead of polluting the implementation with repetitions of the same `#ifdef`-block again and again, developers are encouraged to put this code into a conditionally defined macro. If this approach is used for the implementation of many alternative features, the actual CC of `allyesconfig` will be lower.

In any case, however, higher or even full CC is desirable. In the following section we discuss how to efficiently create a set of configurations in order to achieve full CC.
3. APPROACH

We represent the structure of CPP blocks in a tree-like graph with two types of edges: nesting of blocks and `if`-`else` cascades. Figure 2 illustrates this representation.

The goal of the approach is to find a set of configurations that maximizes the CC with respect to statement coverage. Since a thorough code test requires all blocks to be selected at least once, ideally, a minimal set of configurations is used in order to reduce testing effort. Note that the minimal set is not well-defined; there can be more than one minimal set of configurations for a given source file.

The minimal amount of configurations that when accumulated select all blocks could be calculated by solving the graph coloring problem on the "conflict graph": All nodes represent a conditional block. When two blocks cannot be selected at the same time (e.g., because of the CPP structure or constraints from the KCONFIG model), an edge between the blocks is added that represents the conflict. The minimal number of configurations is the number of colors required so that that two adjacent nodes have different colors. This is NP hard and, hence, infeasible to be applied in a general-purpose development tool that scales to the size of Linux. Therefore, we seek for heuristics to get a suboptimal, but yet sufficiently useful solution for real-world use-cases.

The structure of CPP blocks and the identifiers used in their expressions are translated into a propositional formula such that each CPP identifier and each conditional block is represented as a propositional variable. For each conditional block, we call the condition under which the block is selected the presence condition of the block. The presence condition is influenced by two factors: First, the structure and semantics of the CPP language impose constraints on the presence condition of each block. Second, extra constraints apply that come from the constraints as expressed by the KCONFIG models. In order to incorporate these extra constraints, we reuse the model extractors from [10]. In the following algorithms, the presence condition of a given block is determined by the function $PC(b)$. The function $Phi$ is the formula that conjugates the presence conditions of all conditional blocks given as input. Conjugated with a block $b$, this formula determines a valid configuration of blocks and $CONFIG_{identifiers}$ that select the block $b$.

On this basis, we have developed two algorithms that calculate a set of configurations that (accumulated) selects all blocks:

The first algorithm is depicted in Figure 3. The general idea is to iterate over all blocks (Line 5) and use a SAT solver to generate a configuration that selects the current block (Line 7). As the most expensive operation is the number of SAT queries, covered blocks in already found configurations are skipped (Line 6). The set $B$ collects the already covered blocks (Line 11). The resulting set $R$ contains the found configurations. This algorithm therefore requires $n$ SAT calls in the worst case.

As a further optimization, the SAT solver is tweaked to try to enable as many propositional variables as possible while looking for satisfying assignments for the formula, which increases the amount of selected blocks per configuration.

On the basis of this simple algorithm, Figure 4 shows an improved version of the approach that more aggressively minimizes the number of configurations and hence, improves the quality of the results. Here, the inner loop (Line 7) collects as many blocks as possible to the working set $WS$ so that there is a configuration that covers all blocks in the working set. Blocks that obviously conflict with some other block of the working set, such as $else$ blocks of an already selected $if$ block, are skipped in Line 9. Line 10 verifies that there is configuration, such that all blocks of the current working set and the current block are selected. Otherwise the block is skipped. For the found working set of “compatible” blocks, a configuration is calculated similarly to the simple variant (Line 19) and added to the resulting set $R$ (Line 21). This results in $n^2$ SAT calls for the worst case.

4. RESULTS

We have implemented both algorithms in the UNDERTAKER toolchain\(^3\) as an additional command-line option. An enhanced version of the model extractor from [10] extracts the constraints from KCONFIG. As SAT solver, the picosat

\(^3\)http://vamos.informatik.uni-erlangen.de
package is used. Given a single source file as input, undertaker calculates (partial) configurations that maximize the configuration coverage for this specific file. This allows programmers participating in large projects, such as the Linux kernel, to focus on the specific parts of the kernel that they currently work on.

We run a preliminary prototype of our tool chain to analyze CC of the Linux v2.6.35 kernel. The whole experiment was done under consideration of the constraints of the x86 architecture. Table 1 summarizes the results. Out of 10,365 .c files, we identify 3,163 files that contain configuration-conditionable blocks. On these files, our tool calculates in total 4,505 (naïve version) and 4,435 (greedy version) configurations. Generating the configurations took about 5 minutes for the naïve version and about 15 minutes for the greedy version of the algorithm on a standard quadcore workstation (Intel Core 2, 2.83 Ghz). However, all times where measured in single-thread operation.

The generated configurations do not include selections for items without constraints on the current file. For full configurations, the resulting partial configurations need to be supplemented with the remaining items. The selection of the remaining items is irrelevant. For this expansion process, we reuse the KCONFIG infrastructure. In practice, this process is cumbersome as KCONFIG in some cases silently overwrites user choices instead of signaling incompatible item selections. Due to technicalities of our approximations of the KCONFIG model [12], our preliminary implementation fails to process some files (<7%). The problematic files tend to contain complicated CPP expressions on which the allyesconfig coverage resulted in a coverage of only 50 percent.

Overall, the CC of allyesconfig is 70 percent. With our approach, the resulting configurations increase the coverage to 84 percent.

These numbers, however, do not take dead blocks into account: As described in Section 2, a considerable amount of blocks cannot be selected by any configuration, which skews the comparison. Therefore, we used the undertaker tool to detect such dead blocks and subtract them from the total number of selectable blocks. While this increases the CC in both cases, we see that in this comparison, our approach achieves 1.2 times better coverage (78.49% to 94.40%).

5. DISCUSSION

Header-induced Variability.

Even though our current implementation reaches a significantly higher CC than allyesconfig, it fails to cover a remaining six percent of valid configuration-conditionable blocks. One reason for this is that our current implementation does not take constraints into account that derive from #define and #undef statements in header files. In order to integrate these extra constraints, a sound implementation of a partial preprocessor such as [3] or [11] would be necessary. We are currently looking into integrating such techniques in our approach. First quick experiments with [3] indicate that with a partial expansion of header-induced macros, our approach achieves up to 99 percent coverage and is 2.5 times better than allyesconfig.
Comparison of the Algorithms.

The histograms in Figure 6 show that for more than 90 percent of the files in Linux CC can be reached by at most two configurations. We can therefore conclude that the implemented heuristics yield reasonably small solutions. In direct comparison, the greedy algorithm produced about 1.5 percent fewer configurations than the naïve variant in total, but takes more than twice the run time, which is not worth the effort in the general case. However, whereas for the greedy variant, no file requires more than 7 configurations to reach CC, the naïve variant requires up to 16 configurations in a few cases. Hence, it would be worthwhile to always start with the naïve algorithm and fall back to the greedy algorithm only when the resulting number of configurations passes by a certain threshold (e.g. five).

Coverage Criteria.

The approach presented in this paper generates a set of configuration that achieve statement coverage. While many syntactical and semantical problems can already be found by statement coverage, some issues might still be missed. For instance, consider two conditional blocks, one containing a function definition and the other a corresponding function instance, consider two conditional blocks, one containing a certain threshold (e.g. five).

6. SUMMARY AND FUTURE WORK

We have introduced and discussed the problem of configuration coverage (CC) in preprocessor-configured system software. CC is defined as the fraction of conditional blocks that can be selected in a source file. An analysis on the Linux kernel shows that the generally applied `allyesconfig` configuration selects only 78 percent of all configuration-conditional `#ifdef` blocks.

As the remaining code is likely to be less tested, we propose an approach that efficiently calculates a set of configurations to reach (nearly) full coverage.

In the future, these resulting configurations can be used to systematically apply static analysis tools on code that is impossible to analyze with a single configuration. We invite other researchers to suggest suitable tools to verify the hypothesis that such code tends to contain more bugs.

References


