Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Revealing and Repairing Configuration Inconsistencies in Large-Scale System Software

Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, Daniel Lohmann*

Friedrich–Alexander University Erlangen–Nuremberg

Received: date / Revised version: date

Abstract. System software typically offers a large amount of compile-time options and variability. A good example is the Linux kernel, which provides more than 10,000 configurable features, growing rapidly. This allows users to tailor it with respect to a broad range of supported hardware architectures and application domains.

From the maintenance point of view, compile-time configurability poses big challenges. The configuration model (the selectable features and their constraints as presented to the user) and the configurability that is actually implemented in the code have to be kept in sync, which, if performed manually, is a tedious and error-prone task. In the case of Linux, this has led to numerous defects in the source code, many of which are actual bugs.

In order to ensure consistency between the variability expressed in the code and the configuration models, we propose an approach that extracts variability from both into propositional logic. This reveals inconsistencies between variability as expressed by the C Preprocessor (`CPP`) and an explicit variability model, which manifest themselves in seemingly conditional code that is in fact unconditional. We evaluate our approach with the Linux, for which our tool detects 1,776 configurability defects, which turned out as dead/superfluous source code and bugs. Our findings have led to numerous source-code improvements and bug fixes in Linux: 123 patches (49 merged) fix 364 defects, 147 of which have been confirmed by the corresponding Linux developers and 20 as fixing a previously unknown bug.

* This work was partly supported by the German Research Foundation (DFG) under grant no. SCHR 603/7-1 and SFB/TR 89.

1 Introduction

> *I know of no feature that is always needed. When we say that two functions are almost always used together, we should remember that "almost" is a euphemism for "not".* *David L. Parnas* [1979]

Serving no user value on its own, system software has always been "caught between a rock and a hard place". As a link between hardware and applications, system software is faced with the requirement for variability to meet the specific demands of both. This is particularly true for operating systems, which ideally should be tailorable for domains ranging from small, resource-constrained embedded systems over network appliances and interactive workstations up to mainframe servers. As a result, many operating systems are provided as a software family [35]; they can (and have to) be configured at compile time to derive a concrete operating-system variant.

Configurability as a system property includes two separate – but related – aspects: *implementation* and *configuration*. Kernel developers implement configurability in the code; in most cases they do this by means of conditional compilation and the C preprocessor [42], despite all the disadvantages with respect to understandability and maintainability ("`#ifdef` hell") this approach is known for [41, 27]. Users configure the operating system to derive a concrete variant that fits their purposes. In simple cases they have to do this by (un-)commenting `#define` directives in some global `configure.h` file; however, many operating systems today come with an interactive configuration tool. Based on an internal model of features and constraints, this tool guides the user through the configuration process by a hierarchical / topic-oriented view on the available features. In fact, it performs implicit consistency checks with respect to the selected features, so that the outcome is always a valid configuration that represents a viable variant. In today's operating systems,

this extra guidance is crucial because of the sheer enormity of available features: eCos, for instance, provides more than 700 features, which are configured with (and checked by) eCosConfig [30]; the Linux kernel is configured with Kconfig and provides more than 10,000 (!) features. This is a lot of variability – and, as we show in this paper, the source of many bugs that could easily be avoided by better tool support.

Our Contributions

This article is an extended and revised version of a previous conference paper [44]. In [40], we introduced the extraction of a source-code variability model from CPP-based software, which represents a building block for this work. A short summary of this approach is presented in Section 3. In [44], we extended that work by incorporating other sources of variability and automatically (cross-) checking them for configurability-related implementation defects in large-scale configurable system software. These extensions enabled us to reveal and repair many configuration issues in the Linux kernel. However, that approach had some shortcomings like dealing with only a subset of the CPP directives, using a limited subset of Kconfig constraints, and generating very large logic formulas for some complex cases. This work addresses these issues by making the following contributions:

1. We show that the increasing amount of configurability in system software causes serious maintenance issues. (Section 2.2)
2. We presents a new approach to extract the constraints from Kconfig variability models. The presented tool deals with all Kconfig constraints and translates them to propositional logic. (Section 4.1)
3. The presented tool checks for configurability-related implementation defects under the consideration of both symbolic *and* logic integrity. (Section 5.2)
4. This work presents a practical and scalable tool chain that has detected 1,776 configurability-related defects and bugs in Linux 2.6.35; for 121 of these defects (among them 22 confirmed new bugs) our fixes have already been merged into the mainline tree. (Section 5.3)

This revision extends the conference version [44] by the following contributions:

1. We show a new approach to translating CPP directives into propositional formula that deals with the CPP directives `#define` and `#undef`. (Section 3.2)
2. We present an algorithm that reduces the size of the propositional formulas, which represent the variability of conditional blocks in source code that uses the CPP. This optimization allows us to apply our approach to very large source files and whole compilation units. (Section 3.3)

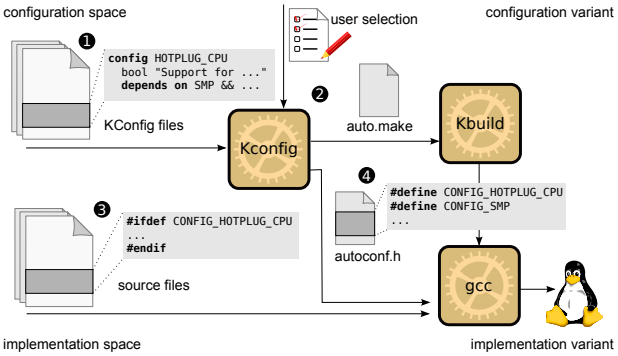

Fig. 1. Linux build process (simplified).

In the following, we first analyze the problem in further detail before presenting our approach in Section 5. We evaluate and discuss our approach in Section 5.3 and Section 6, respectively, and discuss related work in Section 7. The problem of configurability-related defects will be introduced in the context of Linux, which will also be the case study used throughout this paper. Our findings and suggestions, however, should also apply to other compile-time configurable system software.

2 Problem Analysis

Linux today provides more than 10,000 configurable features, which is a lot of variability with respect to hardware platforms and application domains. The possibility to leave out functionality that is not needed (such as x86 PAE support in an Atom-based embedded system) and to choose between alternatives for those features that are needed (such as the default IO scheduler to use) is an important factor for its ongoing success in so many different application and hardware domains.

2.1 Configurability in Linux

The enormous degree of configurability of the Linux kernel demands dedicated means and tools to ensure the validity of the resulting Linux variants. Most features are not self-contained; instead, their possible inclusion is constrained by the presence or absence of other features, which in turn impose constraints on further features, and so on. In Linux, variant validity is taken care of by the Kconfig configuration tool and the Kbuild build system. Figure 1 shows how the relevant tools for configuring and compiling a Linux kernel interact with each other. The four relevant steps are explained in the following:

❶ Linux employs the Kconfig language to specify its configurable features together with their constraints. In version 2.6.35 a total of 761 *Kconfig files* with 110,005 lines of code define 11,283 features plus dependencies. We call this variability specification the

Linux **configuration space**. Section 4.1 gives a brief overview over the KCONFIG language.

❷ The KCONFIG configuration tool implicitly enforces all feature constraints during the interactive feature selection process. The outcome is, by construction, the description of a valid Linux **configuration variant**. Technically, the KCONFIG tool produces a C header file (`autoconf.h`) and a Makefile (`auto.make`) that define a `CONFIG_<FEATURE>` preprocessor macro and a MAKE variable for every selected KCONFIG feature:

```
#define CONFIG_HOTPLUG_CPU 1
#define CONFIG_SMP 1
```

By convention *all* and *only* KCONFIG flags are prefixed with `CONFIG_`.

❸ Features are implemented in the Linux source code. Whereas coarse-grained features are enforced by including or excluding whole compilation units in the build process, in this work we focus on features that are enforced within the source files by means of conditional compilation with the C preprocessor. A total of 27,166 source files contain 82,116 `#ifdef` blocks. We call this variability implementation the Linux **implementation space**.

❹ The KBUILD utility drives the actual variant compilation and linking process by evaluating `auto.make` and embedding the configuration variant definition `autoconf.h` into every compilation unit via the GCC `--force-include` switch.[1] mechanism. The result of this process is a concrete Linux **implementation variant**

2.2 The Issue

Overall, the configurability of Linux is defined by two separate, but related models: The configuration space defines the *intended* variability, whereas the implementation space defines the *implemented* variability of Linux. Given the size of both spaces – 110 KLOC for the configuration space and 12 MLOC for the implementation space in Linux 2.6.35 –, it is not hard to imagine that this is prone to inconsistencies, which manifest as configurability defects, many of which are bugs. We have identified two types of integrity issues, namely *symbolic* integrity issues and *logic* integrity issues, which we introduce in the following by examples from Linux.

Consider the following change, which corrects a simple feature misnaming (detected by our tool and confirmed as a bug) in the file `kernel/smp.c`:[2]

```
diff --git a/kernel/smp.c b/kernel/smp.c
--- a/kernel/smp.c
+++ b/kernel/smp.c
-#ifdef CONFIG_CPU_HOTPLUG
+#ifdef CONFIG_HOTPLUG_CPU
```

Patch 1. Fix for a symbolic defect

The issue, which was present in Linux 2.6.30, is an example of a **symbolic integrity violation**; the implementation space references a feature that does not exist in the configuration space, so the actual implementation of the `HOTPLUG_CPU` feature is incomplete. This bug remained undetected in the kernel code base for more than six months. We cannot claim credit for detecting this particular bug (it had been reported to the respective developer just before we submitted our patch); however, we have found 116 similar defects caused by symbolic integrity violation that have been confirmed as *new*.

A symbolic integrity violation indicates a configuration–implementation space mismatch with respect to a feature *identifier*. However, consistency issues also occur at the level of feature *constraints*. Consider the following fix, which fixes what we call a **logic integrity violation**:

```
diff --git a/arch/x86/include/asm/mmzone_32.h
           b/arch/x86/include/asm/mmzone_32.h
--- a/arch/x86/include/asm/mmzone_32.h
+++ b/arch/x86/include/asm/mmzone_32.h
@@ -61,11 +61,7 @@ extern s8 physnode_map[];

 static inline int pfn_to_nid(unsigned long pfn)
 {
-#ifdef CONFIG_NUMA
    return((int) physnode_map[(pfn)
              / PAGES_PER_ELEMENT]);
-#else
-   return 0;
-#endif
 }

 /*
```

Patch 2. Fix for a logical defect

The patch itself does not look too complicated – the particularities of the issue it fixes stem from the context; in the source, the affected `pfn_to_nid()` function is nested within a larger code block whose presence condition is `#ifdef CONFIG_DISCONTIGMEM`. According to the KCONFIG model, however, the `DISCONTIGMEM` feature *depends on* the `NUMA` feature, which means that it also implies the selection of `NUMA` in any valid configuration. As a consequence, the `#ifdef CONFIG_NUMA` is superfluous; the `#else` branch is dead and both are removed by the patch. The patch has been confirmed as fixing a *new* defect by the respective Linux developers and is currently processed upstream for final acceptance into mainline Linux.

Compared to symbolic integrity violations, logic integrity violations are generally much more difficult to

[1] implemented by the `-include` command-line switch

[2] Shown in unified diff format. Lines starting with -/+ are being removed/added

version	features	#ifdef blocks	source files
2.6.12 (2005)	5338	57078	15219
2.6.20	7059	62873	18513
2.6.25	8394	67972	20609
2.6.30	9570	79154	23960
2.6.35 (2010)	11223	84150	28598
relative growth (5 years)	**110%**	**47%**	**88%**

Table 1. Growth of configurability in Linux

analyze and fix; our experience from submitting patches to Linux developers show that most symbolic violations are caused by misspellings or oversight. So far we have fixed 38 logic integrity violations that have been confirmed as new defects.

Note that Patch 2 does not fix a real bug – it only improves the source-code quality of Linux by removing some dead code and superfluous `#ifdef` statements. Some readers might consider this as "less relevant cosmetical improvement"; however, such "cruft" (especially if it contributes to "`#ifdef` hell") causes long-term maintenance costs and impedes the general accessibility of the source.

2.3 Problem Summary

Overall, we find 1,316 symbolic + 460 logic integrity violations in Linux 2.6.35 – numbers that speak for themselves. The situation becomes more severe every day, given how quickly Linux is growing: Within the last five years, the number of configuration-conditional blocks in the source (`#if` blocks that test for some KCONFIG item) has grown by around fifty percent, the number of features (KCONFIG items) and source files have practically doubled (Table 1).

We think that configurability as a system property has to be seen as a significant (and so far underestimated) cause of software defects in its own respect.

3 Extraction of Variability Models from Source Code

The basic idea of our approach is to extract variability from both the source code and the variability model and translate them into comparable models. The choice of propositional logic as "lingua franca" for the models allows us to use standard tools like SAT-solvers for finding inconsistencies. A first description of our approach for extracting variability from source code has been presented earlier in a conference paper [40]. This section reiterates the fundamentals and enhances the extraction tool by allowing the `CPP` directives `#define` and `#undef` to appear at any place in the source code.

3.1 General Approach

For the purposes of this article, we do not need to consider each and every language feature of `CPP`, but can focus on the features that implement conditional compilation. Therefore we have designed an algorithm [40] that is tailored in this respect in order to be precise and have good performance. In short, our algorithm processes the structure and expressions of `CPP` conditional blocks and produces a boolean formula that describes a source file by means of its conditional compilation structures. In essence, the source code is examined for lines that start with one of the directives `#ifdef, #ifndef, #if, #elif, #else`, which are the constructs responsible for conditional compilation. As result, we receive a formula that describes the presence conditions for each conditional block. It thereby includes all flags (features) that appear in any conditional compilation expression as a propositional variable. We build the presence condition $\mathcal{PC}$ of the conditional block b_i as follows:

$$\mathcal{PC}(b_i) = expr(b_i) \wedge noPredecessors(b_i) \wedge parent(b_i) \quad (1)$$

Let b_i be a conditional block that is nested in some other conditional block and in the middle of a squence of `#elif` blocks. The C compiler will process this block b_i only if a) its expression $expr(b_i)$ evaluates to true, b) none of its predecessors are selected $noPredecessors(b_i)$, and c) its enclosing `#ifdef` block $parent(b_i)$ is also processed. If all these conditions are met, then the `CPP` will necessarily select this block. Additionally, the reverse is true as well: if the `CPP` selects the block, all these presence conditions need to hold. This results in a biimplication: $b_i \leftrightarrow \mathcal{PC}(b_i)$. The formula for a whole file is constructed by conjuncting the presence conditions for all blocks:

$$\mathcal{F}(\mathbf{f}, \mathbf{b}) = \bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i) \quad (2)$$

In this equation, **f** and **b** are bitvectors where each bit in **f** represents a `CPP` flag and each bit in **b** represents the selection a block of in the input.

An example is shown on the right hand side of Figure 7 on page 11: in the upper part we show the source code, and in the lower part we show the generated formula by our algorithm; note that in this example $\mathcal{F}_u(\texttt{[DISCONTIGMEM, NUMA]}, \texttt{[b}_1\texttt{, b}_2\texttt{, b}_3\texttt{]}) = \mathcal{PC}(\texttt{b}_1) \wedge \mathcal{PC}(\texttt{b}_2) \wedge \mathcal{PC}(\texttt{b}_3) = \mathcal{I}$

3.2 Translating `CPP` (re)definition statements into propositional logic

Real-world software-projects often use a generated, central configuration header (like `config.h`), in which the configuration is represented with `#define` statements. This file is then available in all compilation units by either explicitly using the `#include` statement, or implicitly with the `-include` compiler switch. Unfortunately, in C this is not the only way to define `CPP` configuration variables. In many software projects, we observe that

configuration variables are also *overridden* in implementation files using the keywords `#define` and `#undef`.

The algorithm presented in the previous section handles efficiently a subset of the `CPP` constructs. It was applied to real-world projects and enabled us to reveal serious bugs in very large code bases [40, 44]. However, in these two publications the `#define` and `#undef` statements were left unconsidered. This limits the usefulness of the tools on projects that allow the definition of features in the solution space (source code), because this limitation causes inaccurate presence conditions for blocks, whose presence condition is dependent on an earlier `#define` or `#undef` statement. This subsection extends the previous work so that the propositional formulas capture the constraints induced by `#define` and `#undef` statements in `CPP` source code in a correct way.

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 `CPP` expressions (such as `#ifdef` – `#else` – `#endif`) correspond to the conditions on the edges of a loop-free[3] control flow graph (CFG); the thereby controlled fragments of C-code (i.e., the bodies of `#ifdef`-blocks) are the statement nodes. The major challenge for encoding `#define` and `#undef` statements is that these redefinition statements change the *state* of a configuration variable. As propositional logic lacks the concept of state and control flow, we address this by identifying regions, in which the value of the conditional variable is invariant. This means that every redefinition statement introduces a new region that starts at the point of the redefinition statement and renames all dependent propositional variables. For the following example, the identified regions are depicted in Figure 2:

```
1  #if X // Block 0
2  #define A
3  #endif
4
5  #if Y // Block 1
6  #undef A
7  #endif
8
9  #if A // Block 2
10 #endif
```

Each time a redefinition statement introduces a new region, each variable (e.g., `A`) is rewritten to a new variable (e.g., `A'`). The remainder of this section explains how to express the logical relationships between the original conditional variable and its rewritten version (e.g., `A` and `A'`), which are added as additional conjunction to the existing presence conditions.

For the given example, the challenge for the resulting propositional formulas is that the `#define` statement in Line 2 and the `#undef` statement in Line 6 are effective only under the presence conditions of Block 0 and Block 1:

$$
\begin{array}{ll}
1 & (\ \mathrm{B0} \leftrightarrow \mathrm{X}\ ) \wedge \\
2 & (\ \mathrm{B1} \leftrightarrow \mathrm{Y}\ ) \wedge \\
3 & (\ \mathrm{B2} \leftrightarrow A''\ ) \wedge \\
4 & (\ \mathrm{B0} \rightarrow A'\ ) \wedge \\
5 & (\ \neg\mathrm{B0} \rightarrow (\ A \leftrightarrow A'\ ))\ \wedge \\
6 & (\ \mathrm{B1} \rightarrow \neg A''\ ) \wedge \\
7 & (\ \neg\mathrm{B1} \rightarrow (\ A' \leftrightarrow A''\ ))
\end{array}
$$

Line 1 to Line 3 represent the "basic" presence conditions, which are derived directly from the `#ifdef` expressions for this simple example. In Line 3 the conditional variable is already rewritten to A''. Lines 4 to 7 contain implications that control the relationships between the original and rewritten versions of a conditional variable. For both Block 1 and Block 2, two implications (Lines 4 and 5 for Block 1, and Lines 6 and 7 for Block 2) describe the situation with respect to the selection of the block: If it is selected, then the rewritten version of the conditional variable is set to true—if not, then both versions are equivalent.

Generally speaking, these "rewriting" rules constitute an extension to the presence condition of Formula 1 from the previous subsection: Here, the helper function $expr(b_i)$ is extended to substitute all propositional variables with the rewritten version that is active in the "region" of the corresponding conditional block. During the rewriting, the additional implications are conjuncted to $\mathcal{F}_u$. The resulting formula then correctly represents the implementation variability model [40] of `CPP` programs that may include the statements `#define` and `#undef`.

Experimental validation. Since the resulting propositional formulas of real-world source files (like whole driver implementations in Linux) and compilation units are too complex to validate manually, we have to automatically validate the formulas against the actual semantics of `CPP`. Therefore, we use artificial examples as test cases for validation, so that we can apply our approach on real source code from Linux with confidence. For each file, we collect all used configuration flags and generate all possible selec-

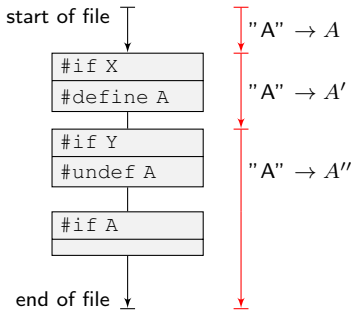

Fig. 2. A file with three `#ifdef` blocks. The upper part of each block contains the `#ifdef` expression, the lower part `#define` and `#undef` statements in the block. The basic idea for encoding a flow of `#define` and `#undef` statements into propositional formula is to introduce "scopes". All references to a variable (A) that occur after an `#define` or `#undef` statements are rewritten.

[3] Leaving aside "insane" `CPP` meta-programming techniques based on recursive `#include`, which are hardly used in practice.

tions of them. For each combination of flags, we call CPP[4] and record the thereby selected blocks. As a result, we have empirically determined the set of configuration flags and corresponding "valid" block selections, according to the semantics of CPP. Respectively, we call all block selections that are not covered by any combination of CPP flags "invalid". With this as reference, we generate the propositional formula with our approach and constrain it with every possible permutation of block selections (e.g., $B1 \wedge \neg B2 \wedge B3$). We then check the validity of each formula with a SAT solver.

While the complexity of the empircal validation is exponential with the number of conditional blocks and CPP flags, it is still a valuable asset for developing and verifying that the generated formulas represent the CPP semantics correctly.

Results. We show the usefulness of the presented extension of the algorithm by analyzing Linux version 3.0 for configuration defects in form of "dead" and "undead" Blocks [45], which are basically blocks that are only seemingly conditional but in fact cannot be selected or unselected under any configuration. The analysis is preformed using the undertaker tool that will be presented in detail in Section 5.3. In total, the tool processed 15,538 header and 16,202 implementation files. When ignoring every redefinition statement (`#define` and `#undef`), in total 76 defects (23 *dead* and 46 *undead* blocks) are detected. Now, by properly respecting redefinition statements, the number of defects increases to an impressive total of 2,514 defects (2182 *dead* and 332 *undead* blocks). The additional defects were confirmed by manual random verification.

3.3 Application on Large Compilation Units

The presented extension in the previous section, which allows to model the variability of CPP meta-programs with propositional logic, adds additional conjunctions to each block presence condition that follows a CPP redefinition statement (i.e., `#define` or `#undef`). For real-world software-projects such as the Linux kernel, this can be problematic because such redefinition statements do not necessarily occur in the same source file.

Figure 3 shows a Linux driver, which consists of a header file `driver.h` and an implementation file `driver.c`. The `#define` overrides the configuration variable `CONFIG_DRIVER_USE_NUMA` only when the variable `CONFIG_NUMA` is set. While in theory, this example could have also been handled within KCONFIG, in practice this is not always the case. Therefore, for including these non-KCONFIG based extra constraints the analysis needs to additionally process `#include` statements instead of analyzing header and implementation files separately and individually.[5]

```
#ifdef CONFIG_NUMA
#define CONFIG_DRIVER_USE_NUMA 1
#endif
```

(a) File: driver.h

```
#include "driver.h"
#ifdef CONFIG_DRIVER_USE_NUMA
\\ BLOCK 1
#endif
```

(b) driver.c

Fig. 3. A Linux driver consiting of header and implementation file. The `#define` in the header file influences the presence condition of Block 1 in the implementation file.

In Linux 3.0, an average of 350 `#include` statements [23] have to be resolved per CPP file. Considering expanded compilation units instead of just a single source file can potentially increase the size of the resulting propositional formula tremendously. However, in practice the propositional formulas for the conditional blocks often share few cross constraints. By cleverly investigating these constraints, the implementation variability model can be *sliced* to the given situation, which keeps the size of the corresponding propositional formula small.

The basic idea for "slicing" the code-constraints is as follows: First, we identify all blocks that can influence the presence condition of the block. The resulting smaller presence condition is then the conjunction of all identified blocks plus the additional clauses that result from redefinition statements (c.f. Section 3.2). Because the C Preprocessor processes the file linearly, only blocks that occur lexically earlier can influence a block's presence condition. Without the redefinition statements `#define` and `#undef`, the algorithm for calculating the presence condition would only need to consider blocks that enclose a block or share a common enclosing block. We call such a group of blocks *related*.

However, if redefinition statements occur in a conditional block that has a non-trivial presence condition (i.e., the redefinition statement is only conditionally in effect), a block which uses one of the redefined variables is also influnced by the block enclosing the redefinition statements. So we have to add these redefining blocks to the related block group recursivly.

From this, we devise an algorithm that collects logical implications recursively for all blocks. It returns formulas that are tailored to groups of related blocks. In the worst case, the resulting formula contains all blocks that occur before the specific block. In fact, for code with few

[4] Technically we define the enabled flags with the `-D` command line option of CPP

[5] Technically, this is done by replacing the directive with the referenced file contents. Like the regular CPP tool, header search paths need to be resolved. However unlike the regular CPP, conditional `#include` statements need to be processed as well.

redefinition statements the resulting formulas are considerably smaller. This particularly applies to the Linux kernel, where this tailoring to related blocks allows us to process all conditional blocks in Linux without restricting the analysis to a subset of blocks. For instance, in [40], only configuration controlled conditional blocks, that is, blocks containing a CPP variable that starts with CONFIG_, were considered. This shows that this model slicing technique allows the extraction of the implementation variability model [40] to scale to the size of the Linux kernel.

3.4 Summary

Propositional logic is a solid foundation for representing variability in source text. In this subsection, which extends previous work [40], we have shown an approach that translates the variability as expressed by the C Preprocessor into an implementation variability model in form of propositional formula. A major challenge is the correct representation of the CPP redefinition statements `#define` and `#undef`, which change the *state* of a conditional variable, without extending "pure" propositional logic. With this extension the resulting code variability model could be significantly improved.

4 Converting the Kconfig Variability Model to Presence Implications

There are several strategies to convert configuration space models into boolean formulas [5, 15]. However, due to the size of real models—the KCONFIG model contains more than 10,000 features—, the resulting boolean formulas become very complex. The search for a solution to problems that use such formulas may become intractable (or unnecessarily slow). For Linux development, the configuration space is described with the KCONFIG tool and language.

The KCONFIG tool set was especially written to support the modeling of features and interdependencies of the Linux kernel. It provides a language to describe a variant model consisting of features (referred to as *config options*) together with their constraints and dependencies. Modularization of the variant model is supported by an inclusion mechanism. In Linux kernel version 3.0, a total of 839 KCONFIG files are employed, consisting of 121,506 lines of code that describe 7,702 KCONFIG features and their dependencies. In many respects, the resulting variant model can be compared to feature models known by the software product-line community [39].

Most of the features are shared among all 24 architectures. Naturally, the largest variability occurs in the Hardware Abstraction Layer. For this reason, each architecture in Linux has its own "top-level" KCONFIG file in `arch/`*`archname`*`/Kconfig`. This file references all other KCONFIG files with the `source` statement that works similar to the CPP `#include` statement. This allows the same feature definition to be shared across architectures and have feature implementation share the same subdirectory as their KCONFIG description. Strictly speaking, this means that Linux comes with more than twenty distinct variability models, one for each architecture. For the sake of simplicity, this work focuses on the most relevant architecture, namely `x86`.

```
config HOTPLUG_CPU
  bool "Support for hot-pluggable CPUs"
  depends on SMP && HOTPLUG
   && SYS_SUPPORTS_HOTPLUG_CPU
```

Fig. 4. A simple, boolean KCONFIG item

The KCONFIG files feature a specific syntax for creating configuration options in a hierarchical menu structure that is very similar to the hierarchical structure of feature models. The syntax has evolved during the last ten years and is now considered by the Linux community as powerful and capable asset for abstracting all the inherent complexities of the kernel configuration. For finding configuration inconsistencies the constraints of the KCONFIG model need to be converted into a format that can be combined with other sources of variability. Propositional logic in form of presence implications is a well-understood notation for this purpose. This section introduces the most important language features and explains how to translate each of them into presence implications for a given feature.

4.1 The KCONFIG language

Translation of Kconfig Features. In this paper, we define a (KCONFIG) **feature** to be any user configurable item in the KCONFIG frontend. They usually have a descriptive help text and declare dependencies on other features. They can be one of the types 'int', 'string', 'hex', 'boolean', 'tristate' and 'choice'. In this work, we mostly ignore the features of type 'int', 'string' and 'hex', as they contribute little to the technical variability in Linux. As this work essentially develops a bug finding tool, this leads to a (small) number of missed bugs (at worst).

A simple feature of type boolean is given in Figure 4.1. Tristate features can have one of the states 'not compiled', 'statically linked into the kernel' or 'compiled as loadable kernel module'. In the latter case, the build scripts take care to compile and link the source files that implement the feature into an kernel object (`.ko`) file that can be loaded (and unloaded) at runtime. Feature dependencies can declare constraints on the exact state of tristate features, which need to be taken into account during the translation of the KCONFIG model into propositional formulas.

```
config MICROCODE
 tristate "microcode support"
 select FW_LOADER
 ---help---
   If you say Y here, you will be able to
   update the microcode. [...]
   To compile this driver as a module,
   choose M here.
```

Fig. 5. A simple tristate feature. The help text is advisory to the user. The statement `select FW_LOADER` means that if the user selects the feature `MICROCODE`, kconfig will automatically enable the feature `FW_CONFIG`.

For KCONFIG features of type 'int', 'string', 'hex' and 'boolean' the translation process assigns a (single, boolean) **configuration variable** that represents the user selection of the feature. For 'tristate' features *two* configuration variables are introduced, one of which by convention ends with the suffix _MODULE. The translation process inserts artificial additional logical constraints to prevent that both configuration variables can be enabled at the same time. For instance, based on the KCONFIG feature declarations related to microcode loading in Linux (cf. Figure 5), the translation process produces the following implications:

$$\begin{aligned}
\text{MICROCODE} &\rightarrow \neg\ \text{MICROCODE_MODULE}\\
\text{MICROCODE_MODULE} &\rightarrow \neg\ \text{MICROCODE} \wedge \text{MODULES}\\
\text{MICROCODE_AMD} &\rightarrow \text{MICROCODE_MODULE} \vee \text{MICROCODE}\\
\text{MICROCODE_INTEL} &\rightarrow \text{MICROCODE_MODULE} \vee \text{MICROCODE}
\end{aligned}$$

Note the special option `MODULES`, which is true if the kernel is configured to support loadable modules at all.

Choices. In KCONFIG the `choice` construct allows grouping of features. Our translation process models these groupings as regular features. However as `choice` constructs are only implicitly declared in the KCONFIG language, the translation process assigns the corresponding configuration variable a name that contains a running number so that different choices are distinguishable. For instance, the following KCONFIG fragment offers the user a choice between two config items:

```
choice " High Memory Support "
  config HIGHMEM4G [...]   # These are 2
  config HIGHMEM64G [...] # alternatives
endchoice
```

From this, we deduce the following presence implications:

$$\text{CHOICE\_0} \rightarrow \text{XOR}(\text{HIGHMEM4G}, \text{HIGHMEM64G})$$

Alternatively, a choice can be declared as *optional*:

```
choice " High Memory Support "
  optional
  config HIGHMEM4G [...]   # These are 2
  config HIGHMEM64G [...] # alternatives
endchoice
```

In this case, the cardinality constraints of the feature group is changed so that no member of the group needs to be selected.

KCONFIG features	7702	dependencies	7339
– boolean	2757	selects	4105
– tristate	4700	choices	57
– other (int, hex, ...)	245	features in choices	201

Table 2. Statistics of KCONFIG features and constraints types in Linux 3.0, x86 Architecture.

select. The keyword `select` allows a programmer to trigger the automatic selection of a KCONFIG item in certain situations. This is a straight-forward way to express cross-tree constraints, which can be restricted by further conditions. Consider the following example:

```
config X86
  select HAVE_ARCH_GDB
  select GENERIC_PENDING_IRQ if SMP
```

This feature gets translated to the configuration variable `X86` with the following presence implications:

$$\begin{aligned}
\text{X86} &\rightarrow \text{HAVE_ARCH_GDB}\\
\text{X86} &\rightarrow (\text{SMP} \rightarrow \text{GENERIC_PENDING_IRQ})
\end{aligned}$$

This subsection shows only the most commonly used language features and are by no means complete. The formal semantics of the KCONFIG language has been studied elsewhere [48, 6]; such formalisms describe in detail how to correctly derive the feature constraints with even more precision.

4.2 Analysis of the KCONFIG *Model*

For Linux 3.0 our tools translate 7,702 KCONFIG features with 7,339 dependencies into 12,217 presence implications for the x86 architecture. Table 2 summarizes the results:

We consider a feature that cannot be selected under any selection of other features *unselectable* or short: a *dead* feature. Using the extracted presence implication I_F for a feature F, the following formula checks if the feature F is dead or alive:

$$(\text{F} \wedge I_F) \vee (\text{F\_MODULE} \wedge I_F)$$

`F` and `F_MODULE` are two configuration variables for the same feature F. The formula consists of two parts. The first part checks if the feature can be compiled in statically. The second checks whether the feature can be compiled as module. If the query fails then the feature cannot be selected and has to be considered as *dead*—at least on the analyzed architecture.

An analysis on the x86 architecture model for Linux 3.0 (Table 3) reveals 1,520 features that cannot be selected on x86; however 1,445 of them are selectable on another architecture. This leaves 75 features that cannot be selected on any architecture. Sixty of them are dead because they depend on KCONFIG features that are not declared in any KCONFIG file in the analyzed source tree. Possible explanations for this phenomenon are discussed in Section 6.1. The remaining 15 are under futher investigation.

Require: $\mathcal{S}$ initialized with an initial set of items
1: $\mathcal{R} := \mathcal{S}$
2: **while** $\mathcal{S} \neq \emptyset$ **do**
3: $item := \mathcal{S}.pop()$
4: $\varphi' := presenceImplication(item)$
5: **for all** i that are referenced in φ' **do**
6: **if** $i \notin \mathcal{R}$ **then**
7: $\mathcal{S}.push(i)$
8: $\mathcal{R}.push(i)$
9: **end if**
10: **end for**
11: **end while**
12: **return** $\mathcal{R}$

Fig. 6. Algorithm for configuration model slicing

4.3 Slicing on the Presence Implications

In order to efficiently analyze configuration derived inconsistencies, it makes sense to not consider the whole KCONFIG model at once, but only the subset that is relevant for a given analysis. Therefore, we devise an algorithm that implements *model slicing* for KCONFIG. This allows us to generate sub-models from the original model that are smaller than the complete model. To illustrate, suppose we want to check if a specific block of the source code can be enabled by any valid user configuration. This is expressed by the satisfiability of the formula $\mathcal{V} \wedge \texttt{Block}_N$. With a full model, the term $\mathcal{V}$ would contain all user-visible features as logical variables; for the Linux kernel it would have more than 10,000 variables. However, not all features influence the solution for this specific problem. The key challenge is to find a sufficient – and preferably minimal – subset of features that can possibly influence the selection of the analyzed source code. That is, a model $\mathcal{V}'$ where $\mathcal{V}' \models \mathcal{V}$ for the operations of interest, for example: $\mathcal{V}' \wedge \texttt{Block}_N \models \mathcal{V} \wedge \texttt{Block}_N$.

Our slicing algorithm for this purpose is depicted in Figure 6. The goal is to find the set of configuration items that can possibly affect the selection of one or more given initial items. (In our tool, which we will present in Section 5.3, this initial set of items will be taken from the `#ifdef` expressions.) The basic idea is to check the presence implications of each item for additional relevant items. Both *direct* and *indirect* dependencies from the initial set of features are thus taken into account such that the resulting set contains all features that can influence the features in the initial set. The algorithm basically requires (1) the initial set of features, and (2) the set $\mathcal{V}$ of presence implications for all features. The logical constraints φ' for the item *feature* is accessed by the helper function *presenceImplication*(*feature*)) in Figure 6. The set $\mathcal{V}$ is calculated by translating the KCONFIG features into propositional formulas as described in Section 4.1. Each member of the set $\mathcal{V}$ has the form: $feature \rightarrow \varphi'$. By this, the constraints for a subset of features can by conveniently constructed by conjuncting presence implications. The following description explains how to determine which presence implications have to be conjuncted for a given set of features.

The algorithm Figure 6 returns the set of all features for the sub-model $\mathcal{V}'$ of $\mathcal{V}$ that contains the feature implications for all features in the "slice". In the first step (Line 1) the resulting set $\mathcal{R}$ is initialized with the list of given features. Then, the algorithm iterates until the working stack $\mathcal{S}$ is empty. The number of iterations depends on the given set of feature implications in $\mathcal{V}$. In each iteration (Lines 2–11), a feature is taken from the stack and its presence implication φ' is returned by the helper function *presenceImplication*(*feature*). This formula represents the constraints that apply when the feature *item* is selected in a configuration. All features i that appear in φ' and have not already been processed (Line 6), are added to the working stack $\mathcal{S}$ and the result set $\mathcal{R}$. The resulting set $\mathcal{R}$ always contains all items that are initially given, plus the items that (recursively) appear in each iteration. At Line 12, $\mathcal{R}$ holds the items that have already been processed and, consequently, contains the set of returned items. This algorithm always terminates; in the worst case, it will return all features and the slice will be exactly like the original model.

To illustrate, consider the following feature defined in the KCONFIG language:

```
config DISCONTIGMEM
        def_bool y
        depends on (!SELECT_MEMORY_MODEL &&
                    ARCH_DISCONTIGMEM_ENABLE) ||
                    DISCONTIGMEM_MANUAL
```

The presence implication for the feature `DISCONTIGMEM` is simply the selection of the feature itself and the expression of the `depends on` option. If a feature has several definitions of prompts and defaults, the feature implies the disjunction of the condition of each option that control its selection.

Using this algorithm, a smaller (*sliced*) model can be built by conjuncting the presence implications for configuration variables that have been identified. This process is best explained by an example, which comes the situation depicted in Figure 7:

DISCONTIGMEM $\rightarrow$
(!SELECT_MEMORY_MODEL $\wedge$
ARCH_DISCONTIGMEM_ENABLE) $\vee$
DISCONTIGMEM_MANUAL
DISCONTIGMEM_MANUAL $\rightarrow$ ARCH_DISCONTIGMEM_ENABLE
ARCH_DISCONTIGMEM_ENABLE $\rightarrow$ **true**
EFI $\rightarrow$ ACPI

In this example model the two configuration variables `DISCONTIGMEM` and `DISCONTIGMEM_MANUAL` "imply" `ARCH_DISCONTIGMEM_ENABLE`. The presence implication for the configuration variable `ARCH_DISCONTIGMEM_ENABLE` itself is a tautology. Additionally, `EFI` implies `ACPI`. For the presence implication of `DISCONTIGMEM` we can safely ignore the latter and the tautology. This allows to simplify the resulting conjunction to:

KCONFIG features	7,514	configuration variables	12,217
– selectable (x86)	5,994	– slice for a variable	114.7
– unselectable (x86)	1,520	logic clauses	202,809
– selectable (other)	1,445	– slice for a variable	2,326.19

Table 3. Statistics of the translated KCONFIG model for Linux 3.0 on the x86 Architecture.

$$
\begin{aligned}
&(\text{DISCONTIGMEM} \rightarrow \\
&\quad (!\text{SELECT\_MEMORY\_MODEL} \wedge \\
&\quad\quad \text{ARCH\_DISCONTIGMEM\_ENABLE}) \vee \\
&\quad \text{DISCONTIGMEM\_MANUAL}) \\
\wedge\ &(\text{DISCONTIGMEM\_MANUAL} \rightarrow \\
&\quad \text{ARCH\_DISCONTIGMEM\_ENABLE})
\end{aligned}
$$

Size of the slices and correctness. The effectiveness of model slicing on the Linux KCONFIG model for the x86 architecture can be seen in Table 3. In total the translation process produces 12,217 configuration variables, for which the corresponding propositional formulas consist of 202,809 clauses in total. When applying the slicing algorithm for each variable, the resulting formula contains on average 115 configuration variables and 2,326 clauses. From these figures we conclude that in average only 0.94% of the configuration variables (1.14% of the clauses) are important for the presence condition of a single configuration variable. While these impressive figures apply for Linux only, a similar effectiveness for configuration models of other operating systems and system software can be expected as well.

We validate our slicing algorithm by writing test-cases. To verify their correctness the same reasoning operation is applied twice, one with the original whole model $\mathcal{V}$ and one with the $\mathcal{V}'$. Naturally, both analyses have to be consistent. The only allowed difference is the time that it takes to perform the operations as the models in general have different sizes. By doing so on defects that our tool reveals, we confirm that the slicing algorithm does not introduce false negatives.

As an more theoretical alternative explanation consider the following situation: Let $\mathcal{V}'$ be an incomplete subset of $\mathcal{V}$, which lacks a necessary implication m from the original model $\mathcal{V}$. This lacking implication m regards the configuration variable v. If no other implication in $\mathcal{V}'$ references this variable v, then v would be a free variable without influence on other implications. On the other hand, some other implication does reference the variable v, then the algorithm would have missed to include m in $\mathcal{V}'$. However, since all variables in $\mathcal{V}'$ are resolved recursively, m cannot be missing by construction. This case is effectively impossible. Therefore $\mathcal{V}'$ always includes all implications that influence the presence implications for the given configuration variables and cannot miss implication that would introduce false negatives.

4.4 Summary

KCONFIG is a sophisticated language and tooling that allows the users of Linux to configure over 10,000 features. It has been adopted for a number of other open source projects, such as busybox,[6] coreboot[7] and many more. The size and the great success of the Linux kernel makes this tooling highly relevant for research.

In this section, we have shown our approach to translate the constraints of features in KCONFIG into propositional formulas with configuration variables as atoms. While we do not aim at encoding the whole model into a single holistic formula, features can still be queried individually for satisfiability, which alone is useful for instance for finding *dead* (i.e., unselectable) features. In the next section, we use this representation for *crosschecking* the variability from KCONFIG with the extracted model from the implementation.

5 Crosschecking Implementation with the Explicit Variability Model

The problem analysis in Section 2.2 points out that many configurability-related defects are caused by inconsistencies that result from the fact that configurability is defined by two (technically separated, but conceptually related) models: the configuration space, and the implementation space. For Linux, both are described in Section 3 and Section 4.

The general idea for finding configuration consistency defects is to extract all configurability-related information from both models into a common representation (a propositional formula), which is then used to cross-check the variability exposed within and across both models in order to find inconsistencies. We call these inconsistencies *configurability defects:*

> A **configurability defect** (short: **defect**) is a configuration-conditional **item** that is either **dead** (never included) or **undead** (always included) under the precondition that its **parent** (enclosing item) is included.

Examples for *items* in Linux are: KCONFIG options, build rules, and (most prominent) `#ifdef` blocks. The `CONFIG_NUMA` example discussed in Section 2.2 (see Figure 7) bears two *defects* in this respect: Block_2 is *undead* and Block_3 is *dead*. Defects can be further classified as:

Confirmed – a defect that has been confirmed as *unintentional* by the corresponding developers. If the defect has an effect on the binary code of at least one Linux implementation variant, we call it a **bug**.

[6] `http://busybox.net`

[7] `http://coreboot.org`

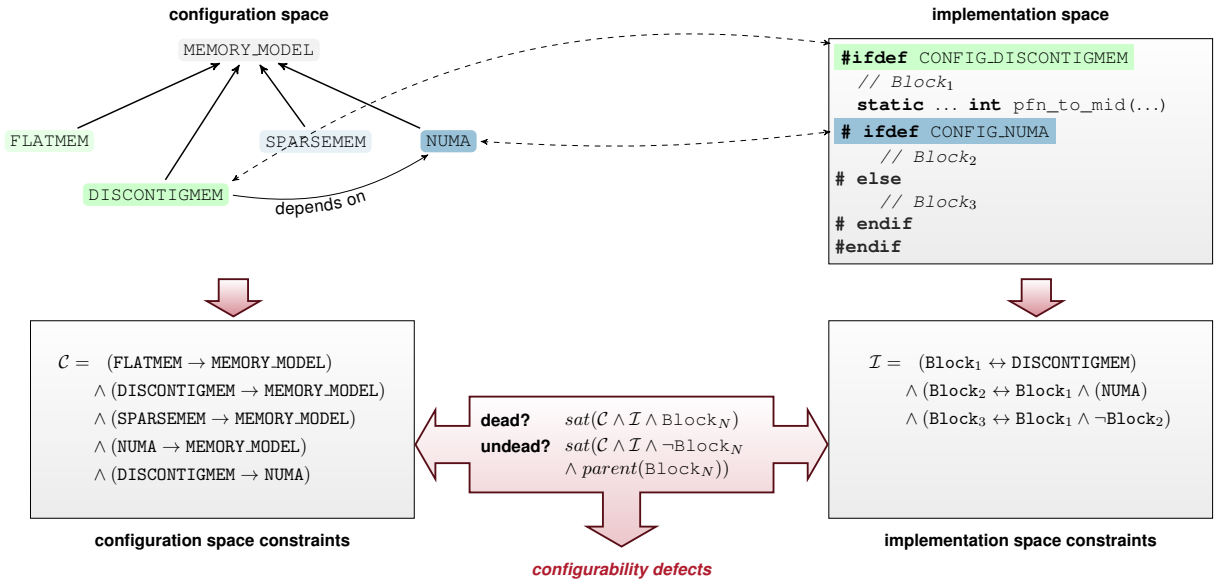

Fig. 7. Our approach at a glance: The variability constraints defined by both spaces are extracted separately into propositional formulas, which are then examined against each other to find inconsistencies we call *configurability defects*.

Rule violation – a defect that, even though it breaks a generally accepted development rule, has been confirmed as *intentional* by the corresponding developers.

Patch 1 discussed in Section 2.2 fixes a *bug*, Patch 2 a *confirmed defect*. In the case of Linux a rule violation is usually the use of the `CONFIG_` prefix for preprocessor flags that are not (yet) defined by KCONFIG. We will discuss the source of rule violations more thoroughly in Section 6.1.

Essential for the analysis of configurability problems is a common representation of the variability that spreads over different software artifacts. The idea is to individually convert each variability source (e.g., source files, KCONFIG, etc.) to a common representation in form of a sub-model and then combine these sub-models into a model that contains the whole variability of the software project. This makes it possible to analyze each sub-model as well their combination in order to reveal inconsistencies across sub-models.

Most of the constructs that model the variability both in the configuration and implementation spaces can be directly translated to propositional logic; therefore, propositional logic is our abstraction means of choice. As a consequence, the detection of configuration problems boils down to a satisfiability problem.

Linux (and many other systems) keep their configuration space ($\mathcal{C}$) and their implementation space ($\mathcal{I}$) separated. The variability model ($\mathcal{V}$) can be defined as:

$$\mathcal{V} := \mathcal{C} \wedge \mathcal{I} \tag{3}$$

$\mathcal{V} \mapsto \{0, 1\}$ is a boolean formula over all features of the system; $\mathcal{C}$ and $\mathcal{I}$ are the boolean formulas representing the constraints of the configuration and implementation spaces, respectively. Properly capturing and translating the variability of different artifacts into the formulas $\mathcal{C}$ and $\mathcal{I}$ is crucial for building the complete variability model $\mathcal{V}$. Once the model $\mathcal{V}$ is built we use it to search for defects.

With this model, we validate the implementation for configurability defects, that is, we check if the conditions for the presence of the block ($\texttt{Block}_N$) are fulfillable in the model $\mathcal{V}$. For example, consider Figure 7: The formula shown for dead blocks is satisfiable for Block_1 and Block_2, but not for Block_3. Therefore, Block_3 is considered to be *dead*; similarly the formula for undead blocks indicates that Block_2 is *undead*.

5.1 Challenges

In order to implement the solution sketch described above in practice for real-world large-scale system software, we face the following challenges:

Performance. As we aim at dealing with huge code bases, we have to guarantee that our tools finish in a reasonable amount of time. More importantly, we also aim at supporting programmers at development time when only a few files are of interest. Therefore, we consider the efficient check for variability consistency during incremental builds essential.

Flexibility. Projects that handle thousands of features will eventually contain *desired* inconsistencies with respect to their variability. Gradual addition or removal of features and large refactorings are examples of efforts that may lead to such inconsistent states within the lifetime of a project. Also, evolving projects may change their requirements regarding their variability descriptions. Therefore, a tool that checks for

configuration problems should be flexible enough to incorporate information about desired issues in order to deliver precise and useful results; it should also minimize the number of false positives and false negatives.

In order to achieve both *performance* and *flexibility*, the implementation of our approach needs to take the particularities of the software project into account. Moreover, the precision of the configurability extraction mechanism has direct a impact on the rate of false positive and false negative reports. As many projects have developed their own, custom tools and languages to describe configuration variability, the configurability extraction needs to be tightly tailored.

5.2 Crosschecking Among Variability Spaces

Our approach converts the different representations of variability to a common format so that we can check for inconsistencies, the configurability *defects*. Defects appear in two ways, either as **dead**, that is, unselectable blocks, or **undead**, that is, always present blocks. Both kinds of defects indicate code that is only seemingly conditional. They can be found within single models as presented in the previous two sections in isolation as well as across multiple models.

Within a single model we have **implementation-only** defects, which represent code blocks that cannot be selected regardless of the systems' selected features; the structure of the source file itself contains contradictions that impede the selection of a block. This can be determined by checking the satisfiability of the formula $sat(b_i \leftrightarrow \mathcal{PC}(b_i))$. **Configuration-only** defects represents features that are present in the configuration-space model but do not appear in any valid configuration of the model, which means that the presence condition of the feature is not satisfiable. We can check for such defects by solving: $sat(feature \rightarrow presenceCondition(feature))$.

Across multiple models we have **configuration-implementation** defects, which occur when the rules from the configuration space contradict rules from the implementation space. We check for such defects by solving $sat((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V})$. **Referential** defects are caused by a *missing feature* (m) that appears in either the configuration or the implementation space *only*. That is, $sat((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V} \wedge \neg(m_1 \vee \cdots \vee m_n))$ is unsatisfiable.

We categorize all identified defects as either *logic* or *symbolic*. Logic defects are those that can only be found by determining the satisfiability of a complex boolean formula. Symbolic defects are referential defects where the expression of the analyzed block $exp(b_i)$ is an atomic formula, that is, the expression consists of a single CPP flag.

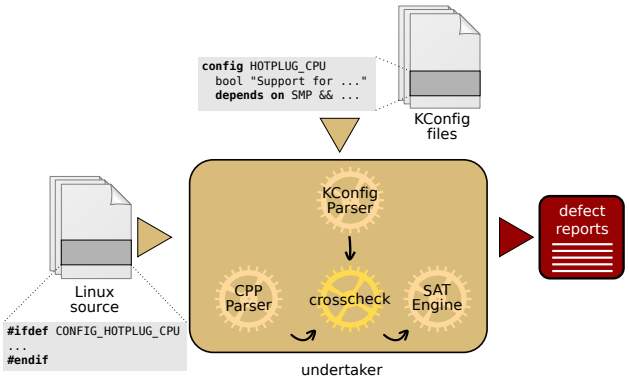

Fig. 8. Principle of Operation

5.3 Implementation for Linux

In order to evaluate our approach, we have developed a prototype tool for Linux and a workflow to submit our results to the kernel developers. We started submitting our first patches in February 2010, at which time Linux version 2.6.33 has just been released. Most of our patches entered the mainline kernel tree during the merge window of version 2.6.36. In the following, we describe our tool and summarize the results.

We named our tool UNDERTAKER, because its task is to identify (and eventually bury) dead and undead CPP-Blocks. Its basic principle of operation is depicted in Figure 8: The different sources of variability are parsed and transformed into propositional formulas. For CPP parsing, we use the PUMA [47] parsing library; for proper parsing of the Kconfig files, we have implemented the extraction tool as modification of the original Linux KCONFIG implementation. The outcome of these parsers is fed into the crosschecking engine as described in Section 5.2 and solved using the PICOSAT.[8] package. The tool itself is published as Free Software and available on our website.[9]

Our tool scans each `.c` and `.h` file in the source tree individually. This allows developers to focus on the part of the source code they are currently working on and to get instant results for incremental changes. The results come as *defect reports* per file: For each file all configurability-related CPP blocks are analyzed for satisfiability, which yields the defect types described in the previous section. For instance, the report produced for the *configuration-implementation defect* from Figure 7 looks like this:

```
Found Kconfig related DEAD in arch/parisc/include
    /asm/mmzone.h,
line 40: Block B6 is unselectable, check the SAT
    formula.
```

Based on this information, the developer now revisits the KCONFIG files. The basis for the report is a formula that is falsified by our SAT solver. For this particular example the following formula was created:

[8] http://fmv.jku.at/picosat/

[9] http://vamos.informatik.uni-erlangen.de/

```
1  #B6:arch/parisc/include/asm/mmzone.h:40:1:logic:undead

2  B2 &
3  !B6 &

4  (B0 <-> !_PARISC_MMZONE_H) &
5  (B2 <-> B0 & CONFIG_DISCONTIGMEM) &
6  (B4 <-> B2 & !CONFIG_64BIT) &
7  (B6 <-> B2 & !B4) &
8  (B9 <-> B0 & !B2) &

9  (CONFIG_64BIT -> CONFIG_PA8X00) &
10 (CONFIG_ARCH_DISCONTIGMEM_ENABLE -> CONFIG_64BIT) &
11 (CONFIG_ARCH_SELECT_MEMORY_MODEL -> CONFIG_64BIT) &
12 (CONFIG_CHOICE_11 -> CONFIG_SELECT_MEMORY_MODEL) &
13 (CONFIG_DISCONTIGMEM -> !CONFIG_SELECT_MEMORY_MODEL &
       CONFIG_ARCH_DISCONTIGMEM_ENABLE |
       CONFIG_DISCONTIGMEM_MANUAL) &
14 (CONFIG_DISCONTIGMEM_MANUAL -> CONFIG_CHOICE_11 &
       CONFIG_ARCH_DISCONTIGMEM_ENABLE) &
15 (CONFIG_PA8X00 -> CONFIG_CHOICE_7) &
16 (CONFIG_SELECT_MEMORY_MODEL -> CONFIG_EXPERIMENTAL |
       CONFIG_ARCH_SELECT_MEMORY_MODEL)
```

This formula can be deciphered easily by examining its parts individually. The first line shows an "executive summary" of the defect; here, Block B6, which starts in Line 40 in the file `arch/parisc/include/asm/mmzone.h`, is a *logical* configuration defect in form of a block that cannot be unselected ("undead"). Lines 4 to 8 show the *presence conditions* of the corresponding blocks (cf. Section 3 and [40]); they all start with a block variable and by construction cannot cause the formula to be unsatisfiable. From the structure of the formula, we see that Block B0 implements the CPP "include guard" and therefore encloses all other blocks. Moreover, the way the Blocks B4 on Line 6 and B6 on Line 7 reference B2 indicate that B2 encloses B4 and B6. Lines 9 et seqq. contain the extracted implications from KCONFIG (cf. Section 4). In this case, it turns out that the KCONFIG implications from Line 9 to 16 show a *transitive* dependency from the KCONFIG item CONFIG_DISCONTIGMEM (cf. Block B2, Line 5) to the item CONFIG_64BIT (cf. Block B4, Line 6). This means that the KCONFIG selection has no impact on the evaluation of the `#ifdef` expression and the code can thus be simplified. We have therefore proposed the following patch to the Linux developers:[10]

```
1  diff --git a/arch/parisc/include/asm/mmzone.h b/arch/
       parisc/include/asm/mmzone.h
2  --- a/arch/parisc/include/asm/mmzone.h
3  +++ b/arch/parisc/include/asm/mmzone.h
4  @@ -35,6 +35,1 @@ extern struct node_map_data node_data
       [];
5
6  -#ifndef CONFIG_64BIT
7  #define pfn_is_io(pfn) ((pfn & (0xf0000000UL >>
       PAGE_SHIFT)) == (0xf0000000UL >> PAGE_SHIFT))
8  -#else
9  -/* io can be 0xf0f0f0f0f0xxxxxx or 0xffffffff0000000 */
10 -#define pfn_is_io(pfn) ((pfn & (0xf000000000000000UL >>
       PAGE_SHIFT)) == (0xf000000000000000UL >> PAGE_SHIFT))
11 -#endif
```

Please note that this is one of the more complicated examples. Most of the defects reports have in fact only a few lines and are much easier to comprehend.

[10] http://lkml.org/lkml/2010/5/12/202

subsystem	#ifdefs	logic	symbolic	total
arch/	33757	345	581	926
drivers/	32695	88	648	736
fs/	3000	4	13	17
include/	7241	6	11	17
kernel/	1412	7	2	9
mm/	555	0	1	1
net/	2731	1	49	50
sound/	3246	5	10	15
virt/	53	0	0	0
other subsystems	601	4	1	5
$\sum$	85291	460	1316	1776
fix proposed		150 (1)	214 (22)	364 (23)
confirmed defect		38 (1)	116 (20)	154 (21)
confirmed rule-violation		88 (0)	21 (2)	109 (2)
pending		24 (0)	77 (0)	101 (0)

Table 4. Upper half: `#ifdef` blocks and defects per subsystem in Linux version 2.6.35; Lower half: acceptance state of defects (bugs) for which we have submitted a patch

Results. Table 4 (upper half) summarizes the defects that UNDERTAKER finds in Linux 2.6.35, differentiated by subsystem. When counting defects in Linux, some extra care has to be taken with respect to architectures: Linux employs a separate KCONFIG-model per architecture that may also declare architecture-specific features. Hence, we need to run our defect analysis over every architecture and intersect the results. This prevents us from counting, for example, MIPS-specific blocks of the code as *dead* when compiling for x86. The the code below `arch/` is architecture-specific by definition and only checked against the configuration model of the respective architecture.

Most of the 1,776 defects are found in `arch/` and `drivers/`, which together account for more than 75 percent of the configurability-related `#ifdef`-blocks. For these subsystems, we find more than three defects per hundred `#ifdef`-blocks, whereas for all other subsystems this ratio is below one percent (`net/` below two percent). These numbers support the common observation (e.g., [16]) that "most bugs can be found in driver code", which apparently also holds for configurability-related defects. They also indicate that the problems induced by "`#ifdef`-hell" grow more than linearly, which we consider as a serious issue for the increasing configurability of Linux and other system software.

Even though the majority of defects (74%) are caused by symbolic integrity issues, we also find 460 logic integrity violations, which would be a lot harder to detect by "developer brainpower".

Performance. We have evaluated the performance of our tool with Linux 2.6.35. A full analysis of this kernel processes 27,166 source files with 82,116 configurability-conditional code blocks. This represents the information from the implementation space. The configuration space provides 761 KCONFIG files defining 11,283 features.

A *full* analysis that produces the results as shown in Table 4 takes around 15 minutes on a modern Intel

Fig. 9. Processing time for 27,166 Linux source files

quadcore with 2.83 GHz and 8 GB RAM. However, the implementation still leaves a lot of room for optimization: Around 70 percent of the consumed CPU time is *system* time, which is mostly caused by the fact that we `fork()` the SAT solver for every single `#ifdef` block.

Despite this optimization potential, the runtime of UNDERTAKER is already appropriate to be integrated into (much more common) incremental Linux builds. Figure 9 depicts the file-based runtime for the Linux source base: Thanks to our slicing algorithm, 94 percent of all source files are analyzed in less than half a second; less than one percent of the source files take more than five seconds and only four files take between 20 and 30 seconds. The upper bound (29.1 seconds) is caused by `kernel/sysctl.c`, which handles a very high number of features; changes to this file often require a complete rebuild of the kernel anyway. For an incremental build that affects about a dozen files, UNDERTAKER typically finishes in less than six seconds.

5.4 Evaluation of Findings

To evaluate the quality of our findings, we have given our defect reports to two undergraduate students to analyze them, propose a change, and submit the patch upstream to the responsible kernel maintainers. Figure 10 depicts the whole workflow.

The first step is *defect analysis*: The students have to look up the source-code position for which the defect is reported and understand its particularities, which in the case of logical defects (as in the `CONFIG_NUMA` example presented in Figure 7) might also involve analyzing KCONFIG dependencies and further parts of the source code. This information is then used to develop a *patch* that fixes the defect.

Based on the response to a submitted patch, we *improve and resubmit* and finally classify it (and the defects it fixes) in two categories: *accept (confirmed defect)* and *reject (confirmed rule violation)*. The latter means that the responsible developers consider the defect for some reason as *intended*; we will discuss this further in Section 6.1. As a matter of pragmatics, these defects are added into a local whitelist to filter them out in future runs.

In the period of February to July 2010, the students have submitted 123 patches. The submitted patches focus on the `arch/` and `driver/` subsystems and fix 364 out of 1,776 identified defects (20%). 23 (6%) of the analyzed and fixed defects were classified as bugs. If we extrapolate this defect/bug ratio to the remaining defects, we can

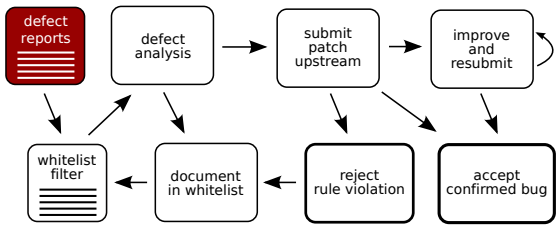

Fig. 10. Based on the analysis of the defect reports, a patch is manually created and submitted to kernel developers. Based on the acceptance, we classify the defects that are fixed by our patch either as *confirmed rule violation* or *confirmed defect*.

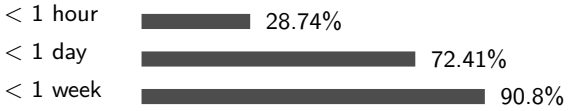

Fig. 11. Response time of 87 answered patches

expect to find another 80+ configurability-related bugs in the Linux kernel.

Reaction of Kernel Maintainers. Table 5 lists the state of the submitted patches in detail; the corresponding defects are listed in Table 4 (lower half). In general, we see that our patches are well received: 87 out of 123 (71%) have been answered; more than 70 percent of them within less than one day, some even within minutes (Figure 11). We take this as an indication that many of our patches are easy to verify and in fact appreciated.

Contribution to Linux. Table 5 also classifies the submitted patches as *critical* and *noncritical*, respectively. Critical patches fix *bugs*, that is, configurability defects that have an impact on the binary code. We did not investigate in detail the run-time observable effects of the 23 identified bugs. However, what can be seen from Table 5 is that the responsible developers consider them as worth fixing: 16 out of 17 (94%) of our critical patches have been answered; 9 have already been merged into Linus Torvalds' master git tree for Linux 2.6.36.

The majority of our patches fix defects that affect the source code only, such as the examples shown in Section 2.2. However, even for these noncritical patches 57 out of 106 (54%) have already reached acknowledged state or better. These patches clean up the kernel sources by removing 5,129 lines of configurability-related dead code and superfluous `#ifdef` statements ("cruft"). We consider this as a strong indicator that the Linux community is aware of the negative effects of configurability on the source-code quality and welcomes attempts to improve the situation.

Figure 12 depicts the impact of our work on a timeline of Linux kernel releases. To build this figure, we ran our tool on previous kernel versions and calculated the number of configurability defects that were *fixed* and

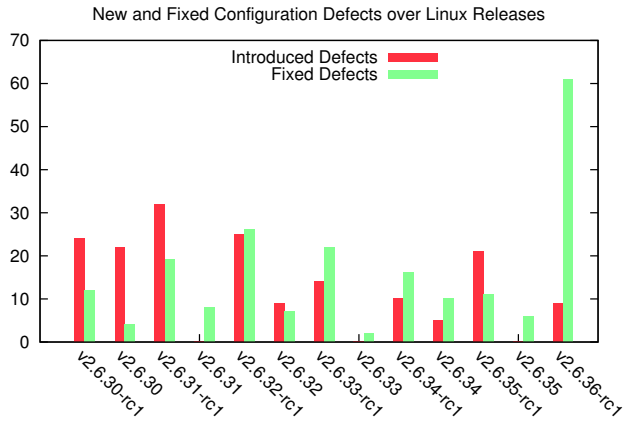

Fig. 12. Evolution of defect blocks over various Kernel versions. Most of our work was merged after the release of Linux version 2.6.35.

patch status	critical	noncritical	$\sum$
submitted	17	106	123
unanswered	1	35	36
ruleviolation	1	14	15
acknowledged	1	14	15
accepted	5	3	8
mainline	9	40	49

Table 5. *Critical* patches do have an effect on the resulting binaries (kernel and runtime-loadable modules). Noncritical patches remove text from the source code only.

introduced with each release. Most of our patches entered the mainline kernel tree during the merge window of version 2.6.36. Given that the patch submissions of two students have already made such a measurable impact, we expect that a consequent application of our approach, ideally directly by developers that work on new or existing code, could significantly reduce the problem of configurability-related consistency issues in Linux.

6 Discussion

Our findings have yielded a notable number of configurability defects in Linux. In the following, we discuss some potential causes for the introduction of defects and rule violations, threats to validity, and the broader applicability of our approach.

6.1 Interpretation of the Feedback

About 57 of the 123 submitted patches were accepted without further comments. We take this as an indication that experts can easily verify the correctness of our submissions. Because of the distributed development of the Linux kernel, drawing the line between acknowledged and accepted (i.e., patches that have been merged for the next release), is challenging. We therefore count the 87 patches for which we received comments by Linux maintainers that maintain a public branch on the internet or are otherwise recognized in the Linux community as a confirmation that we identified a valid defect.

Causes for Defects. We have not yet analyzed the causes for defects systematically; doing this (e.g., using HERODOTOS [33]) remains a topic for further research. However, we can already name some common causes, for which we need to consider how changes get integrated into Linux:

Logical defects are often caused by *copy and paste* (which confirms a similar observation in [16]). Apparently code is often copied *together* with an enclosing `#ifdef-#else` block into a new context, where either the `#ifdef` or the `#else` branch is always taken (i.e., undead) and the counterpart is dead.

The most common source for symbolic defects is *spelling mistakes*, such as the `CONFIG_HOTPLUG` example in Patch 1. Another source for this kind of defects is *incomplete merges* of ongoing developments, such as architecture-specific code that is maintained by respective developer teams who maintain separate development trees and only submit hand-selected *patch series* for inclusion into the mainline. Obviously, this hand selection does not consider configurability-based defects – despite the recommendations in the patch submission guidelines:[11]

```
6: Any new or modified CONFIG options don't muck up the
   config menu.
7: All new Kconfig options have help text.
8: 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 provides a systematic, tool-based approach for this demanded checking of KCONFIG combinations.

Reasons for Rule Violations. On the other hand, we count 15 patches that were rejected by Linux maintainers. For all these patches, the respective maintainers confirmed the defects as valid (in one case even a bug!), but *nevertheless* prefer to keep them in the code. Reasons for this (besides *carelessness* and *responsibility uncertainties*) include:

Documentation. Even though all changes to the Linux source code are kept in the version control system (GIT), some maintainers have expressed their preference to keep outdated or unsupported feature implementations in the code in order to serve as a reference or template (e.g., to ease the porting of driver code to a newer hardware platform).

Out-of-tree development. In a number of cases, we find configurability-related items that are referenced from code in private development trees only. Keeping these symbolic defects in the kernel seems to be considered helpful for future code submission and review.

[11] `Documentation/SubmitChecklist` in the Linux source

While it is debatable if all of the above are good reasons or not, of course we have to accept the maintainers preferences. The whitelist approach provides a pragmatic way to make such preferences explicit – so that they are no longer reported as defects, but can be addressed later if desired.

6.2 Threats to Validity

Accuracy. A strong feature of our approach is the accuracy with which configurability defects can be found. In our approach, false positives are conditional blocks that are falsely reported as unselectable. This means that there is a KCONFIG selection for which the code is seen by the compiler. By design, our approach operates *conceptually exact*. So with exact models we would find all defects. However, since the *exact* extraction of constraints from languages without well-defined semantics (there is no formal semantic of KCONFIG available, only the implementation) is unfeasible, we focus on constraints that we have verified against the implementation. This avoids false positives (minus implementation bugs), so the major threat to validity is the rate of false negatives, that is, the rate of the remaining, unidentified issues.

In fact, we have found for 2 (confirmed) defects explicit `#error` statements in the source that provoke compilation errors in case an *invalid* set of features has been selected. In our experiment, we classified these defects as confirmed rule violations. On top of that, we can find 28 similar `#error` statements in Linux 2.6.35. This indicates some distrust of developers in the variability declarations in KCONFIG, which our tool helps to mitigate by checking the effective constraints accurately.

Coverage. The current implementation does not yet analyze nonpropositional expressions in `#ifdef` statements, like comparisons against the integral value of some `CONFIG_` flag. This affects about 2% out of 82,116 `#ifdef` blocks. We are currently looking into improving our implementation to reduce this number even further by rewriting the extracted constraints and process them using a satisfiability modulo theories (SMT) or constraint solving problem (CSP) engine.

An important, yet not considered source of feature constraints is the build system (makefiles). 91 percent of the Linux source files are feature-dependent, that is, they are not compiled at all when the respective feature has not been selected. Incorporation of these additional constraints into our approach is straight-forward: they can simply be added as further conjunctions to the variability model. These additional constraints could possibly restrict the variability even further, and thereby lead to false negatives.

Subtle semantic details and anachronisms of the KCONFIG language and implementation [48, 6] made our engineering difficult and contributed to the number of false negatives. At the time we conducted the experiment in Section 5.3, our implementation did not completely support the KCONFIG features *default value* and *select*. Since then, we have fixed these issues in the undertaker, which increases the raw number of defects from 1,776 to 2,972.

In no case did our approach result in a change that proposes to remove blocks that are used in production. However, in one case[12] we stumbled across old code that is useful with some additional debug-only patches that have never been merged. It turned out that the patches in question are no longer necessary in favor of the new tracing infrastructure. Our patch therefore has contributed to the removal of otherwise useless and potentially confusing code.

6.3 General Applicability of the Approach

Linux is the most configurable piece of software we are aware of, which made it a natural target to evaluate accuracy and scalability of our approach. However, the approach can be implemented for other software families as well, given there is some way to extract feature identifiers and feature constraints from all sources of variability. This is probably always the case for the implementation space (code), which is generally configured by `CPP` or some similar preprocessor. Extracting the variability from the configuration space is straight-forward, too, as long as features and constraints are described by some semi-formal model, such as KCONFIG. The configurability of eCos, for instance, is described in the configuration description language (CDL) [30], whose expressiveness is comparable to KCONFIG.

KCONFIG itself is employed by more and more software families besides Linux. Examples include OpenWRT[13] and BusyBox.[14] For these software families our approach could be implemented with minimal effort.

However, even if the system software is configured by a simple configure script (such as in FreeBSD or GNU/autoconf projects), it would still be possible to extract feature identifiers and, hence, use our approach to detect symbolic configurability defects. As our Linux case-study in Section 5.3 shows—74 percent of all defects were symbolic—this means a considerable number of defects. Feature constraints, on the other hand, are more difficult to extract from configure files, as they are commonly given as human-readable comments only. A possible solution might be to employ techniques of natural language processing to automatically infer the constraints from the comments, similar to the approach suggested in [43].

In a more general sense, our approach could be combined with other tools to make them *configurability aware*. For instance, modifications on in-kernel APIs and other

[12] `http://kerneltrap.org/mailarchive/linux-ext4/2010/2/8/6762333/thread`

[13] `http://www.openwrt.org`

[14] `http://www.busybox.net`

larger refactorings are commonly tool assisted (e.g., [32]). However, refactoring tools are generally not aware of code liveness and suggest changes in dead code. Our approach contributes to avoiding such useless work.

6.4 Variability-Aware Languages

The high relevance of static configurability for system software gives rise to the question if we need better programming languages. Ideally, the language and compiler would directly support configurability (implementation and configuration), so that symbolic and semantic integrity issues can be prevented upfront by means of type-systems or at least be checked for at compile-time.

With respect to the implementation of configurability it is generally accepted that CPP might not be the right tool for the job [41, 27]. Many approaches have been suggested for a better separation of concerns in configurable (system) software, including, but not limited to: object-orientation [10], component models [18, 37], aspect-oriented programming (AOP) [12, 28], or feature-oriented programming (FOP) [2]. However, the systems community tends to be reluctant to adopt new programming paradigms, mostly because we fear unacceptable run-time overheads and immature tools. For instance, C++ was ruled out of the Linux kernel for exactly these reasons.[15] The authors certainly disagree here (in previous work with embedded operating systems we could show that C++ class composition [8] and AOP [29] provide excellent means to implement overhead-free, fine-grained static configurability). Nevertheless, we have to accept CPP as the still de-facto standard for implementing static configurability in system software [42, 27].

With respect to modeling configurability, feature modeling and other approaches from the product line engineering domain [13, 36] provide languages and tooling to describe the variability of software systems, including systematic consistency checks. KCONFIG for Linux or CDL for eCos fit in here. However, what is generally missing is the bridge between the modeled and the implemented configurability. Hence tools like UNDERTAKER remain necessary.

7 Related Work

This work discusses the (static) analysis of a high-profile and widely deployed software project, the Linux kernel. Due to its sheer size, importance, and source-code availability, Linux has been a first-class evaluation subject for approaches and tools for static analyses in the the systems as well as the software engineering communities. This section discusses the related from different communities in sequence.

[15] *Trust me – writing kernel code in C++ is a BLOODY STUPID IDEA* LINUS TORVALDS [2004], http://www.tux.org/lkml/#s15-3

Analysis of CPP variability So far, there have been proposed several approaches for analyzing conditional compilation usage. In 2000 Hu et al. [21] proposes to analyze conditional compilation with symbolic execution. This approach maps conditional compilation to execution steps: Inclusion of headers map to *calls*, alternative blocks to branches in a CFG, which is then processed with traditional symbolic execution techniques. Lattendresse [25] improves this technique by using *rewrite systems* in order to find presence conditions for every line of code. Similarly to our approach, the presence conditions of all conditional blocks are calculated during the process as well. However, our extraction variability from CPP source code trades performance and practicality over theoretical soundness and completeness. In essence, we "simulate" the mechanics of CPP using propositional formulas, so that we can easily integrate further constraints from the configuration space. Our lightweight approach does scale (the algorithm for generating the boolean formula grows linearly with respect to the number of blocks) to code bases of millions of code, as presented by our evaluation on the Linux kernel in Section 5.3.

Analyzing large scale software projects with transformation systems are related to our approach; DMS [3] proposed by Baxter is probably the most renowned one. In context of this framework, an approach has been published [4] that aims at simplifying CPP statements by detecting dead code and simplifying CPP expressions. Unlike our approach, this work uses concrete configurations to evaluate the expressions partially [22]. In contrast to that, our work does not require concrete values for CPP identifiers, but produces a logical model in form of a propositional formula that can either be evaluated directly or can be combined with further constraints like the ones extracted from the feature model. We believe that our approach would fit great in the DMS framework.

Other approaches extend the grammar of the C/C++ parser by CPP directives. Badros et al. [1] propose the PCp^3 framework to integrate *hooks* into the parser in order to perform many common software-engineering analyses. However, the focus is on mapping between unprocessed and preprocessed code, whereas our work aims at mapping towards higher level models. Garrido [19] extends the lexer with the concept of conditional abstract syntax trees (ASTs) which enables preprocessor-aware refactorings on CPP-based source files. A more sound implementation, a technique coined *variability aware parsing* that avoids using heuristics, is presented by Kästner et al. [23] These works basically integrate the CPP variability into tools for static analysis and thus, allow variability aware type-checking in the parser. Mainly because of implementation challenges, TypeChef focuses on a subset of Linux (namely arch-x86) and requires assistance in form of additional constraints by tools like [7] or this work presented in this article.

Model Checking and Type Systems The evaluation of variability from models is also related to our work. Czarnecki et al. [15] present an approach to transform logic formulas into *feature models*. With this as theoretical basis, the same group of researchers extends this work to show the semi-automatic reverse engineering of feature models from source code. [38]. Our approach does not require manual interaction for extracting the variability models. Generalized feature models [15], that is, feature models without domain-specific structural ordering, are sufficient for our cross-checks.

Benavides et al. [5] present several techniques for *reasoning* on feature models after transforming them into boolean formulas. Combined with reverse engineered feature models [38], these reasonings could be integrated with our work to enhance the crosschecking.

The crosschecking between different variability models is heavily related to our work. Metzger et al. [31] present several formalizations that allow for consistency checks between variability models. Czarnecki et al. [14] present an approach to checking the consistency of feature-based model templates, that is, checking consistency of feature models and model templates. Thaker et al. [46] present an approach to the safe composition of product lines, it is much related to our idea of crosschecking. However, all these approaches rely on novel programming paradims, such as *feature-oriented* software development or similar concepts. Our approach aims at allowing these concepts to be applied on *legacy software*, which in general does not use such novel techniques.

Tools for automatic finding of Bugs 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 [43, 24, 26, 17, 16]. Basically, all of these approaches extract some *internal model* about what the code *should* look like/behave and then match this model against the reality to find defects that are potential bugs. For instance, iComment [43] employs means of natural language processing to find inconsistencies between the programmer's intentions expressed in source-code comments and the actual implementation; [24] and colleagues use logic and probability to automatically infer specifications that can be checked by static bug-finding tools. However, none of the existing approaches take *configurability* into account when inferring the internal model. In fact, the existing tools are more or less *configurability agnostic* – they either ignore configuration-conditional parts completely, fall back to simple heuristics, or have to be executed on preprocessed source code. Thereby, important information is lost. Our analysis framework could be combined with these approaches to make them configurability-aware and to systematically improve their coverage with respect to the (extremely high) number of Linux variants. However, we also think that configurability has to be understood as a significant source of bugs in its own respect. Our approach does just that.

An interesting evolutionary analysis about source code defects in Linux has been given by Palix et al. in [34]. That work tries to reproduce a ten year old analysis on the Linux kernel [11] in order to investigate the evolutionary development of Linux across the last decade. As the old experiment did not state the exact configuration that was used, the old environment could only be approximated. The paper shows that configuration can (and does) affect the results of static analysis tools considerably. We take this anecdote as motivational lesson to further work on integration of configuration consistency checks into source-code analysis tools.

A reason that most existing source-code analysis tools for automated bug finding ignore configurability (more or less) might be that conditionally-compiled code tends to be hard to analyze in real-world settings. Many approaches for analyzing conditional-compilation usage have been suggested, usually based on symbolic execution. However, even the most powerful symbolic execution techniques (such as KLEE [9]) would currently not scale to the size of the Linux kernel. Hence, several authors proposed to apply transformation systems to symbolically simplify `CPP` code with respect to configurability aspects [4, 21]. Our approach is technically similar in the sense that we also analyze only the configurability-related subset of `CPP`. However, by "simulating" the mechanics of the `CPP` using propositional formulas [40], we can more easily integrate (and check against) other sources of configurability, such as the configuration-space model.

So far we have submitted 123 patches to the Linux community, which is a reasonably high number to confirm many observations of [20]: Patches for actively-maintained files are *a lot* more likely to receive responses. It really is worth the effort to figure out who is the principal maintainer (which is not always obvious) and to ensure that patches are easy reviewable and easy to integrate.

8 Summary and Conclusions

> *`#ifdef`'s sprinkled all over the place are neither an incentive for kernel developers to delve into the code nor are they suitable for long-term maintenance.*[16]

To cope with a broad range of application and hardware settings, system software has to be highly configurable. Linux 2.6.35, as a prominent example, offers 11,283 configurable features (KCONFIG items), which are implemented at compile time by 82,116 conditional blocks (`#ifdef`, `#elif`, ...) in the source code. The number of features

[16] Linux maintainer Thomas Gleixner in his ECRTS '10 keynote *"Realtime Linux: academia v. reality"*. `http://lwn.net/Articles/397422`

has more than doubled within the last five years! From the maintenance point of view, this imposes big challenges, as the configuration model (the selectable features and their constraints) and the configurability that is *actually* implemented in the code have to be kept in sync. In the case of Linux, this has led to numerous inconsistencies, which manifest as dead `#ifdef`-blocks and bugs.

We have suggested an approach for automatic consistency checks for compile-time configurable software. Our implementation for Linux has yielded 1,776 configurability issues. Based on these findings, we so far have proposed 123 patches (49 merged, 8 accepted, 15 acknowledged) that fix 364 of these issues (among them 20 confirmed new bugs) and improve the Linux source-code quality by removing 5,129 lines of unnecessary `#ifdef`-code. The performance of our tool chain is good enough to be integrated into the regular Linux build process, which offers the chance for the Linux community to prevent configurability-related inconsistencies from the very beginning. We are currently finalizing our tools in this respect to submit them upstream.

The lesson to learn from this paper is that configurability has to be seen as a significant (and so far underestimated) cause of software defects in its own respect. Our work is meant as a call for attention on these problems – as well as a first attempt to improve on the situation.

References

1. Greg J Badros and David Notkin. A framework for preprocessor-aware C source code analyses. *Software: Practice and Experience*, 30(8):907–924, 2000.
2. Don Batory. Feature-oriented programming and the AHEAD tool suite. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 702–703. IEEE Computer Society Press, 2004.
3. Ira D. Baxter. DMS: program transformations for practical scalable software evolution. In *Proceedings of the 5th International Workshop on Principles of Software Evolution (IWPSE'02)*, pages 48–51. ACM Press, 2002.
4. Ira D. Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE '01)*, page 281. IEEE Computer Society Press, 2001.
5. D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAISE '05)*, volume 3520, pages 491–503, Heidelberg, Germany, 2005. Springer-Verlag.
6. Thorsten Berger and Steven She. Formal semantics of the CDL language. Technical note, University of Leipzig, 2010.
7. Thorsten Berger, Steven She, Krzysztof Czarnecki, and Andrzej Wasowski. Feature-to-code mapping in two large product lines. Technical report, University of Leipzig (Germany), University of Waterloo (Canada), IT University of Copenhagen (Denmark), 2010.
8. Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '99)*, pages 45–53. IEEE Computer Society Press, May 1999.
9. Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th Symposium on Operating System Design and Implementation (OSDI '08)*. USENIX Association, 2008.
10. Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9), 1993.
11. Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88. ACM Press, 2001.
12. Yvonne Coady and Gregor Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In Mehmet Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 50–59, Boston, MA, USA, March 2003. ACM Press.
13. Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, May 2000.
14. Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE '06)*, pages 211–220. ACM Press, 2006.
15. Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, pages 23–34. IEEE Computer Society Press, Sept. 2007.
16. 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 *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72. ACM Press, 2001.

17. Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 449–458. ACM Press, 2000.
18. Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall, and Gilles Muller. THINK: A software framework for component-based operating system kernels. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 73–86. USENIX Association, June 2002.
19. Alejandra Garrido. *Program refactoring in the presence of preprocessor directives*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2005. Adviser-Johnson, Ralph.
20. Philip J. Guo and Dawson Engler. Linux kernel developer responses to static analysis bug reports. In *Proceedings of the 2009 USENIX Annual Technical Conference*. USENIX Association, June 2009.
21. Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Lagüe. C/C++ conditional compilation analysis using symbolic execution. In *Proceedings of the 16th IEEE International Conference on Software Maintainance (ICSM'00)*, page 196. IEEE Computer Society Press, 2000.
22. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
23. 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 *Proceedings of the 26th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*. ACM Press, October 2011.
24. Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: inferring the specification within. In *7th Symposium on Operating System Design and Implementation (OSDI '06)*, pages 161–176. USENIX Association, 2006.
25. Mario Latendresse. Rewrite systems for symbolic evaluation of c-like preprocessing. In *CSMR '04: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, page 165. IEEE Computer Society Press, 2004.
26. Zhenmin Li and Yuanyuan Zhou. PR-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference and the 13th ACM Symposium on the Foundations of Software Engineering (ESEC/FSE '00)*, pages 306–315. ACM Press, 2005.
27. Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE '10)*. ACM Press, 2010.
28. Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*, pages 215–228. USENIX Association, June 2009.
29. Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, pages 191–204. ACM Press, April 2006.
30. Anthony Massa. *Embedded Software Development with eCos*. New Riders, 2002.
31. Andreas Metzger, Patrick Heymans, Klaus Pohl, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines. In *Proceedings of the 15th IEEE International Conference on Requirements Engineering (RE'07)*, pages 243–253. IEEE Computer Society, 2007.
32. Yoann Padioleau, Julia L. Lawall, Gilles Muller, and René Rydhof Hansen. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys '08)*. ACM Press, March 2008.
33. Nicolas Palix, Julia Lawall, and Gilles Muller. Tracking code patterns over multiple software versions with Herodotos. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD '10)*, pages 169–180. ACM Press, 2010.
34. Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 305–318. ACM Press, 2011.
35. David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.
36. Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.
37. Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *4th Symposium on Operat-*

ing System Design and Implementation (OSDI '00), pages 347–360. USENIX Association, October 2000.

38. Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *Proceedings of the 33nd International Conference on Software Engineering (ICSE '11)*. ACM Press, May 2011.
39. Julio Sincero and Wolfgang Schröder-Preikschat. The Linux kernel configurator as a feature modeling tool. In Steffen Thiel and Klaus Pohl, editors, *Proceedings of the 12th Software Product Line Conference (SPLC '08), Second Volume*, pages 257–260. Lero Int. Science Centre, University of Limerick, Ireland, 2008.
40. Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Efficient extraction and analysis of preprocessor-based variability. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE '10)*. ACM Press, 2010.
41. Henry Spencer and Gehoff Collyer. #ifdef considered harmful, or portability experience with C News. In *Proceedings of the 1992 USENIX Annual Technical Conference*. USENIX Association, June 1992.
42. Diomidis Spinellis. A tale of four kernels. In Wilhem Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pages 381–390. ACM Press, May 2008.
43. Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: Bugs or bad comments?*/. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 145–158. ACM Press, 2007.
44. 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 *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys '11)*, pages 47–60. ACM Press, April 2011.
45. Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Dead or alive: Finding zombie features in the Linux kernel. In *Proceedings of the 1st Workshop on Feature-Oriented Software Development (FOSD '09)*, pages 81–86. ACM Press, 2009.
46. Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE '07)*, pages 95–104. ACM Press, 2007.
47. Matthias Urban, Daniel Lohmann, and Olaf Spinczyk. The aspect-oriented design of the PUMA C/C++ parser framework. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD '10)*, pages 217–221. ACM Press, 2010.
48. Christoph Zengler and Wolfgang Küchlin. Encoding the Linux kernel configuration in propositional logic. In Lothar Hotz and Alois Haselböck, editors, *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration 2010*, pages 51–56, 2010.