Feature Consistency in Compile-Time–Configurable System Software [*]

Facing the Linux $10{,}000$ Feature Problem

Reinhard Tartler, Daniel Lohmann, Julio Sincero, Wolfgang Schröder-Preikschat

Friedrich–Alexander University Erlangen–Nuremberg

{tartler, lohmann, sincero, wosch}@cs.fau.de

Abstract

Much system software can be configured at compile time to tailor it with respect to a broad range of supported hardware architectures and application domains. A good example is the Linux kernel, which provides more than $10{,}000$ configurable features, growing rapidly.

From the maintenance point of view, compile-time configurability imposes 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.

We suggest an approach to automatically check for configurability-related implementation defects in large-scale configurable system software. The configurability is extracted from its various implementation sources and examined for inconsistencies, which manifest in seemingly conditional code that is in fact unconditional. We evaluate our approach with the latest version of Linux, for which our tool detects $1{,}776$ configurability defects, which manifest 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 new bug.

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

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

General Terms Algorithms, Design, Experimentation, Management, Languages

Keywords Configurability, Maintenance, Linux, Static Analysis, VAMOS

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 [Parnas 1979]; they can (and have to) be configured at compile time to derive a concrete operating-system variant.

Configurability as a system property includes two separated – 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 [Spinellis 2008], despite all the disadvantages with respect to understandability and maintainability ("`#ifdef` hell") this approach is known for [Liebig 2010, Spencer 1992]. 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 [Massa 2002]; 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 builds upon previous work. In [Sincero 2010], we have introduced the extraction of a source-code variability model from C Preprocessor (CPP)-based software, which represents a building block for this work. A short summary of this approach is presented in Section 3.2.3. In this paper, we extend that work by incorporating other sources of variability and automatically (cross-) checking them for configurability-related implementation defects in large-scale configurable system software. We evaluate our approach with the latest version of Linux. In summary, we claim the following contributions:

1. It is the first work that shows the problem with the increasing configurability in system software that causes serious maintenance issues. (Section 2.2)
2. It is the first work that checks for configurability-related implementation defects under the consideration of both symbolic *and* logic integrity. (Section 3.1)
3. It presents an algorithm to effectively slice very large configuration models, which are commonly found in system software. This greatly assists our crosschecking approach. (Section 3.2.2)
4. It 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 4)

In the following, we first analyze the problem in further detail before presenting our approach in Section 3. We evaluate and discuss our approach in Section 4 and Section 5, respectively, and discuss related work in Section 6. 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, 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 configurability of the Linux kernel demands dedicated means 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 tool chain, which is depicted in Figure 1:

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 the thereby specified variability the Linux **configuration space**.

The following KCONFIG lines, for instance, describe the (optional) Linux feature to include support for hot CPU plugging in an enterprise server:

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

The HOTPLUG_CPU feature *depends on* general support for hot-pluggable hardware and must not be selected in a single-processor system.

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 output is a C header file (autoconf.h) and a Makefile (auto.make) that define a CONFIG_*<FEATURE>* preprocessor macro and make variable for every selected KCONFIG feature:

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

It's a convention that *all* and *only* KCONFIG flags are prefixed with CONFIG_.

❸ Features are implemented in the Linux source base. Whereas some coarse-grained features are enforced by including or excluding whole compilation units in the build process, the majority of features are enforced within the source files by means of the conditional compilation with

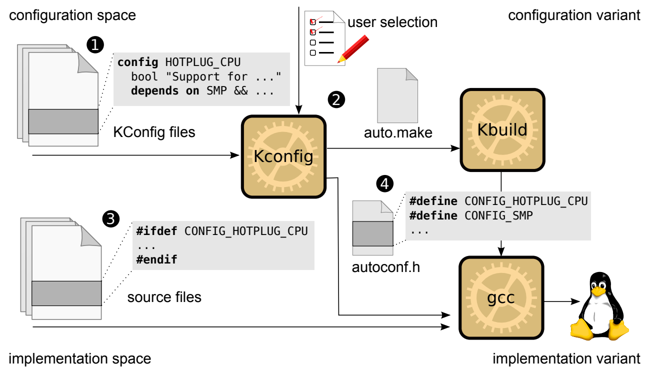


Figure 1. Linux build process (simplified).

the C preprocessor. A total of 27,166 source files contain 82,116 `#ifdef` blocks. We call the thereby implemented variability 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 GCC's "forced include"[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 separated, 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* and *logic*, 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 spaces, 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 analyze and fix. 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`

[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

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. The Approach

As pointed out in Section 2.2, 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. The general idea of our approach 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 2) 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**.

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 5.1.

3.1 Challenges in Analyzing Configurability Consistency – "What's wrong with GREP?"

Since version 2.6.23, Linux has included the (AWK/GREP-based) script `checkkconfigsymbols.sh`. This script is supposed to be used by maintainers to check for referential integrity between the KCONFIG model and the source code before committing their changes.

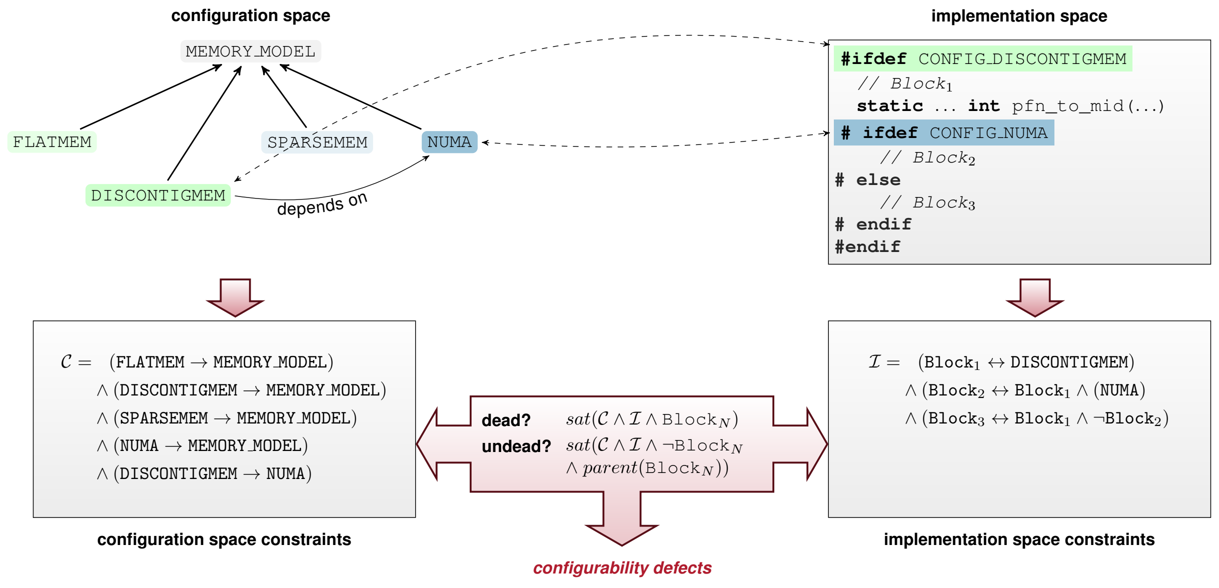
However, for Linux 2.6.30 this script reports 760 issues, among them the `CONFIG_CPU_HOTPLUG` issue discussed in Section 2.2, which remained in the kernel for more than six months. Apparently, kernel maintainers do *not* use this script systematically. While we can only speculate why this is the case, we have identified a number of shortcomings:

Accuracy. The output is disturbed by many **false positives**, defect reports that are not valid, but caused by some `CONFIG_` macros being mentioned in a historical comment. We consider this as a constant annoyance that hinders the frequent employment of the script.

Performance. The script can only be applied on the complete source tree. On reasonable modern hardware (Intel quadcore with 2.83 GHz) it takes over 7 minutes until the output begins. We consider this as too long and too inflexible for integration into the daily incremental build process.

Coverage. Despite its verbosity, the script misses many *valid* defects. **False negatives** are caused, on the one hand, by logic integrity issues, like the `CONFIG_NUMA` example from Figure 2, as logic integrity is not covered at all. However, even many referential integrity issues are not detected – the script does not deal well with KCONFIG's tristate options (which are commonly used for loadable kernel modules). We consider this as a constant source of doubt with respect to the script's output.

The lack of accuracy causes a lot of noise in the output. This, and the fact that the script cannot be used during incremental builds, renders the script barely usable. Most of these shortcomings come from the fact that it does not actually parse and analyze the expressed variability, but just employs regular expressions to cross-match `CONFIG_` identifiers. We conclude that the naïve GREP-based approach is (too) limited in this respect and that this problem has not been considered seriously in the past.



```
#ifdef CONFIG_DISCONTIGMEM
  // Block1
  static ... int pfn_to_mid(...)
# ifdef CONFIG_NUMA
    // Block2
# else
    // Block3
# endif
#endif
```

$$\mathcal{C} = (\texttt{FLATMEM} \rightarrow \texttt{MEMORY\_MODEL}) \wedge (\texttt{DISCONTIGMEM} \rightarrow \texttt{MEMORY\_MODEL}) \wedge (\texttt{SPARSEMEM} \rightarrow \texttt{MEMORY\_MODEL}) \wedge (\texttt{NUMA} \rightarrow \texttt{MEMORY\_MODEL}) \wedge (\texttt{DISCONTIGMEM} \rightarrow \texttt{NUMA})$$

$$\mathcal{I} = (\texttt{Block}_1 \leftrightarrow \texttt{DISCONTIGMEM}) \wedge (\texttt{Block}_2 \leftrightarrow \texttt{Block}_1 \wedge (\texttt{NUMA})) \wedge (\texttt{Block}_3 \leftrightarrow \texttt{Block}_1 \wedge \neg\texttt{Block}_2)$$

dead? $sat(\mathcal{C} \wedge \mathcal{I} \wedge \texttt{Block}_N)$

undead? $sat(\mathcal{C} \wedge \mathcal{I} \wedge \neg\texttt{Block}_N \wedge parent(\texttt{Block}_N))$

Figure 2. 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*.

3.2 Our Solution

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 represented by the following boolean formula:

$$\mathcal{V} = \mathcal{C} \wedge \mathcal{I} \tag{1}$$

$\mathcal{V} \mapsto \{0,1\}$ 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 2: The formula shown for dead blocks is satisfiable for $\texttt{Block}_1$ and $\texttt{Block}_2$, but not for $\texttt{Block}_3$. Therefore, $\texttt{Block}_3$ is considered to be *dead*; similarly the formula for undead blocks indicates that $\texttt{Block}_2$ is *undead*.

3.2.1 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.

```
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:     $\mathcal{PC} = presenceCondition(item)$
5:     for all $i$ such that $i \in \mathcal{PC}$ 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 $\mathcal{R}$
```

Figure 3. Algorithm for configuration model slicing

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.

In the following sections, we describe how we have approached these challenges to achieve good performance, flexibility, and, at the same time, a low number of false positives and false negatives.

3.2.2 Configuration Space

There are several strategies to convert configuration space models into boolean formulas [Benavides 2005, Czarnecki 2007]. 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.

Therefore, we have devised an algorithm that implements *model slicing* for KCONFIG. This allows us to generate submodels 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. Nevertheless, 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 code block under analysis.

Our slicing algorithm for this purpose is depicted in Figure 3. 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 4.1, this initial set of items will be taken from the #ifdef expressions.) The basic idea is to check the presence conditions 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.

In the first step (Line 1) the resulting set $\mathcal{R}$ is initialized with the list of input features. Then, the algorithm iterates until the working stack $\mathcal{S}$ is empty. In each iteration (Lines 2–11), a feature is taken from the stack and its *presence condition* is calculated through the function $presenceCondition(feature)$, which returns a boolean formula of the form $feature \rightarrow \varphi$. This formula represents the condition under which the feature can be enabled; φ is a boolean formula over the available features. Then, all features that appear in φ and have not already been processed (Line 6), are added to the working stack $\mathcal{S}$ and the result set $\mathcal{R}$. This algorithm always terminates; in the worst case, it will return all features and the slice will be exactly like the original model.

To implement our algorithm for Linux, we also have to implement the function $presenceCondition()$ that takes the semantic details of the KCONFIG language into account. In a nutshell, the KCONFIG language supports the definitions of five types of features. Moreover, the features can have a number of attributes like prompts, direct and reverse dependencies, and default values. The presence condition of a feature is the set of conditions that must be met, so that either the user can select it or a default value is set automatically. 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 condition 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. The formal semantics of the KCONFIG language has been studied elsewhere [Berger 2010, Zengler 2010]; such formalisms describe in detail how to correctly derive the presence conditions.

3.2.3 Implementation Space

Many techniques [Baxter 2001, Hu 2000] have been proposed to translate the CPP semantics to boolean formulas. However, for our approach, we need to consider the language features of CPP that implement conditional compilation only. Therefore we devised an algorithm [Sincero 2010] that is tailored in this respect in order to be precise and have good performance. In short, our algorithm generates a boolean formula that describes a source file by means of its *conditional com-*

pilation structures; it therefore examines the CPP 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 logical variable. We build the presence condition $\mathcal{PC}$ of the conditional block b_i as follows:

$$\mathcal{PC}(b_i) = \mathit{expr}(b_i) \wedge \mathit{noPredecessors}(b_i) \wedge \mathit{parent}(b_i) \quad (2)$$

Let b_i be a conditional block. In order for this block to be selected, it is required that its expression $\mathit{expr}(b_i)$ evaluates to true, in `#elif` cascades none of its predecessors are selected $\mathit{noPredecessors}(b_i)$, and for nested blocks, its enclosing `#ifdef` block $\mathit{parent}(b_i)$ is selected. If all these conditions are met, then CPP will necessarily select this block. Additionally, also the reverse is true: if the CPP selects the block, all these presence conditions need to hold. This results in a biimplication: $b_i \leftrightarrow \mathcal{PC}(b_i)$. Therefore, the formula for a file with several blocks can be built as follows:

$$\mathcal{F}_u(\vec{f}, \vec{b}) = \bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i) \quad (3)$$

where $\vec{f}$ is a vector containing all flags present in the file, and $\vec{b}$ is a vector containing a variable for each block of the file. An example is shown on the right hand side of Figure 2: 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 $\mathcal{F}_u(\texttt{[DISCONTIGMEM, NUMA]}, \texttt{[Block}_1\texttt{, Block}_2\texttt{, Block}_3\texttt{]}) = \mathcal{PC}(\texttt{Block}_1) \wedge \mathcal{PC}(\texttt{Block}_2) \wedge \mathcal{PC}(\texttt{Block}_3) = \mathcal{I}$

3.2.4 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 $\mathit{sat}(b_i \leftrightarrow \mathcal{PC}(b_i))$. **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: $\mathit{sat}(\mathit{feature} \rightarrow \mathit{presenceCondition}(\mathit{feature}))$.

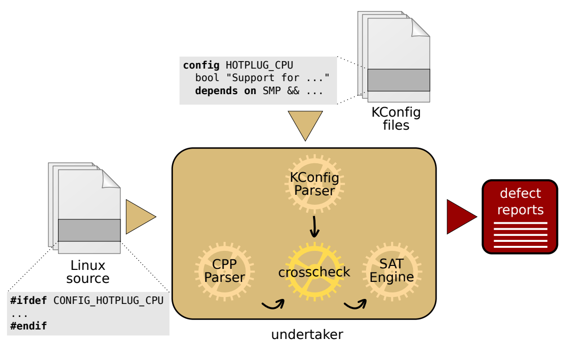

```
config HOTPLUG_CPU
  bool "Support for ..."
  depends on SMP && ...
```
```
#ifdef CONFIG_HOTPLUG_CPU
...
#endif
```

Figure 4. Principle of Operation

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 $\mathit{sat}((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V})$. **Referential** defects are caused by a *missing feature* (m) that appears in either the configuration or the implementation space *only*. That is, $\mathit{sat}((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V} \wedge \neg(m_1 \vee \cdots \vee m_n))$ is unsatisfiable.

Implementation-only defects have already been addressed in [Sincero 2010]; this paper focuses on the detection of configuration-implementation and referential defects in Linux. The defect analysis can be done using the *dead* and *undead* formulas as shown in the center of Figure 2.

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 belong to a sub-group of referential defects where the expression of the analyzed block $\mathit{exp}(b_i)$ is an atomic formula.

4. Evaluation

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.

4.1 Implementation for Linux

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 4: The different sources of variability are parsed and transformed into propositional formulas. For CPP parsing, we use the BOOST::WAVE[3] parsing library; for proper parsing of the `Kconfig` files, we have hooked up in the original Linux KCONFIG implementation. The outcome of these parsers is fed into the crosscheck-

[3] `http://www.boost.org`

ing engine as described in Section 3.2.4 and solved using the PICOSAT[4] package. The tool itself is published as Free Software and available on our website.[5]

Our tool scans each `.c` and `.h` file in the source tree individually. Unlike the script `checkkonfigsymbols.sh` as discussed in Section 3.1, 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 2 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:

```
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`, inhibits 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 3.2.3 and [Sincero 2010]); 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 B2 is the enclosing block. Lines 9ff. contain the extracted implications from KCONFIG (cf. Section 3.2.2). 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[6]:

```
diff --git a/arch/parisc/include/asm/mmzone.h b/arch/parisc/include/
      asm/mmzone.h
--- a/arch/parisc/include/asm/mmzone.h
+++ b/arch/parisc/include/asm/mmzone.h
@@ -35,6 +35,1 @@ extern struct node_map_data node_data[];

-#ifndef CONFIG_64BIT
#define pfn_is_io(pfn) ((pfn & (0xf0000000UL >> PAGE_SHIFT)) == (0
      xf0000000UL >> PAGE_SHIFT))
-#else
-/* io can be 0xf0f0f0f0f0xxxxxx or 0xffffffff0000000 */
-#define pfn_is_io(pfn) ((pfn & (0xf000000000000000UL >> PAGE_SHIFT))
       == (0xf000000000000000UL >> PAGE_SHIFT))
-#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.

Results. Table 2 (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. An exception of this rule is the code below `arch/`, which is architecture-specific by definition and checked against the configuration model of the respective architecture only.

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., [Engler 2001]) 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 2 takes around 15 minutes on a modern Intel quadcore with 2.83 GHz and 8 GB RAM. However, the implementation

[4] `http://fmv.jku.at/picosat/`

[5] `http://vamos.informatik.uni-erlangen.de/trac/undertaker/`

[6] `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 2. 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

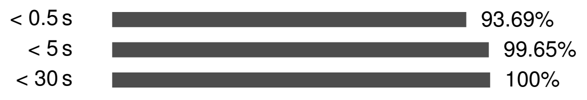


Figure 5. Processing time for 27,166 Linux source files

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 5 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.

4.2 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 6 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 2) 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 5.1. 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 so far 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 expect to find another 80+ configurability-related bugs in the Linux kernel.

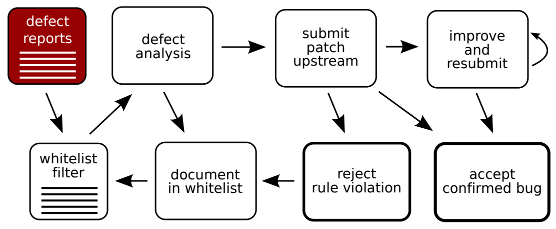


Figure 6. Based on the analysis of the defect reports, a patch is manually created and submitted to kernel developers. Based on the acceptance, we classify the defects that are fixed by our patch either as *confirmed rule violation* or *confirmed defect*.

Reaction of Kernel Maintainers. Table 3 lists the state of the submitted patches in detail; the corresponding defects are listed in Table 2 (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 7). We take this as indication that many of our patches are easy to verify and in fact appreciated.

Contribution to Linux. Table 3 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 3 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 fixes 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.

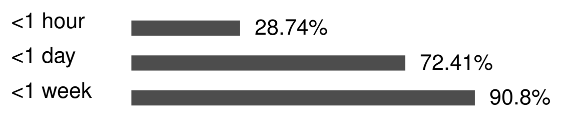


Figure 7. Response time of 87 answered patches

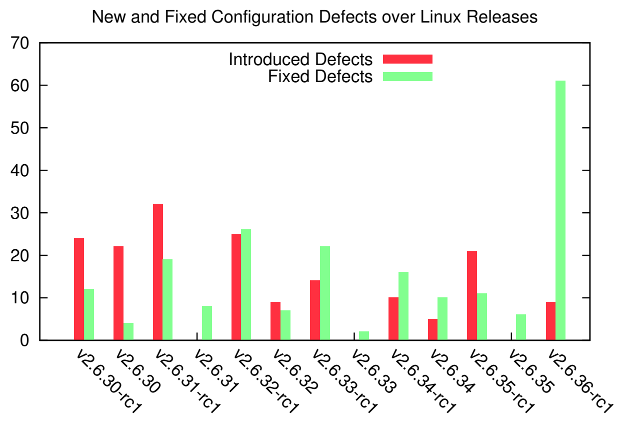


Figure 8. Evolution of defect blocks over various Kernel versions. Most of our work was merged after the release of Linux version 2.6.35.

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 8 depicts the impact of our work on a larger scale. To build this figure, we ran our tool on previous kernel versions and calculated the number of configurability defects that were *fixed* and *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.

5. 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.

5.1 Interpretation of the Feedback

About 57 of the 123 submitted patches were accepted without further comments. We take this as 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.

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

Causes for Defects. We have not yet analyzed the causes for defects systematically; doing this (e.g., using HERODOTOS [Palix 2010]) 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 [Engler 2001]). 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:[7]

```
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:

[7] `Documentation/SubmitChecklist` in the Linux source.

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.

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.

5.2 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 *exact* and avoids this kind of error. Since by construction we avoid false positives (sans implementation bugs), 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. So far we do not consider the (discouraged) 509 `#undef` and `#define CONFIG_` statements that we find in the code. However, these statements could possibly lead to incomplete results for only at most 4.51 percent of the $11{,}283$ KCONFIG items.

Another restriction of the current implementation is that we do 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 [Berger 2010, Zengler 2010] made our engineering difficult and contributed to the number of false negatives. At the time we conducted the experiment in Section 4, our implementation did not completely support the KCONFIG features *default value* and *select*. Meanwhile, we have fixed these issues in the undertaker, which increases the raw number of defects from $1{,}776$ to $2{,}972$.

In no case our approach resulted in a change that proposes to remove blocks that are used in production. However, in one case[8] 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.

Despite all potential sources of false negatives: Compared to the 760 issues reported by the GREP-based approach (including many false positives!, see Section 3.1), our tool already finds more than twice as many defects. As our approach prevents false positives, this has to be considered as a *lower bound* for the number of configurability defects in Linux!

5.3 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) [Massa 2002], whose expressiveness is comparable to KCONFIG.

KCONFIG itself is employed by more and more software families besides Linux. Examples include OpenWRT[9] or

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

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

BusyBox.[10] 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 FreeBSD), it would still be possible to extract feature identifiers and, hence, use our approach to detect symbolic configurability defects – which in the case of Linux account for 74 percent of all 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 [Tan 2007].

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 larger refactorings are commonly done tool assisted (e.g., Padioleau [2008]). However, refactoring tools are generally not aware of code liveness and suggest changes in dead code. Our approach contributes to avoiding such useless work.

5.4 Variability-Aware Languages

The high relevance of static configurability for system software gives rise to the question if we are in need of 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 implementation of configurability it is generally accepted that `CPP` might not be the right tool for the job [Liebig 2010, Spencer 1992]. Many approaches have been suggested for a better separation of concerns in configurable (system) software, including, but not limited to: object-orientation [Campbell 1993], component models [Fassino 2002, Reid 2000], aspect-oriented programming (AOP) [Coady 2003, Lohmann 2009], or feature-oriented programming (FOP) [Batory 2004]. However, in the systems community we tend to be reluctant to adopt new programming paradigms, mostly because we fear unacceptable runtime overheads and immature tools. C++ was ruled out of the Linux kernel for exactly these reasons.[11] The authors certainly disagree here (in previous work with embedded operating systems we could show that C++ class composition [Beuche 1999] and AOP [Lohmann 2006] 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 [Liebig 2010, Spinellis 2008].

With respect to modeling configurability, feature modeling and other approaches from the product line engineering domain [Czarnecki 2000, Pohl 2005] 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 the UNDERTAKER remain necessary.

6. Related Work

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 [Engler 2001, Ernst 2000, Kremenek 2006, Li 2005, Tan 2007]. 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 [Tan 2007] employs means of natural language processing to find inconsistencies between the programmer's intentions expressed in source-code comments and the actual implementation; Kremenek [2006] 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 takes *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.

A reason for the fact that existing source-code analysis tools 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 [Cadar 2008]) 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 [Baxter 2001, Hu 2000]. 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 [Sincero 2010], we can more easily integrate (and check against) other sources of configurability, such as the configuration-space model.

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

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

So far we have submitted 123 patches to the Linux community, which is a reasonably high number to confirm many observations of Guo [2009]: 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.

7. 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.[12]

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 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 out 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.

Acknowledgments

We would like to thank our students Christian Dietrich and Christoph Egger for their enduring and admiring work on the implementation and evaluation. We thank Julia Lawall for inspiring conversations and her helpful comments on an early version of this work. Many thanks go to the anonymous reviewers for their constructive comments that helped to improve this paper a lot as well as to our shepherd Dawson Engler.

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

References

[Batory 2004] 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.

[Baxter 2001] 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, Washington, DC, USA, 2001. IEEE Computer Society Press. ISBN 0-7695-1303-4.

[Benavides 2005] 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.

[Berger 2010] Thorsten Berger and Steven She. Formal semantics of the CDL language. Technical note, University of Leipzig, 2010.

[Beuche 1999] 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, St Malo, France, May 1999.

[Cadar 2008] 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)*, Berkeley, CA, USA, 2008. USENIX Association.

[Campbell 1993] 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.

[Coady 2003] 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.

[Czarnecki 2000] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, May 2000. ISBN 0-20-13097-77.

[Czarnecki 2007] 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.

[Engler 2001] 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, New York, NY, USA, 2001. ACM Press.

[Ernst 2000] 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, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-206-9.

[Fassino 2002] 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.

[Guo 2009] Philip J. Guo and Dawson Engler. Linux kernel developer responses to static analysis bug reports. In *Proceedings of the 2009 USENIX Annual Technical Conference*, Berkeley, CA, USA, June 2009. USENIX Association. ISBN 978-1-931971-68-3.

[Hu 2000] 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, Washington, DC, USA, 2000. IEEE Computer Society Press. ISBN 0-7695-0753-0.

[Kremenek 2006] 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, Berkeley, CA, USA, 2006. USENIX Association.

[Li 2005] 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, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-014-0.

[Liebig 2010] 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)*, New York, NY, USA, 2010. ACM Press.

[Lohmann 2009] 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, Berkeley, CA, USA, June 2009. USENIX Association. ISBN 978-1-931971-68-3.

[Lohmann 2006] 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, New York, NY, USA, April 2006. ACM Press.

[Massa 2002] Anthony Massa. *Embedded Software Development with eCos*. New Riders, 2002. ISBN 978-0130354730.

[Padioleau 2008] 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)*, Glasgow, Scotland, March 2008.

[Palix 2010] 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, New York, NY, USA, 2010. ACM Press. ISBN 978-1-60558-958-9.

[Parnas 1979] David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.

[Pohl 2005] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005. ISBN 978-3540243724.

[Reid 2000] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *4th Symposium on Operating System Design and Implementation (OSDI '00)*, pages 347–360, Berkeley, CA, USA, October 2000. USENIX Association.

[Sincero 2010] 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)*, New York, NY, USA, 2010. ACM Press.

[Spencer 1992] Henry Spencer and Gehoff Collyer. #ifdef considered harmful, or portability experience with C News. In *Proceedings of the 1992 USENIX Annual Technical Conference*, Berkeley, CA, USA, June 1992. USENIX Association.

[Spinellis 2008] 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, New York, NY, USA, May 2008. ACM Press. ISBN 987-1-60558-079-1.

[Tan 2007] 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, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-591-5.

[Zengler 2010] 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.