Lehrstuhl für Informatik 4

Verteilte Systeme und Betriebssysteme

Valentin Rothberg

Years of Variability Bugs in Linux – How to Avoid them

Masterarbeit im Fach Informatik

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme

Martensstr. 1 · 91058 Erlangen · Germany

www4.cs.fau.de

Years of Variability Bugs in Linux – How to Avoid them

Masterarbeit im Fach Informatik

vorgelegt von

Valentin Rothberg

angefertigt am

Lehrstuhl für Informatik 4

Verteilte Systeme und Betriebssysteme

Department Informatik

Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Dr. Daniel Lohmann**
Beginn der Arbeit:	**März 2014**
Abgabe der Arbeit:	**August 2014**

Abstract

Compile-time configurable system software requires a thorough design and implementation of the resulting variability. The Linux kernel constitutes a prominent example of such system software; Linux v3.12 ships over 12 000 configurable features with dozens of supported hardware architectures. Previous research shows that the Linux kernel suffers from hundreds of variability related defects. In this thesis, I shall present an empirical case study of variability defects in 24 versions of the Linux kernel, and present a tool that checks GIT commits for such bugs in order to avoid the introduction to the source code.

Kurzfassung

Konfigurierbare System Software erfordert einen modularen Softwareentwurf, der in vielen Fällen durch die Instrumentierung eines Kompilierers umgesetzt wird. Der Linux Betriebssystemkern sei hierbei mit über 12 000 konfigurierbaren Merkmalen (Version 3.12) und zahlreichen unterstützten Hardware-Architekturen besonders hervorgehoben. Die daraus entstehende Variabilität wurde vielfachs untersucht, und führt zu hunderten von Software-Fehlern und Variabilitätsdefekten. Diese Arbeit umfasst eine empirische Studie von Variabilitätsdefekten in 24 Versionen des Linux Kerns, und stellt ein Software-Programm vor, dass die Analyse von Variabilitätsdefekten in GIT Commits ermöglicht.

Contents

Chapter 1

Introduction

System software is a typical use case for compile-time configuration in order to tailor the system with respect to a broad range of supported hardware architectures, application domains and use cases. A major example is the Linux kernel. The Linux kernel constitutes an important subject to variability research with more than 12 000 configurable features in Linux v3.12 and almost 30 supported hardware architectures, including several sub-architectures. The variability management of such software is a difficult and error-prone task. Various layers and different programming languages need to be considered to get a holistic view of the feature definitions and the actual implementation of such in the source code. In case of Linux, variability is implemented by means of different tools and languages such as the C preprocessor (CPP), the GNU C compiler (GCC), KCONFIG (configuration system), KBUILD and MAKE (build system). A feature as well as its type and dependencies is defined in the KCONFIG language, whereas the actual feature implementation takes place in the source code which is conditionally compiled by means of KBUILD, the CPP and the GCC. The result is two different but related models in the feature distribution of Linux. The *configuration space* which constitutes the intended variability, and the *implementation space*, the implemented variability.

Tartler et al. [2011, 2012] and Sincero et al. [2010] show that the interaction of these models cause variability related defects and anomalies, that can lead to well manifested errors in the operating system [Abal et al., 2014]. Nevertheless, these works are mostly presenting problems – variability defects in Linux. Only few [Nadi et al., 2013] studied the cause and effect of defects by analyzing previously applied patches from Tartler et al. [2012] and by mining the stable GIT repository of the Linux kernel. Furthermore, only a few Linux versions have been subject of investigation. As a consequence, a holistic view on the problem is barely possible and leads to wrong assumptions of the problem's size and complexity in general.

Despite thorough research in the field of variability related defects in system software, and despite the fact that tools to automatically detect such defects exist for several years, the Linux kernel still suffers from a considerably high number of variability defects. Such defects constitute well manifested errors in the operating system, and potentially lead to NULL pointer dereferences, buffer overflows, and application programming interface (API) violations [Abal et al., 2014]. I blame state-of-the-art tools to be too unpractical for conventional Linux development, and, as a consequence, not being used by developers. Furthermore, only few research is dedicated to the analysis of variability defects which, in addition, present questionable results and do not cover all classes of variability defects.

In this thesis, I will present the *first empirical case study of variability related defects* and present a tool, UNDERTAKER–CHECKPATCH, that analyzes GIT commits to avoid the introduction of variability defects to the code base of Linux. First, we shall discuss the nature of variability related defects in Linux and how they arise during the build process of the Linux operating-system kernel (Chapter 2). Second, I will describe my approach of detecting, tracking, and analyzing variability defects over 24 versions of the Linux kernel and evaluate the results (Chapter 3) which will present a different quality and quantity of variability defects than described in previous research papers. In Chapter 4, I shall describe the implementation of the tool and how I use the results of the precedent case study to detect, analyze and report variability defects in GIT commits. Finally, I evaluate the tool on previously analyzed GIT commits and on the current development branch of Linux v3.16 (Chapter 5), and discuss the usability and benefit of the tool in daily Linux development (Chapter 6).

Chapter 2

Variability in Linux and Problem Statement

Operating systems and software systems in general demand a highly modularized design [Parnas, 1972]. Various software features, such as scheduling strategies or memory-management internals as well as the support of different hardware architectures and application domains, require substantial configuration mechanisms. The Linux kernel, a prominent example of such system software, contains more than 12 000 configurable features with a growing number by each new release. In general, we can identify several variation points in the Linux kernel that contribute to and make use of the system's variability:

Hardware Architectures The operating-system kernel is the closet software abstraction layer to communicate with the underlying hardware. The Linux kernel supports more than 60 different hardware platforms such as `ia32`, `DEC` and `sun-4`, which are organized to the correspondent architectures in the `arch/` directory in the source tree of the kernel. Linux v3.12 supports 30 different hardware architectures (e.g., `i386`, `ARM`, `Sparc`, `M68k`), including dozens of sub-architectures.

Subsystems The kernel subsystems are independent from the underlying hardware architecture, and can be classified as follows [Sincero et al., 2007]: *kernel* (architecture-independent kernel code, e.g. `IRQ`-handling, process scheduler, etc.), *fs* (file systems implementation), *init* (kernel initialization routines), *mm* (memory management), *sound* (sound subsystem), *block* (abstraction layer for disk access), *ipc* (inter-process communication), *net* (network protocols), and *lib* (library functions, such as `CRC` and `SHA1` algorithms). The *drivers* subsystem constitutes with 60 percent of source lines of code (v3.12) the biggest subsystem of the Linux kernel.

Configuration Options A *software feature*[1] is defined as a distinguishing characteristic of a software item (e.g., performance, portability, or functionality) and also helps to reduce code redundancy [Lopez-herrejon, 2005]. *Software features* can be used on a very fine-granular level to modularize smaller code artifacts, such as functions, data structures, or interfaces. However, modularization may also happen on a coarse-grained level to control the inclusion and exclusion of entire subsystems (modules) in the software system. In the context of operating systems, such software features can be closely related to hardware architectures (`x86`, `ARM`, etc.), hardware features (paging, interrupts, etc.) or devices (drivers). Other features may relate to a software module (e.g., scheduling). The features of the Linux kernel are defined in and distributed with the KCONFIG tool. A user can define a feature and its type and if desired, she can define further dependencies and constraints. Linux developers use such features to define logic constraints in KCONFIG, a processor's frequency value in form of an integer, a simple version string, or a boolean to conditionally compile source code units. Additionally, KCONFIG ships a *tristate* type; a special boolean switch to optionally compile software modules as runtime loadable kernel modules (LKMs).

Today, the Linux kernel runs on various devices in various application domains. It constitutes the operating system base for thousands of millions of Android smart phones and tablets as well as the base for over 97 percent of the top-500 supercomputers[2]. The rapidly growing variety of supported hardware and application domains requires substantial configuration mechanisms and a robust implementation of the resulting variability in the system. As a consequence, the number of configurable features in the Linux kernel nearly tripled since the year 2005 (Figure 2.1 on page 5). These configurable features distribute and implement variability in the Linux kernel to conditionally compile software modules and source-code artifacts. In Section 2.1, I will explain how this variability is realized in order to transfer the conditionally compiled source code into an executable operating-system kernel.

[1]IEEE Std. 829-1998
[2]`http://en.wikipedia.org/wiki/TOP500`, accessed 08-11-2014

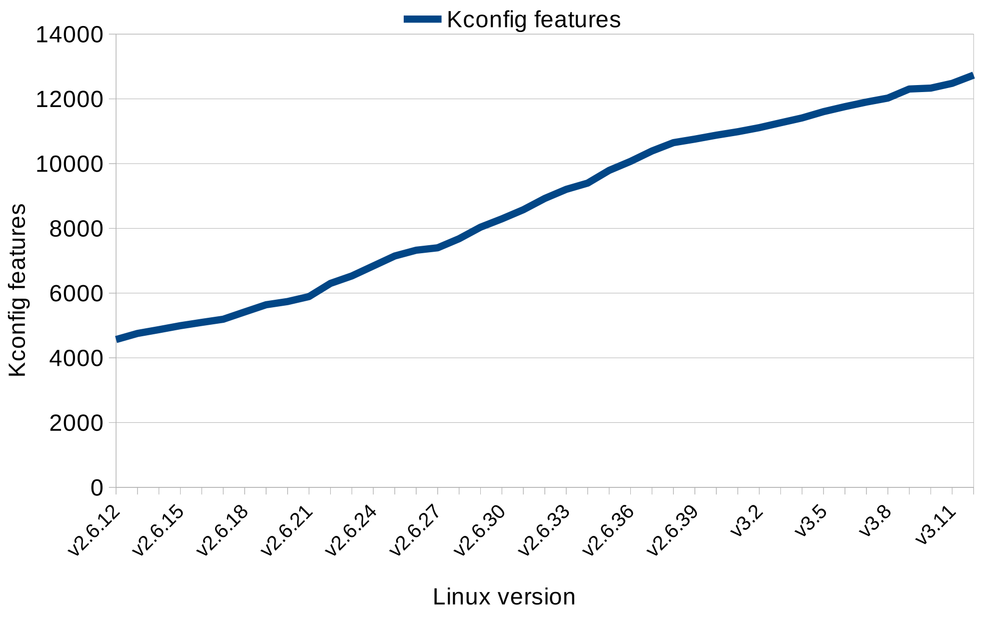


Figure 2.1 – KCONFIG feature evolution from Linux v2.6.12 (2005) until v3.12 (late 2013).

2.1 The Linux Build Process

The implementation of variability in the Linux kernel is realized by means of different tools and languages such as KCONFIG, KBUILD and MAKE, the GCC and the CPP. Variation points for the same feature may occur on different layers and of course in different source files and languages but, nevertheless, they remain interlinked as they derive from a common source: a KCONFIG configuration with which the kernel is compiled. This user-selected configuration dominates the build process as it determines the evaluation of features, as well as the compiling process; from including and excluding entire compilation units to fine-granular source code selection. As a matter of fact, feature implementations and the corresponding variability in Linux happen on very different but interacting layers, which we will examine separately to fully understand the variability implementation in the Linux kernel. In the following, I describe how Linux' variability is realized in the configuration system and the build system, and how the process instruments the user-selected kernel configuration to conditionally compile source code artifacts and software modules.

2.1.1 The Configuration System

Linux' features are designed and implemented in the KCONFIG language. KCONFIG defines features and their constraints (*intended variability*) and provides interfaces to specify, edit and manage a concrete kernel configuration. The user selects during the configuration process a Linux configuration in KCONFIG, which is accordingly saved to a `.config` named file. Linux v3.12 contains 12 764 of such features, defined in 133 222 lines spread over 1030 KCONFIG files. A feature is defined via an identifier, a type *(integer, hex, string, boolean, tristate)* and additional dependencies. Listing 2.1.1 depicts a KCONFIG feature definition from the x86 hardware architecture. `HOTPLUG_CPU` is the unique identifier in the *configuration space* of the architecture. This feature has a boolean value and depends on another feature, namely `SMP`. A dependency puts certain constraints on the decision if a feature can be selected or not. In general, a dependency is a *boolean formula* to conditionally evaluate if a KCONFIG feature can be selected or not. In this example, `HOTPLUG_CPU` can only evaluate true if `SMP` is true; `SMP` may entail further feature constraints. Note that feature dependencies in KCONFIG can be arbitrarily complex boolean formulas combined with arbitrarily complex conditional assignments in KCONFIG (if-else). Figure 2.2 on page 8 shows an exemplary feature selection saved in the `.config` file.

```
config HOTPLUG_CPU
    bool "Support for hot-pluggable CPUs"
    depends on SMP
    ---help---
    Say Y here to allow turning CPUs off and on. CPUs can be
    controlled through /sys/devices/system/cpu.
    ( Note: power management support will enable this option
    automatically on SMP systems. )
    Say N if you want to disable CPU hotplug.
```

Listing 2.1.1 – KCONFIG feature definition example from `arch/x86/Kconfig` (Linux v3.12). `HOTPLUG_CPU` can only evaluate true if `SMP` is true; `SMP` may entail further feature constraints. Note that feature dependencies in KCONFIG can be arbitrarily complex boolean formulas combined with arbitrarily complex conditional assignments in KCONFIG (if-else).

2.1.2 The Build System

The feature selection (`.config` file) is further interpreted by the build system, KBUILD, which constitutes *coarse-grained variability* by including and excluding complete translation units during the build process. The generated build product includes object files, loadable kernel modules and the bootable kernel image.

KBUILD transforms the KCONFIG-encoded .config file into further representations: a *Makefile* (auto.make) and a *CPP-header* (autoconf.h). The auto.make file is generated for KBUILD itself and contains the user selection in MAKE-syntax as illustrated in Listing 2.2. The "CONFIG_" string is a reserved prefix for feature identifiers in the *implementation space* and the build system. This representation is used afterwards during the build process in Makefiles and KBUILD files to conditionally include and exclude translation units. An exemplary Makefile-snippet of this process is depicted in Listing 2.1.2. Due to the expansion of MAKE-variables, the corresponding object files and the arm/ directory are added to internal lists. Depending on the value (obj-{y,n,m}), the objects are then statically compiled (y), compiled as a LKM (m), or excluded from compilation (n). This process can also be applied to conditionally add linker and compiler flags. The idea of this pattern dates back to 1997. It was proposed by Michael Elizabeth Castain[3] under the working title *"Dancing Makefiles"*. Linus Torvalds began shortly before the release of Linux v2.4 to adopt this concept to Makefiles of the Linux kernel [Dietrich et al., 2012].

```
obj-$(CONFIG_HOTPLUG_CPU)    += hotplug.o
obj-$(CONFIG_SMP)            += locks.o
obj-$(CONFIG_APM)            += apm.o
obj-$(CONFIG_ARM)            += arm/
```

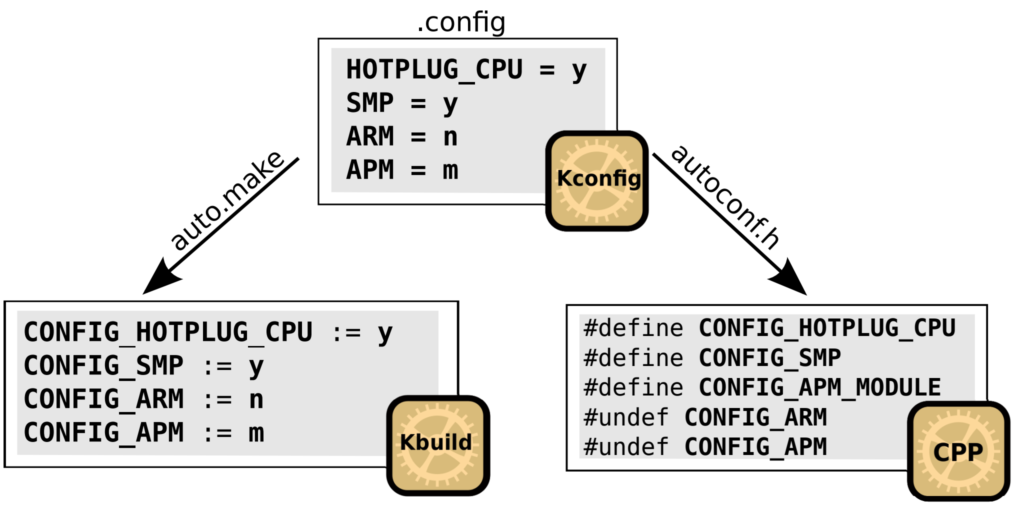
Listing 2.1.2 – Makefile/Kbuild example. Due to the expansion of MAKE-variables, the corresponding object files and the arm/ directory are added to internal lists. Depending on the value (obj-{y,n,m}), the lists are then statically compiled ("y"), compiled as a LKM ("m"), or excluded from compilation ("n").

2.1.3 The C Preprocessor

KBUILD implements variability on a *coarse-grained* level, where configurable feature-switches include and exclude entire compilation units. The C preprocessor makes use of CPP #ifdef macros on the level of *source code* and thereby implements *fine-grained variability*. Such blocks are part of the compilation units, if and only if the #ifdef condition of the block (*implementation space*) can evaluate true under the constraints of the selected KCONFIG features (configuration space). Linux v3.12 contains 34 238 of such variation points in C source files. However, an additional normalization step is required for an adequate treatment of *tristate features*. Many features, especially in the Linux driver subsystem, can be statically compiled into the kernel image or as loadable kernel modules. To ease the use of #ifdef items as well as to guarantee a consistent representation of tristates (*yes*, ***n**o*, ***m**odule*) across different layers, the representation of tristates is suffixed with "_MODULE" in the CPP

[3] https://lkml.org/lkml/1997/1/29/1

and MAKE syntax. During the build process, the build system (KBUILD) generates a C header file (`autoconf.h`) that contains this representation, and is used in a following step by the C preprocessor to conditionally include and exclude `#ifdef` blocks in the compilation units.



```
HOTPLUG_CPU = y
SMP = y
ARM = n
APM = m
```

```
CONFIG_HOTPLUG_CPU := y
CONFIG_SMP := y
CONFIG_ARM := n
CONFIG_APM := m
```

```
#define CONFIG_HOTPLUG_CPU
#define CONFIG_SMP
#define CONFIG_APM_MODULE
#undef CONFIG_ARM
#undef CONFIG_APM
```

Figure 2.2 – Linux feature representations in KCONFIG, KBUILD and the C preprocessor.

The Build Process

To conclude, KCONFIG dominates the entire build process as an entry point and interface to the user. KCONFIG features are distributed over the entire system and are used to implement variability in the following, summarized steps (Figure 2.3 on page 9):

❶ The configuration system (KCONFIG) defines features and their constraints (*intended variability*) and provides interfaces to specify, edit and manage a kernel configuration. Thereby the user selects a concrete *configuration variant* which is accordingly saved to the `.config` file.

❷ The build system (KBUILD) constitutes *coarse-grained variability* by including and excluding complete translation units in the build process. The generated build product includes object files, loadable kernel modules and the bootable kernel image. KBUILD translates the user-selected `.config` file into MAKE and CPP representations.

❸ The C preprocessor implements *fine-grained variability* on the source-code level by including and excluding `#ifdef` blocks in the preselected translation units. The outcome of the following compilation process (GNU C compiler) is the executable Linux kernel image – the *implementation variant*.

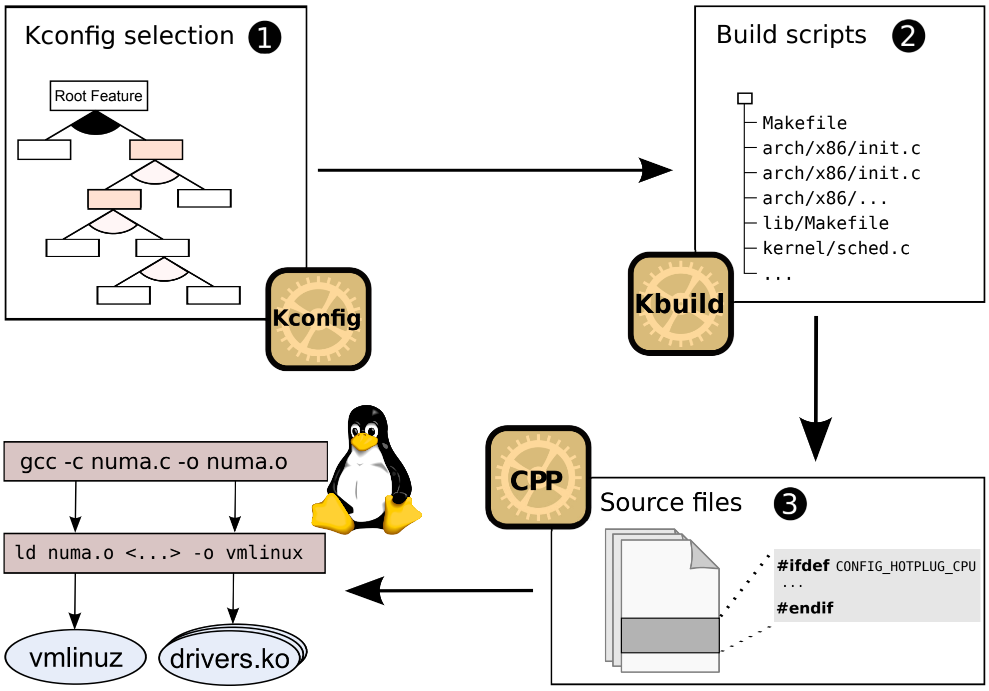


Figure 2.3 – The Linux build process: KCONFIG dominates the entire process as an entry point and interface to the user. KCONFIG features are distributed over the entire system and are used to implement *coarse-grained variability* by including and excluding entire complete translation units by KBUILD. The C preprocessor implements *fine-grained variability* by selecting units on the source-code level.

2.2 Variability Defects and Anomalies

As aforementioned, there are two different but related models in the Linux feature distribution and variability implementation. On one side, there is KCONFIG spanning the *configuration space* which describes the *intended variability*. The actual implementation of such takes place in the *implementation space* by means of KBUILD, the C preprocessor and the GNU C compiler. The interaction of those two spaces is a common cause of variability related defects in the Linux kernel [Tartler et al., 2011]. Such defects can be classified as **symbolic integrity violations** (Section 2.2.1) and **logic integrity violations** (Section 2.2.2).

2.2.1 Symbolic Integrity Violations

A *symbolic integrity violation* is any reference on an undefined KCONFIG feature. Undefined KCONFIG features always evaluate false and can thereby cause contradictory constraints in KCONFIG itself, and `#ifdef` blocks to never or always be selected in the compilation process.

Listing 2.2.1 shows a PATCH from Tartler et al. [2011] that fixes a *symbolic integrity violation* in the source code by renaming the referenced KCONFIG item. The previously referenced identifier `CONFIG_CPU_HOTPLUG` did not reference a defined KCONFIG feature (typo) so that the `#ifdef` block has never been part of a compilation unit. As such defects are caused by misleading references, they are also called *referential anomalies* [Nadi et al., 2013] or *missing defects* (see Section 3.1.2) and will be used synonymously in my thesis.

```
diff --git a/kernel/smp.c b/kernel/smp.c
index ad63d85..94188b8 100644
--- a/kernel/smp.c
+++ b/kernel/smp.c
@@ -57,7 +57,7 @@ hotplug_cfd(struct notifier_block *nfb, ...
                return NOTIFY_BAD;
            break;

-#ifdef CONFIG_CPU_HOTPLUG
+#ifdef CONFIG_HOTPLUG_CPU
        case CPU_UP_CANCELED:
        case CPU_UP_CANCELED_FROZEN:
```

Listing 2.2.1 – Patch-snippet for a symbolic integrity violation in `kernel/smp.c` (Linux v2.6.30). The previously referenced identifier `CONFIG_CPU_HOTPLUG` did not reference a defined KCONFIG feature (typo) so that the `#ifdef` block has never been part of a compilation unit.

In general, there are various possibilities to fix such defects in the *implementation space* as well as in the *configuration space*. Nadi et al. [2013] state that up to 54 percent of symbolic bug fixes take place in the source code (CPP patches), whereas 3 percent are fixed by changes to KCONFIG files. Figure 2.4 on page 11 shows the evolution of such defects between Linux v2.6.29 and v3.12. We see that symbolic defects are rapidly repaired in the first seven Linux versions (v2.6.30 – v2.6.36), what I attribute to the results of previous research work and related contributions [Tartler et al., 2010, 2011, 2012]. My observation of decreasing *symbolic anomalies* in this version interval aligns with previous results from Nadi et al. [2013].

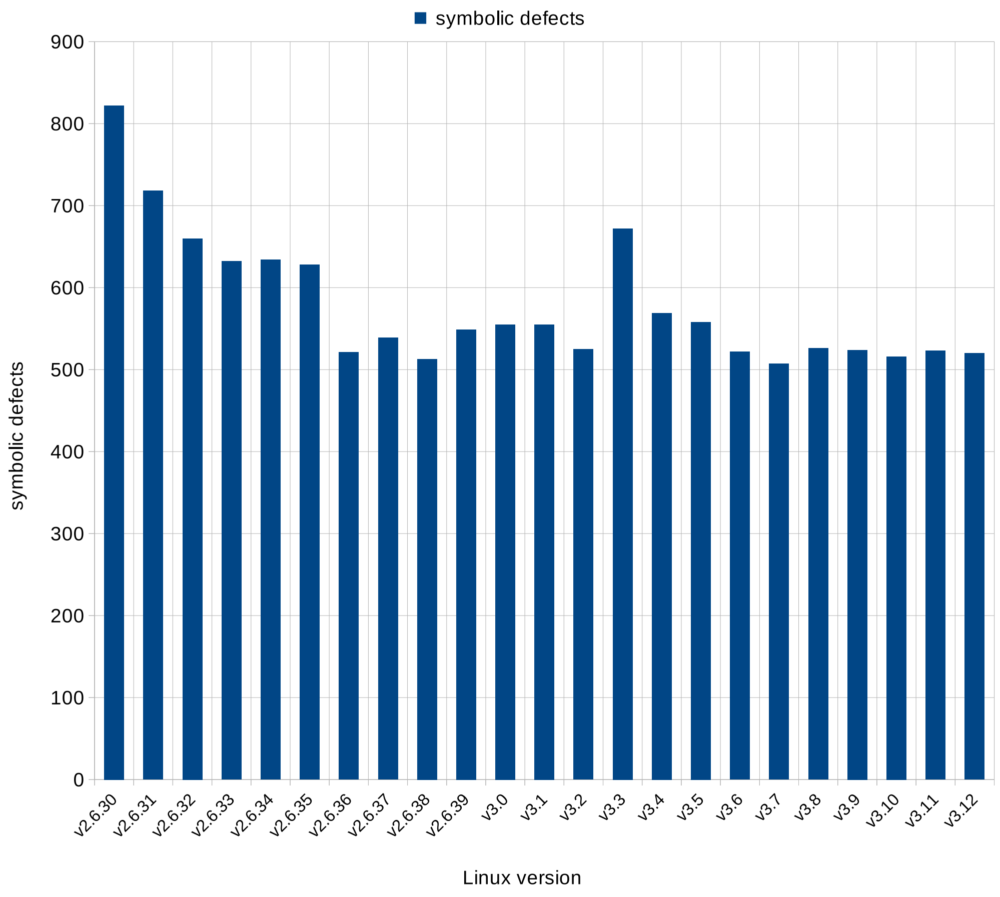


Figure 2.4 – Evolution of symbolic integrity violations in Linux

2.2.2 Logic Integrity Violations

A *symbolic integrity violation* indicates a mismatch of the configuration and the implementation space with respect to a *feature identifier*, and thereby violates the referential integrity of the system. However, variability related issues also occur on the level of *feature constraints*; so called *logic integrity violations*. A *logic integrity violation* describes contradictory KCONFIG and CPP constraints which lead to (a) unintentionally dead or undead KCONFIG features, or (b) to dead or undead `#ifdef` blocks. Dead features and `#ifdef` blocks always evaluate false, undead always evaluate true.

Listing 2.2.2 shows the fix of a *logic integrity violation*. The PATCH removes the entire `#ifdef` block but keeps the source code of the `#else` block – the dead block is removed, the code of the undead block is made unconditional. The cause of the variability defect in this case is that the displayed code artifacts are further enclosed by another `#ifdef` block which references `CONFIG_DEBUG_OBJECTS_RCU_HEAD`. This

KCONFIG feature depends on CONFIG_PREEMPT so that the precondition of the #ifndef block is a contradiction. As a consequence, the displayed #ifndef block always evaluates false.

```
diff --git a/kernel/rcupdate.c b/kernel/rcupdate.c
index a23a57a..afd21d1 100644
--- a/kernel/rcupdate.c
+++ b/kernel/rcupdate.c
@@ -215,10 +215,6 @@ static int rcuhead_fixup_free(void *addr, ...
                  * If we detect that we are nested in a RCU rea...
                  * section, we should simply fail, otherwise we...
                  */
-#ifndef CONFIG_PREEMPT
-               WARN_ON(1);
-               return 0;
-#else
                if (rcu_preempt_depth() != 0 || preempt_count()...
                    irqs_disabled()) {
                        WARN_ON(1);
@@ -229,7 +225,6 @@ static int rcuhead_fixup_free(void *addr, e...
                rcu_barrier_bh();
                debug_object_free(head, &rcuhead_debug_descr);
                return 1;
-#endif
        default:
                return 0;
        }
```

Listing 2.2.2 – Excerpt of patch fixing a logic integrity violation in ./kernel/rcupdate.c in Linux v2.6.30. The cause of the variability defects is that the displayed code artifacts are further enclosed by another #ifdef block, which references CONFIG_DEBUG_OBJECTS_RCU_HEAD. This KCONFIG feature depends on CONFIG_PREEMPT so that the precondition of the #ifndef block is a contradiction. As a consequence, the displayed #ifndef block always evaluates false.

In principle, logic defects can stem from a tautology or a contradiction in the *configuration space* (dependencies and constraints), a mistake in the build system [Dietrich et al., 2012] or from contradictory preconditions of #ifdef blocks (*implementation space*). As a consequence, there are two subclasses of *logic integrity violations*, that indicate which of both spaces is involved in the defect. **Kconfig defects** are caused by contradictions that occur in the interaction of constraints from KCONFIG with constraints from the #ifdef blocks, so that both spaces are involved. **Code defects** indicate that the defect is caused by a contradiction in the CPP structure of #ifdef blocks. Such defects solely affect the *implementation space* and thereby do not require any additional information from the *configuration* *space*, namely KCONFIG. *Code defects* mostly stem from redundant double-checks of KCONFIG features in nested `#ifdef` blocks.

Figure 2.5 depicts the evolution of *logic defects* in the Linux kernel. In general, there are much more *symbolic* than *logic defects*. Nevertheless, the percentage share of *logic defects* is growing (19 percent in v2.6.30, 26 percent in v3.12) since the number remains at a comparatively constant level of two hundred defects per version, whereas the number of *symbolic defects* is decreasing at the beginning of the investigated period of time.

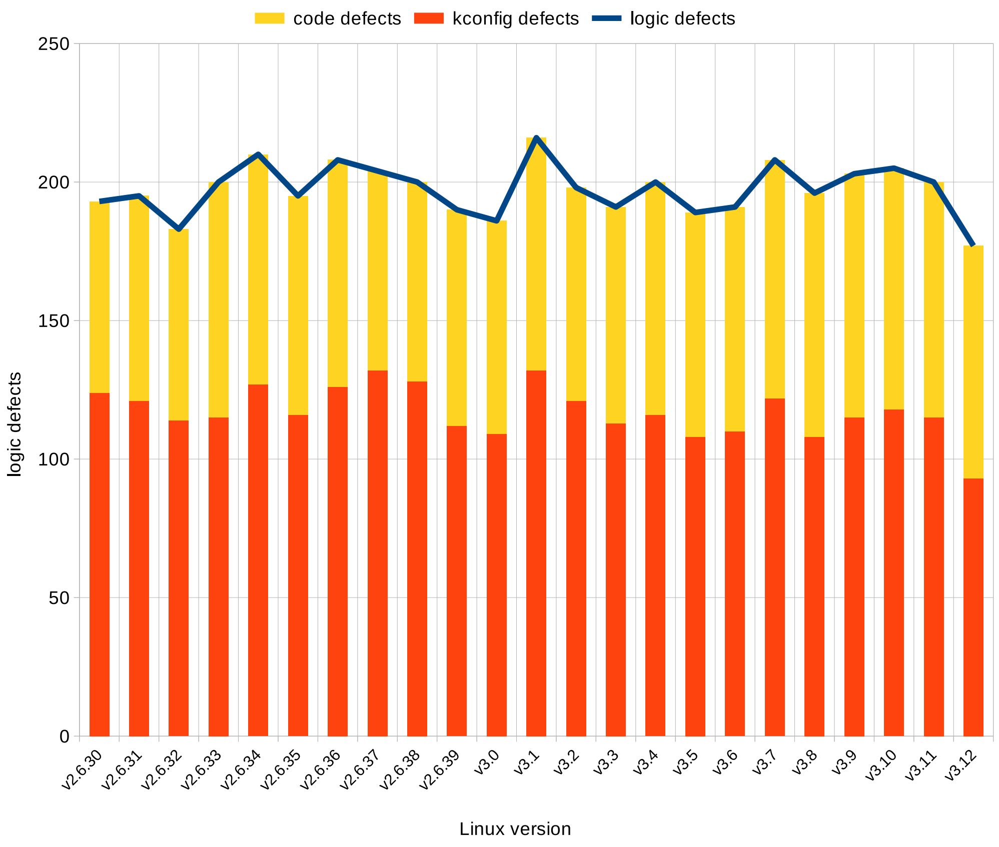


Figure 2.5 – Evolution of logic integrity violations in Linux from v2.6.30 until v3.12.

2.3 Problem Statement and Related Work

Variability related defects present a delicate and error-prone issue to the Linux community: "The tricky parts these days are configuration issues, i.e. code that fails to build for certain configurations, due to various reasons (forgot to handle a case in another `#ifdef` branch, code inside vs. outside `#ifdef`, different indirect includes on different architectures [...]" [4]. Such bugs are hard to analyze and oftentimes manifest in critical errors to the system, such as `NULL` pointer dereferences, buffer overflows, or API violations [Abal et al., 2014]. Despite the fact that tools to detect such defects exist for more than four years [Tartler et al., 2010], the Linux community still suffers under the problem of managing the system's variability and its defects. I blame the usability of current research tools and the underlying approaches to be too unpractical for a Linux developer's work flow. I believe that tools which allow a seamless integration into the work flow of Linux development, or tools that can run independently on dedicated servers, can be a significant contribution in the context of variability related defects. As a consequence, it is my desire to develop a tool that can actually be used by the Linux community to reveal and analyze variability defects. I want to achieve this goal by developing a tool, that checks PATCH files for variability related defects.

The Linux kernel is developed using the distributed source-code management system GIT[5] which bases on the idea of *patching* source code with so called GIT commits. As matter of fact, every new line of source code is added via the integration of GIT commits. Consequently, every Linux developer is forced to use this tool; I consider GIT as the perfect candidate to achieve a seamless integration into the work flow of a developer. To conclude. I will present a tool that is (a) checking PATCH files for variability related defects, (b) that integrates seamlessly into the work flow of a Linux developer and (c) that is able to further analyze a defect's cause to help to repair the defect.

However, only few research is dedicated to the analysis of how variability bugs are caused and fixed in the Linux kernel – this knowledge is mandatory for my approach. Nadi et al. [2013] took previously applied patches from Tartler et al. [2011] to investigate how variability defects can be repaired. By manually analyzing the *106 patches* they found out that over *90 percent* of patches are related to dead-code removals. The remaining *10 percent* add, edit or delete KCONFIG features in the source code as well as in KCONFIG files. Nevertheless, they all fix bugs in the Linux source code. In addition to the analysis, the authors describe a mechanism to semi-

[4]`http://lists.linuxfoundation.org/pipermail/ksummit-discuss/2014-June/001010.html`, Geert Uytterhoeven on Ksummit-discuss

[5]`https://git.kernel.org/`

automatically map defects in several Linux versions to GIT commits that introduce or repair the corresponding variability defects. In this case, the paper focuses on *referential integrity violations*. Besides the fact that there is no such research work on *logic integrity violations*, the presented work suffers from several drawbacks:

The presented data bases on patches from Tartler et al. [2011], where most of the submitted patches entered the mainline kernel tree during the merge window of Linux version v2.6.36. Due to the short time frame, I do not consider this data as sufficiently representative for todays Linux kernel development in general. Personal correspondence with Greg Kroah-Hartman on the topic of *variability related defects* manifested my doubts on older data, since (a) today, the kernel team includes developers looking for KCONFIG related defects, and (b) kernel developers restructured many `#ifdef` blocks until today. Furthermore, the authors use a deprecated version of the UNDERTAKER[6] tool that they instrument to report variability related defects in the Linux source. This version of the UNDERTAKER tool reported a considerably high number of hundreds of false positives and also failed in the detection of some defects (see Chapter 3, and Chapter 6). As a consequence, I assume the presented data to be contaminated with false positives. For my approach, I am in need of (a) a more holistic view on *all* variability related defects, and (b) I am in need of current data from state-of-the-art research tools.

The results of the paper show that the used approach to mine the main GIT repository of Linux *fails in the automatized mapping from defects to patches*. A big share of *referential anomalies* could not be mapped to GIT commits at all, whereas 73 percent of mapped commits are false positives. As a consequence, I need to develop a more robust approach without threats to validity.

The paper *does not examine logic integrity violations*. This defect class describes bugs that occur due to contradictions in preconditions of `#ifdef` blocks or KCONFIG itself. As a consequence, I need to present a first case study about the analysis of both defect classes. I expect that an analysis of *logic defects* will reveal a different image than *symbolic defects*. *90 percent* of referential anomalies of Tartler et al. [2011] are fixed by simply removing dead code. *Logic defects* may be repaired without removing code at all.

Furthermore, the authors state that "KCONFIG *patches are those which modify a feature definition in* KCONFIG "[Nadi et al., 2013]. However, variability related defects

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

can also be caused by broken references in the constraints of a KCONFIG feature[7]. Furthermore, the authors claim that *symbolic defects* can only be caused by changes to KCONFIG files. I also disagree with that statement, as the authors completely ignore the fact that changes to the source code may cause such defects as well.

All in all, the mentioned drawbacks of Nadi et al. [2013] suggest a different, more holistic approach on the analysis of variability defects in Linux. On one side, statistics based on 106 patches (of one source) can be highly imprecise. More patches of more Linux versions need to be taken into account so that we get a better understanding of the qualities and quantities of the defect classes. On the other side, only a few Linux versions (10) have been subject of analysis, which gives a rather small snapshot of one year and nine months of Linux GIT history. This short interval has a significant impact on the precision of presented data such as the lifespans of defects. Consequently, more Linux versions and a better tracking of defects is needed.

In the following chapters, I will present first how I address the difficulties in generating empirical data on variability related defects (Section 3.1), second how we can analyze and map such defects to GIT commits (Section 3.1.4), and third I will describe (Chapter 4) and evaluate (Chapter 5) the tool.

[7]In the context of the paper [Nadi et al., 2013], a modification of feature definition is described as renaming the identifier or deleting the feature itself. Other papers may attribute constraints to the feature's definition as well.

Chapter 3

Feature-Consistency Analysis in Linux

Before the implementation of a tool that checks PATCH files for variability related defects, we need to gain a thorough understanding in the detection and analysis of such. In principle, we have to answer the following question:

⇒ ***How are variability related defects caused and fixed in the Linux kernel?***

We need this data, (a) to understand the actual needs of developers and how we can help to avoid and fix variability defects, and (b) to implement metrics and algorithms to automatically analyze such defects. Until now, only few research papers are dedicated to an *empirical* analysis of variability related defects in the Linux kernel. As aforementioned in Section 2.3, I do not consider the results of previous research [Nadi et al., 2013] to be a sufficiently meaningful data base. Consequently, I need to develop a more robust approach to generate empirical data on variability related defects in the Linux kernel.

This chapter describes my overall approach to generate the desired data. I make use of existing research which I improve in order to gain in depth knowledge of variability bugs in Linux. First, I describe my approach and the tools I use to detect variability related defects in the Linux kernel, how I track them over different Linux versions and how I map them to the defect introducing and fixing GIT commits. After that, I present the results of the generated data and describe the causes and complications of the aforementioned defect classes. Finally, I conclude the examined results and explain how they can be used to implement a tool that is checking GIT commits for variability related defects.

3.1 Approach

Tartler et al. [2011] present an approach to automatically check for configurability related implementation defects in large-scale configurable system software. They implemented a tool, the UNDERTAKER[8], to reveal variability related defects in the source code of the Linux kernel. Figure 3.1 illustrates the working process of the tool, which bases on the extraction of a boolean formula from the *configuration space* (KCONFIG) and from the CPP structure of the source files. The formulas are then used to instrument a satisfiability solver (SAT) to detect contradictions and tautologies in the preconditions of CPP `#ifdef` blocks. Thereby the UNDERTAKER tool detects *symbolic integrity violations* (Section 2.2.1) and *logic integrity violations* (Section 2.2.2).

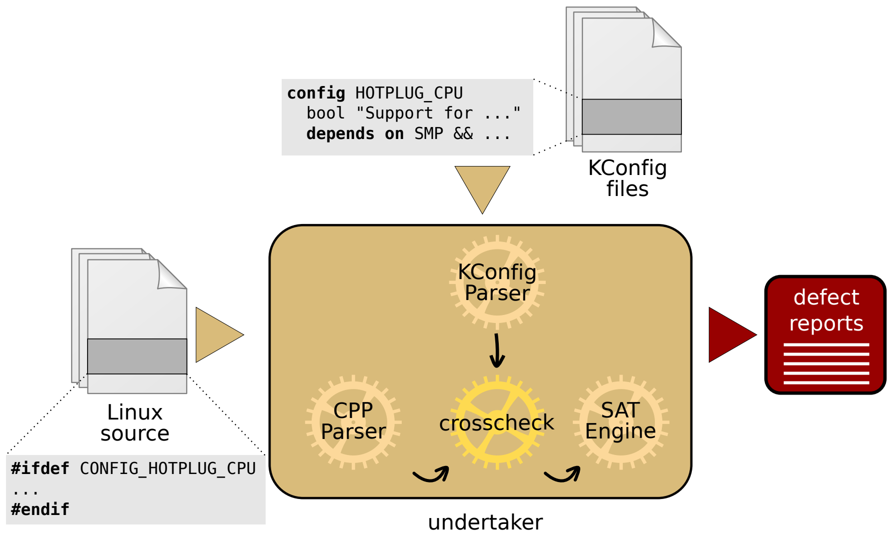


Figure 3.1 – Defect analysis with the UNDERTAKER tool: after extracting variability artifacts from the *configuration space* (KCONFIG) and the *implementation space* (source code), the tool instruments a SAT solver to reveal variability related defects.

All in all, the UNDERTAKER tool provides the functionality to analyze the source code of the Linux kernel and to report variability related defects. Consequently, I instrument this tool to detect such defects. The boolean formulas base on the extraction of variability models from the *configuration space* (KCONFIG) of Linux. These models are needed to reveal contradictions with a *SAT* solver. In the following section, I shall explain how we can extract these models in order to detect variability defects in Linux.

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

3.1.1 Extraction of Variability Models

The UNDERTAKER tool detects variability defects by instrumenting a SAT solver. This process requires the extraction of variability models from the *configuration space* of Linux; the models map all feature definitions, dependencies and constraints to boolean formulas. The variability extractor, UNDERTAKER-KCONFIGDUMP, generates a model for each supported hardware architecture within a given Linux tree. This process takes around 2 minutes of computation on a today's workstation with an Intel Core i7 Quad-Core processor and 16 GB of RAM. The generated models have a size ranging from 44 MB in Linux v2.6.29 to 110 MB in Linux v3.12.

In the context of *tracking defects* over time, the most accurate approach is to implement a mechanism to follow defects *from one commit to another*. By doing so, we have an exact mapping to the defect introducing and fixing GIT commit and the exact life spans of the defect. Consequently, we need to extract new variability models as soon as GIT commit changes KCONFIG files. However, the consumption of memory resources and the computational time to extract variability models are limiting factors, as they make an exact tracking of defects from one commit to another impossible in an acceptable period of time. Every change to a KCONFIG file is varying the *configuration space* of the system, and thereby requires the generation of new models. Between Linux v2.6.29 and Linux v3.12, *13 807 commits* change *18 058* KCONFIG *files*. As a matter of fact, the generation of models for all *13 807 commits* would take *more than 25 days* and result in an estimated consumption of *1.3 petabyte of memory resources*. Both, the computational time to generate the models and the memory space to store them are complicating the analysis approach.

I solve this problem by considering stable Linux versions as discrete points in the period of time that is target of analysis. This reduces the problem size to the computation of *24 models*, what takes averagely 45 minutes consuming 1.8 GB of memory space. Thereby only these 24 discrete versions are subject to a following dead analysis. As defects cannot be tracked from one commit to another, I need to find the defect causing and defect fixing GIT commits in a following step.

3.1.2 Defect Analysis

As soon as all variability models are extracted, we can trigger the UNDERTAKER tool to generate defect reports for each Linux version. A *defect report* includes the source-file location of the `#ifdef` block and the contradictory boolean formula that causes the defect. This formula contains the structure of CPP macros in the source file (*implementation space*) as well as the constraints and dependencies of referenced KCONFIG identifiers (*configuration space*). The reports cover the following defect classes:

Missing Defects

Missing defects are symbolic integrity violations (Section 2.2.1). *"Missing"* thereby indicates that at least one of the referenced KCONFIG features (literals in the formula) is not defined – a referential integrity violation of the *configuration space*. *Missing defects* can stem from misspelled feature identifiers in `#ifdef` macros (e.g., typos) or from references on features that are not defined in the *configuration space*. Such defects can be caused by broken references in the source code (`#ifdef CONFIG_UNDEFINED`), as well as in KCONFIG files (`depends on UNDEFINED`), see Listing 3.2.1. In general, such defects are caused by the interaction of the *configuration space* with the *implementation space* with respect to a KCONFIG feature-identifier.

```
config SERIAL_8250_RM9K
   bool "Support for MIPS RM9xxx integrated serial port"
   depends on SERIAL_8250 != n && SERIAL_RM9000
   select SERIAL_8250_SHARE_IRQ
   help
     Selecting this option will add support for the integrated serial
     port hardware found on MIPS RM9122 and similar processors.
     If unsure, say N.
```

Listing 3.1.1 – KCONFIG snippet of `drivers/tty/serial/8250/Kconfig` (Linux v3.8). The displayed KCONFIG feature depends on `SERIAL_RM9000`, which has been removed by a GIT commit. As the referenced feature is not defined in the *configuration space*, the commit caused a referential integrity violation. Note that all references on `SERIAL_8250_RM9K` will result in a *missing defect*.

Kconfig Defects

Kconfig defects are logic integrity violations (Section 2.2.2). *"Kconfig"* thereby indicates that the contradictory formula grounds in the feature constraints and feature definitions in KCONFIG itself. A *kconfig defect* occurs when KCONFIG features are referenced in the source code (*implementation space*), such that the preconditions of referencing `#ifdef` blocks result in a tautology or contradiction; the `#ifdef` blocks are then always or never selected in the compilation process. Furthermore, *kconfig defects* may also be caused by a contradiction or tautology in KCONFIG (*implementation space*), so that features evaluate dead or undead. Listing 3.1.3 depicts a *kconfig defect*, where the (boolean) preconditions of two blocks cause a contradiction and a tautology, depending on the affected `#ifdef` macro. As a consequence, this defect class affects both, the *implementation* and the *configuration space*.

```
#ifdef CONFIG_DEBUG_OBJECTS_RCU_HEAD
...
#ifndef CONFIG_PREEMPT
    WARN_ON(1);
    return 0;
#else
    if (rcu_preempt_depth() != 0 || preempt_count() != 0 ||
            irqs_disabled()) {
        WARN_ON(1);
        return 0;
    }
    rcu_barrier();
    rcu_barrier_sched();
    rcu_barrier_bh();
    debug_object_free(head, &rcuhead_debug_descr);
    return 1;
#endif
...
#endif
```

Listing 3.1.2 – This source-code snippet of `kernel/rcupdate.c` (Linux v2.6.36) shows a *kconfig defect*, where the inner `#ifndef` block is dead and the corresponding `#else` block is undead. `DEBUG_OBJECTS_RCU_HEAD` depends on `PREEMPT`, so that the precondition of the outer `#ifdef` block contradicts with the condition of the inner `#ifndef`.

Code Defects

Code defects are logic integrity violations (Section 2.2.2). *"Code"* thereby indicates that the contradictory formula grounds in the source code itself, and solely affects the *implementation space*. A *code defect* occurs due to contradictory usage of at least one KCONFIG feature in `#ifdef` blocks, whereas the contradiction is independent from the underlying variability model of the *configuration space*. Most *code defects* are caused by *double checks* of the same KCONFIG feature (Listing 3.1.3), resulting in a contradiction or tautology in the corresponding boolean formula.

```
#ifdef CONFIG_PPC_ADV_DEBUG_REGS
...
#ifdef CONFIG_PPC_ADV_DEBUG_REGS
            if (DBCR_ACTIVE_EVENTS(current->thread.dbcr0,
                      current->thread.dbcr1))
                regs->msr |= MSR_DE;
            else
                /* Make sure the IDM bit is off */
                current->thread.dbcr0 &= ~DBCR0_IDM;
#endif
        }

        _exception(SIGTRAP, regs, TRAP_TRACE, regs->nip);
    } else
        handle_debug(regs, debug_status);
}
#endif /* CONFIG_PPC_ADV_DEBUG_REGS */
```

Listing 3.1.3 – This source-code snippet of `arch/powerpc/kernel/traps.c` (Linux v2.6.34) shows a real *code defect*, where the inner `#ifdef` block is undead since it double checks the same KCONFIG feature than its enclosing `#ifdef` block. Such cases can also lead to dead blocks, depending on the CPP statement and on the structure of blocks (e.g, a following `#else` block).

To conclude. Variability related defects occur in the interaction of the two spaces. On one side, there is KCONFIG, the *configuration space*, where features and its dependencies are defined. On the other side, there is the source code, the *implementation space*, where the logical structure of C preprocessor macros adds additional constraints to the decision, if an `#ifdef` block can be selected or not. Such defects can be of a symbolic nature, so that references on absent KCONFIG features cause *referential integrity violations*. There are also *logic defects*, which describe defects that are caused by contradictory constraints of KCONFIG features. These contradictions can stem from the *configuration space* as well as from the *implementation space* space. Note that the UNDERTAKER tool reports every contradiction or tautology in a boolean formula as a *missing defect* as soon as it detects a *referential integrity violation*; regardless if the corresponding formula contains logic defects as well. However, I identify that almost 30 percent of reported defects are caused by CPP identifiers that can be related to debugging (i.e., "`#ifdef DEBUG`"). I consider such cases as false positives, and, as a consequence, I extend the UNDERTAKER tool with the following defect class:

No KCONFIG Defects

No KCONFIG *defects* cannot be directly related to variability related defects in the context of KCONFIG. *"No* KCONFIG" thereby indicates that at least one of the referenced literals in the contradictory formula is not related to KCONFIG and is thereby not prefixed with "`CONFIG_`". Such literals can be any defined CPP item outside the *configuration space*, such as debug flags or processor frequencies, which are defined in source or header files. As *No* KCONFIG *defects* are not related to KCONFIG, I exclude such from any of my data sets that are subject of analysis. Note that every defect with at least one item that is not related to KCONFIG will be classified as a *no* KCONFIG *defect*, regardless if there are *missing items* or *logic defects* as well.

Analysis of Defects

In total, I analyze 24 versions of the Linux kernel (v2.6.29 – v3.12). I trigger the UNDERTAKER tool on each version separately with the corresponding variability model from Section 3.1.1. This process takes 3 hours and 40 minutes in total, and generates 34 724 defect reports which consume 919 MB of memory space. Due to the new defect class, I can remove all *No* KCONFIG *defects* from the set of reported defects and can thereby reduce the amount of defects from *34 724* to *26 873*. Figure 3.2 compares the evolution of *symbolic*, and *logic integrity violations*, which are further split in *kconfig* and *code defects*. Throughout all analyzed Linux versions, there are much more (two to four times more) *missing defects* than *logic defects*. This is an interesting observation, since *missing defects* can be detected comparatively easy by cross-checking referenced KCONFIG features with the *implementation space*. I conclude, that most of such defects occur due to insufficient review processes, what enforces the need of a tool that avoids the integration of such defects.

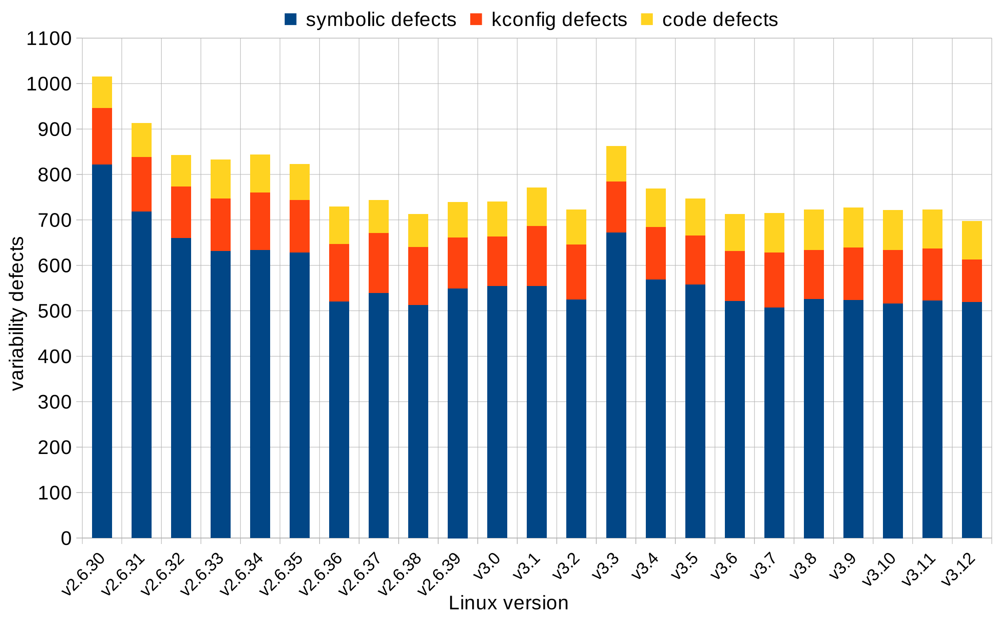


Figure 3.2 – Evolution of symbolic and logic integrity violations.

3.1.3 Tracking of Defects

In the aforementioned steps, I first extract variability models from the *configuration space* of Linux, and then generate defect reports with the UNDERTAKER tool. At this point, I need to develop a mechanism to track the reported defects over time; it allows the additional aggregation of data such as the *life span* of defects, and the *defect history* (e.g., changes to CPP macros and referenced KCONFIG features). Consequently, I need to find the correlation of `#ifdef` blocks in the set of defect reports and thereby answer the following question:

⇒ ***Which block in Linux version $\underline{A}$ corresponds to which block in version $\underline{B}$?***

I realize the required tracking functionality by using GIT to generate a DIFF between two versions for each defect affected source file. In a second step, I parse the DIFF file to update the blocks' line ranges; now there is an exact mapping of blocks between two Linux versions. To realize this process, I extended the UNDERTAKER with the functionality to display a list of all `#ifdef` blocks and the corresponding line ranges of a specified source file.

The described process maps the previously generated *26 873* defect reports to *4727* unique defects, whereas *923* are introduced and fixed in the analyzed period of time. The remaining exceed the analysis interval and, as a consequence, are ignored in further analysis steps as a mapping to the introducing or to the fixing commit, or both, is not possible. I implemented this process in a PYTHON script, which takes 17 minutes on average on a machine with an Intel Core i7 Quad-Core and 16 GB of RAM.

3.1.4 Mapping Defects to GIT Commits

The mapping from defects to GIT commits entails several challenges. If we want to find the introducing and fixing GIT commit of a variability defect, we need to understand the defect first. However, the analysis of a defect can be arbitrarily complex since the underlying boolean formula may contain hundreds of thousands of literals – the manual identification of the defect causing KCONFIG features is barely possible. In addition to that, there is no tool to *analyze* such defects; the UNDERTAKER simply detects defects, but does no further analysis of the defect's cause and effect.

Previous research shows that automated approaches to analyze *missing defects* caused by typos produce intolerably high rates of up to 73 percent of false positives [Nadi et al., 2013]. *Missing defects* caused by typos are the most trivial case as they are solely introduced and fixed by changes to CPP items in the source code itself. However, *missing defects* can generally stem from changes to the *configuration space* as well. *Logic defects* have not been subject of analysis at all, and, in addition to that, they constitute a finite limit of automated analysis approaches; it is generally impossible to automatically detect a single defect causing literal in a boolean formula. This issue exceeds the complexity of decision problems and further complicates automated mapping approaches.

Consequently, I decide to drive a manual analysis approach entailing the additional advantage of gaining expertise in the analysis of variability related defects. In the following, I shall describe my approach of manually analyzing variability defects, how to map them to GIT commits, and describe the scripts I developed during analysis.

Missing Defects

A *missing defect* is caused by at least one *referential integrity violation* in the precondition of the corresponding `#ifdef` block. In other words, some literals of the boolean defect formula are not defined in KCONFIG and are thereby absent in the underlying variability model. As a consequence, the analysis of *missing defects* can be reduced to searching strings. All we need to do, is to parse the boolean formula and then check if the referenced KCONFIG features are defined in the corresponding variability model; undefined features directly contribute to the defect and must be reported as *defect causing*.

At this point, we know which KCONFIG features are absent in the *configuration space* and thereby cause the *missing defect*. Now, we need to find the defect causing and defect fixing GIT commit, what can be simplified by making use of GIT internal functionality, mainly `git log`. With this function, we reduce the search space of GIT commits to those that (a) are in the given life span of the defect, and (b) that change the defect causing KCONFIG feature.

Kconfig Defects

Kconfig defects are comparatively hard to analyze since they ground on a contradiction or a tautology in a boolean formula with up to hundreds of thousands of literals. Therefore, a colleague and I extended the UNDERTAKER tool with the functionality to generate a minimally unsatisfiable subset (MUS) [Liffiton and Sakallah, 2005] of the contradictory boolean formula. The MUS only includes items that are part of the defect causing constraint, and thereby significantly reduces the potential search space from hundreds of thousands of literals to two to ten. The usage of this functionality is a fast and easy way to find the defect causing KCONFIG features for a given *Kconfig defect*.

Listing 3.1.4 depicts such a minimally unsatisfiable subset of a *Kconfig defect*, which we previously examined in Section 3.1.2, Listing 3.1.3. The displayed boolean formula of the dead `#ifndef` block contains information from the *configuration space* (KCONFIG features), as well as information from the *implementation space*, namely the CPP structure of the affected source file (`./kernel/rcupdate.c`). `B{3,4}` denote the third and the fourth `#ifdef` block in the file. The formula thereby gives us two important informations: (a) the defect is caused by the conflicting preconditions of the CPP block `B3` and block `B4`, and (b) the defect is further caused by logical conflicts among the displayed KCONFIG features.

Once you generated a MUS, I recommend to check the CPP structure of the affected `#ifdef` blocks first and then to further examine the constraints of the affected KCONFIG features.

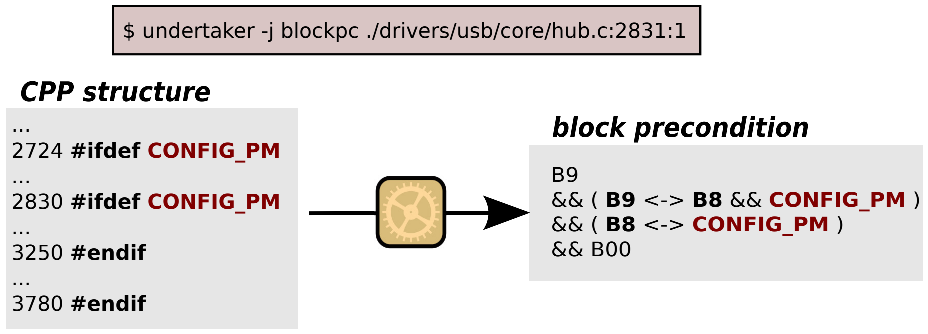
```
(B3) ^ (!CONFIG_PREEMPT) ^ (!B4) ^ (B4) ^
(!B3 v CONFIG_DEBUG_OBJECTS_RCU_HEAD) ^ (CONFIG_PREEMPT) ^
(!CONFIG_DEBUG_OBJECTS_RCU_HEAD)
```

Listing 3.1.4 – The displayed boolean formula of the dead `#ifndef` block (see Listing 3.1.3) contains information from the *configuration space* (KCONFIG features), as well as information from the *implementation space*, namely the CPP structure of the affected source file (`./kernel/rcupdate.c`); `B{3,4}` denote the third and the fourth `#ifdef` block in the file. The formula thereby gives us two important informations: (a) the defect is caused by the conflicting preconditions of the CPP block `B3` and block `B4`, and (b) the defect is further caused by logical conflicts between the display KCONFIG features.

The mapping to the defect causing and defect fixing GIT commits is orthogonal to *missing defects*, whereas we need to extend the search space to the KCONFIG definitions of affected KCONFIG features; a *Kconfig defect* can also be caused and fixed by changes to the dependencies of a feature.

Code Defects

Code defects are caused by contradictory CPP structures such as the double check of KCONFIG features. The detection of *code defects* is independent from the underlying variability model as only the information from the *implementation space* is mandatory. Consequently, such defects are comparatively easy and fast to analyze, since we only have to examine the CPP structure of the defect affected `#ifdef` block. In this context, I recommend to use a built-in functionality of the UNDERTAKER which displays the CPP precondition of a specified block. The precondition instantly indicates which KCONFIG features directly contribute to the contradictory formula. Figure 3.3 shows the precondition output of the UNDERTAKER for a *code defect* in `./drivers/usb/core/hub.c` in Linux version v3.10. The affected block `B9` is enclosed by another block `B8` which references the same KCONFIG item, making block `B9` *undead*.



```
$ undertaker -j blockpc ./drivers/usb/core/hub.c:2831:1
```

```
...
2724 #ifdef CONFIG_PM
...
2830 #ifdef CONFIG_PM
...
3250 #endif
...
3780 #endif
```

```
B9
&& ( B9 <-> B8 && CONFIG_PM )
&& ( B8 <-> CONFIG_PM )
&& B00
```

Figure 3.3 – UNDERTAKER: precondition of a code defect in ./drivers/usb/core/hub.c, Linux v3.10. The affected block `B9` is enclosed by another block `B8` which references the same KCONFIG item, making block `B9` *undead*.

The mapping to the defect causing and defect fixing GIT commits is orthogonal to *missing defects*, whereas *code defects* can only be caused by changes to the *implementation space*. Thereby, we can reduce the search space to the defect affected source file, and take the KCONFIG features of the block's precondition (see Figure 3.3) as search identifiers for `Git log`.

3.1.5 Work Flow

The entire process of ***feature consistency analysis in Linux*** is illustrated in Figure 3.4. First, I extract variability models ❶ from the *configuration space* of Linux, namely KCONFIG. These variability models cover the boolean feature constraints of KCONFIG, which I use in a second step to detect variability related defects ❷ by making use of the UNDERTAKER tool. At this point, I use self-developed PYTHON scripts to find the mapping of defects between different Linux versions. Thereby, the mapping allows us to track variability related defects ❸ over time in the Linux GIT repository. Finally, I manually identify the causes and effects of the tracked defects and map them to the introducing and fixing GIT commits ❹.

During the manual analysis of defects in the mapping process (❹), I experienced the need of tools that help to analyze the causes and effects of variability defects. The results of my empirical case study are needed not only to have a better understanding of such defects, but also to identify the most common mistakes in Linux development. I use this data for the implementation of my tool, UNDERTAKER–CHECKPATCH, to further analyze the causes and effects of detected defects. I present the results of the analysis in the following sections separately for each defect class.

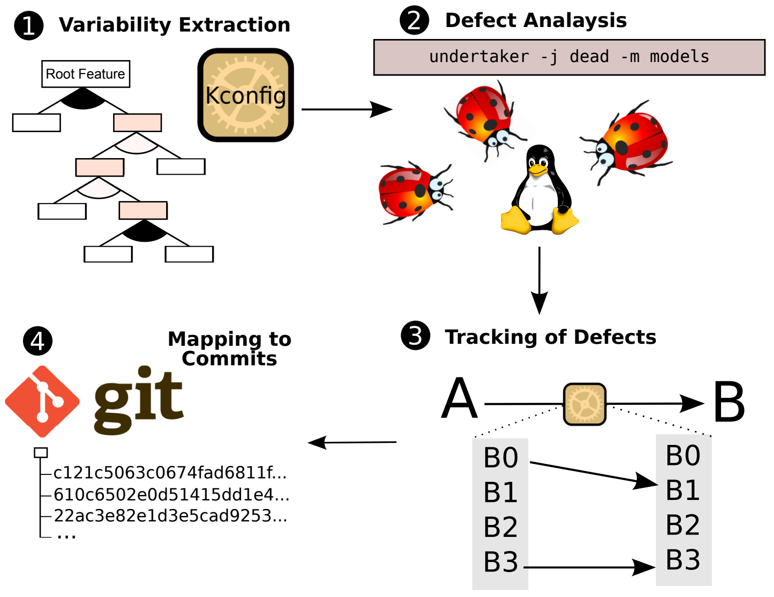


Figure 3.4 – Tool chain of feature consistency analysis in Linux. After the extraction of variability models (1) from Kconfig, I instrument the UNDERTAKER tool to report variability related defects (2) in the source code of the Linux kernel. In a following step, I track the previously generated defects (3) throughout all analyzed versions of Linux (v2.6.29 – v3.12). Finally, I manually identify the causes of tracked defects and map them to the introducing and fixing GIT commits (4).

3.2 Symbolic Integrity Violations

Symbolic integrity violations are known to show a higher distribution than logic defects [Nadi et al., 2013, Tartler et al., 2011]. This observation aligns with my data of tracked defects between Linux v2.6.29 and Linux version v3.12. *768 of 923 tracked defects* in this period of time are reported as missing and constitute a total share of *83.21 percent* of defects. The majority of symbolic defects occurs in the *hardware abstraction layer* (63.41 percent) and the *drivers subsystem* (31.25 percent). The remaining 5.31 percent of symbolic defects occur in the subsystems *file system (fs), kernel, memory management (mm), network (net)* and the *sound subsystem* (sound). In total, I manually analyzed 277 of the 768 tracked symbolic defects, what I consider to be representative enough to (a) cover all possible defect causes and fixes as well as to (b) generate meaningful data that can be used and referenced in future research.

Symbolic integrity violations are described to stem from broken references in the source code [Nadi et al., 2013, Tartler et al., 2011], such as an `#ifdef` block, which references an item that is not defined in KCONFIG and thereby leads to a defect. However, I discover *another cause of symbolic defects* that primarily affects the configuration space, namely KCONFIG itself. In those cases, *missing defects* are caused by broken dependencies in KCONFIG, where referenced items from `#ifdef` blocks *depend on* at least one feature that is *not defined* in KCONFIG and thereby cause a violation of the referential integrity. Listing 3.2.1 illustrates such a violation, where one KCONFIG feature depends on another, which is not defined in KCONFIG and is thereby absent in the variability model.

```
config SERIAL_8250_RM9K
   bool "Support for MIPS RM9xxx integrated serial port"
   depends on SERIAL_8250 != n && SERIAL_RM9000
   select SERIAL_8250_SHARE_IRQ
   help
     Selecting this option will add support for the integrated serial
     port hardware found on MIPS RM9122 and similar processors.
     If unsure, say N.
```

Listing 3.2.1 – KCONFIG snippet of `drivers/tty/serial/8250/Kconfig` (Linux v3.8). The displayed KCONFIG feature depends on `SERIAL_RM9000`, which has been removed by a GIT commit. As the referenced feature is not defined in the *configuration space*, the commit caused a referential integrity violation. Note that all references on `SERIAL_8250_RM9K` will result in a *missing defect*.

None of the previous research papers [Nadi et al., 2013, Sincero et al., 2010, Tartler et al., 2011, 2012] discusses the fact that missing defects can be caused by broken dependencies. In my data set of analyzed symbolic violations, *four defects* are directly caused by such dead references in dependencies of KCONFIG features. Although the number of broken dependencies in KCONFIG is minute, the potential consequences of such bugs can cause well manifested errors in the system, as to be seen in the code example of Listing 3.2.2. The referenced KCONFIG feature `CONFIG_HIGHMEM_START_BOOL` in the C preprocessor macro depends on `CONFIG_HIGHMEM` which is not defined for the `Microblaze` architecture. As undefined items evaluate *false*, the `#ifdef` block is dead and the `#else` block is undead. The displayed C source code is a conditional assignment to the base address of the memory mapped IO address space of the `Microblaze` kernel. As a matter of fact, the intentionally conditional assignment is unconditional since the `#else` block is part of every compilation unit. This defect clearly violates the integrity of the operating system since it can manifest in undefined system behaviors as well as in a complete failure of the system (e.g., `NULL` pointer dereferences). An even more surprising fact is that developers were not aware of this defect at all as it was more or less "accidentally" repaired by defining a fixed base address. Thereby the conditional assignment has been made superfluous so that both CPP macros have been removed. Note that this defect was present for two Linux versions (v3.2, v3.3).

```
#ifdef CONFIG_HIGHMEM_START_BOOL
    ioremap_base = CONFIG_HIGHMEM_START;
#else
    ioremap_base = 0xfe000000UL;    /* for now, could be 0xfffff000 */
#endif /* CONFIG_HIGHMEM_START_BOOL *
```

Listing 3.2.2 – Missing defect in **/arch/microblaze/mm/init.c** (Linux v3.2) caused by a dependency from the referenced item `CONFIG_HIGHMEM_START_BOOL` on `CONFIG_HIGHMEM` which is not the defined for the Microblaze architecture.

Although only four of the analyzed defects cover exactly this case, I examine at least eleven cases where missing items are still referenced in other files, such as source files, header files, MAKE or KBUILD, or even in KCONFIG itself. I will show in Chapter 5 that KCONFIG patches, that do not properly propagate changes to other files, are a common cause of *symbolic integrity violations*. The remaining defects are caused by broken references from C preprocessor macros. In the following sections I describe my insights into the 277 analyzed missing defects and how they are introduced and fixed by GIT commits.

3.2.1 Causes of Symbolic Integrity Violations

How are missing defects introduced?

The introduction of missing defects to the source of Linux can stem from changes to source files as well as from changes to KCONFIG files. Nadi et al. [2013] identify two cases of a patch introducing a *referential integrity violation*:

❶ A patch **removes** a feature in KCONFIG

❷ A patch **renames** a feature in KCONFIG

My results reveal a different image of how *missing defects* can be introduced to the Linux kernel, such that they can be caused not only by changes to the *configuration space*, but also by changes to the *implementation space*. Table 3.1 includes the identified cases of patches that introduce *symbolic integrity violations*. The data shows that such defects can stem from changes to KCONFIG as well as from changes to source files, whereas the majority is caused by renaming a KCONFIG related item in a C preprocessor macro: 133 of 277 (48 percent) missing defects. 50 defects are caused by patches adding source files referencing items that are absent in KCONFIG. A total amount of 37 defects is caused by patches touching KCONFIG files of which four defects are caused by *adding* a KCONFIG feature with broken dependencies. Furthermore, I identify 16 defects as *intentional exclusion* of source code where authors "added the symbol to hide the dependent from compilers" [9] as parts of the code were held back until another driver has been integrated two Linux versions after.

Cause	**# Defects**
Rename CPP item	133
Add file	50
Add `#ifdef` block	41
Remove KCONFIG feature	22
Intentional exclusion	16
Rename KCONFIG feature	11
Add KCONFIG feature	4

Table 3.1 – Identified causes of *symbolic integrity violations* between Linux v2.6.29 and v3.12

[9]Linux GIT commit: 7b4050381127ae11fcfc74a106d715a5fbbf888a

Figure 3.5 illustrates the set of introduced *symbolics defects* and correlates them to the total progression of symbolic defects. The graph includes one anomaly in the progression line at version v3.3 which includes 160 new *missing defects*. This peak derives from a single patch changing C preprocessor macros in 26 source files in the `blackfin` architecture. Since the renamed item is not defined in KCONFIG, the patch[10] introduces 119 *referential integrity violations* to the source code by renaming CPP items to a feature which is not defined in KCONFIG. I am equally surprised that this patch was not corrected before integration or before the merge window has been closed in the stable Linux GIT, as well as that the defects could have been avoided by simply grepping the `blackfin` KCONFIG file. Developers and maintainers regularly fail in checking GIT commits for such fairly trivial defects, what I explain with (a) that developers may not be aware of this problem, and (b) that maintainers do not have enough time to check GIT commits for such issues. This case enforces the need of a tool to check configuration-critical patches before submission.

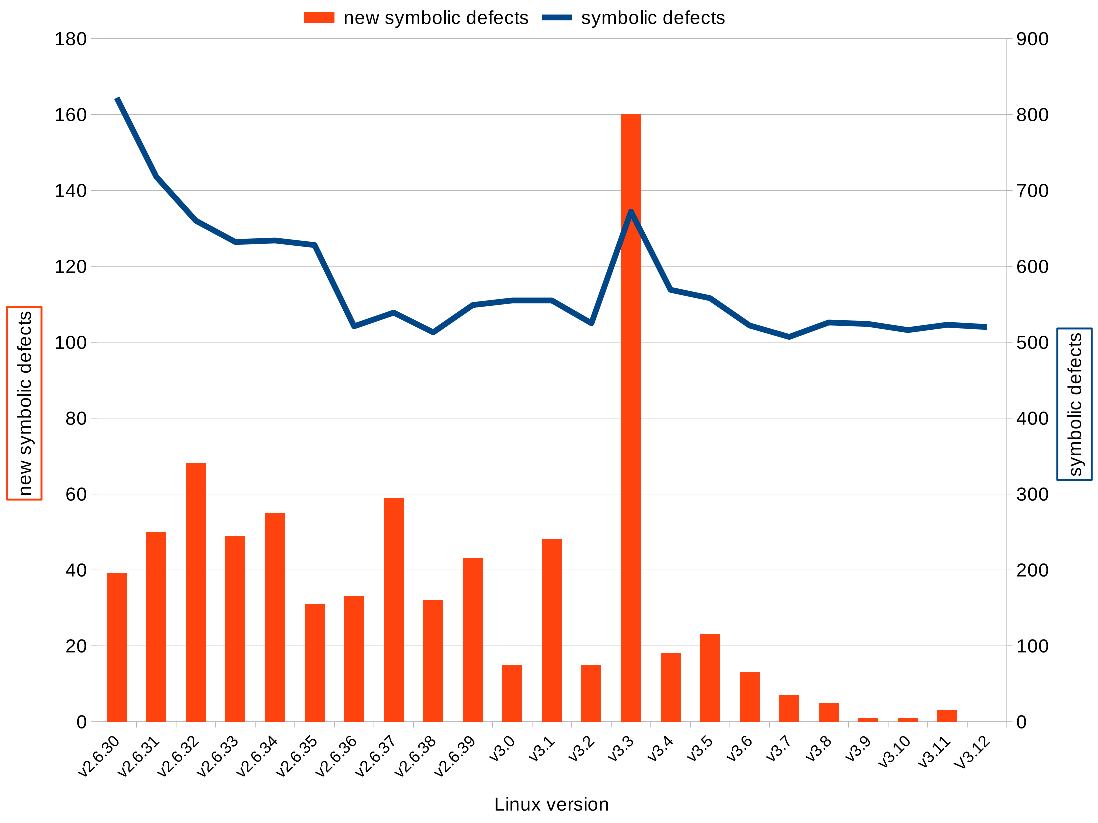


Figure 3.5 – Evolution of introduced symbolic defects in the Linux kernel correlated to the total progression of symbolic defects between v2.6.29 and v3.12.

[10] Linux GIT commit: 7d157fb02bc3f4dc74e6830725864ba501d92da7

3.2.2 Fixes of Symbolic Integrity Violations

How are missing defects repaired?

The fix of a missing defect is orthogonal to the defect's cause – you can fix it with changes to the *implementation space* as well with changes to the *configuration space*. Nadi et al. [2013] identify four cases of a patch fixing a *referential integrity violation*:

❶ A patch adds a feature to KCONFIG

❷ A patch renames a feature in KCONFIG

❸ A patch removes a CPP condition

❹ A patch renames a feature in a CPP macro

I agree with this observation besides the case that a defect can also be fixed by removing or renaming a dead dependency in KCONFIG. Table 3.2 shows that the majority of analyzed missing defects is repaired by renaming a KCONFIG feature such as the previously mentioned defects in `arch/blackfin`. Furthermore, 52 defects are repaired by the removal of the entire file. In four cases, a patch removes blocks as well as a feature in KCONFIG, as the entire functionality was removed from the source. Fourteen defects are fixed by removing the entire `#ifdef` block, whereas the code of the `#else` block is preserved. I conclude that the reparation of such defects requires certain domain knowledge of the source code as well as of the feature model in Linux. In many cases it is the developer's decision how to fix a bug. As consequence, it is generally impossible to automatically fix such defects. However, a tool that helps to analyze and understand a defect's cause can help developers to find a correct solution.

Fix	**# Defects**
Rename KCONFIG feature	127
Remove file	52
Rename CPP item	41
Remove `#ifdef` block	15
Remove if, keep else	14
Add KCONFIG feature	7
Remove block and KCONFIG feature	4

Table 3.2 – Identified fixes of *symbolic integrity violations* between Linux v2.6.29 and v3.12

Figure 3.6 illustrates the set of fixed *symbolic defects* and correlates them to the total progression of symbolic defects. I remove such defects from the set of tracked defects that exceed the left and right border as no mapping to the introducing and fixing commits is possible. This leads to a chart similar to normal distributions. Besides Linux v3.4, an average of 27.5 symbolic defects is fixed with each new Linux version. The peak of fixed *symbolic integrity violations* in Linux v3.4 can be attributed to the aforementioned issue in the `blackfin` architecture. A total of 119 *missing defects* in Linux v3.3 is caused by a patch that renames KCONFIG features in C preprocessor macros. These defects are repaired with Linux v3.4 by renaming a feature in KCONFIG itself to the previously changed CPP item. *Missing defects* remain unfixed for averagely 3.65 Linux versions in the analyzed period of time.

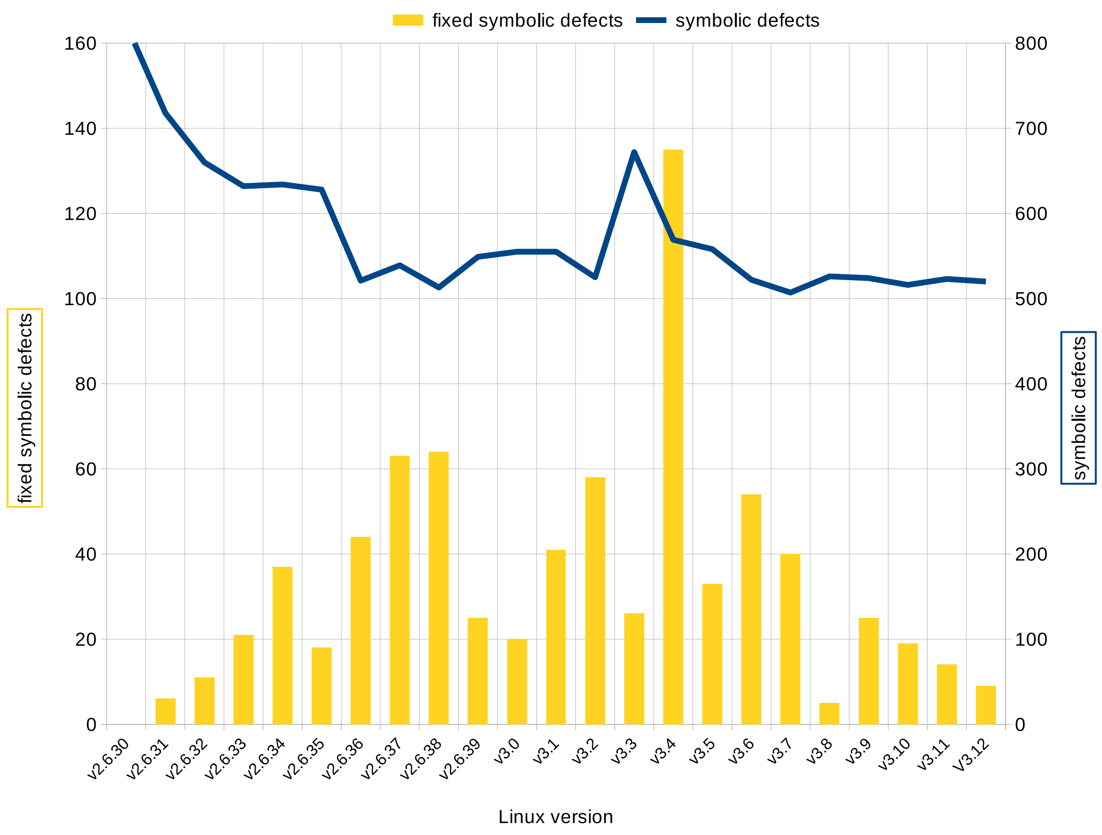


Figure 3.6 – Evolution of fixed symbolic defects in the Linux kernel correlated to the total progression of symbolic defects between v2.6.29 and v3.12.

3.2.3 Distribution to Subsystems

Thirteen years ago, "Chou et al. [2001] published a study of faults found by applying a static analyzer to Linux versions 1.0 through 2.4.1. A major result of their work was that the drivers directory contained up to 7 times more of certain kinds of faults than other directories. This result inspired numerous efforts on improving the reliability of driver code." [Palix et al., 2014] Until today, this observation has been applied to variability related defects as well[11]. However, Palix et al. [2014] discover that faults in the `drivers` subsystem of the Linux kernel nowadays are below other subsystems, such as `arch`.

My analysis reveals a similar observation. Table 3.3 shows this observation and contains the amount of *missing defects* per subsystem, the amount of `#ifdef` blocks, the amount of defects per `#ifdef` block, the source lines of code, and the amount of defects per one thousand source lines of code. In general, `arch` shows the highest amount of *missing defects* and also the highest rate of defects per `#ifdef` block and per one thousand source lines of code (SLOC). We can see that `arch` contains 17 times more *missing defects* per one thousand SLOC (0.4149) than `drivers` (0.0240). I assume that Linux developers do not use proper tools to avoid the introduction of variability defects. A tool to identify and analyze such defects in GIT commits could thereby contribute to the overall quality of the Linux source code and support developers at work.

Subsystem	Defects	`#ifdef` Blocks	Defect/Block	SLOC	Defect/1k SLOC
arch	487	20 613	0.0236	1 173 609	0.4149
arch/arm	289	4172	0.0693	346 137	0.8349
arch/blackfin	110	2436	0.0452	49 603	2.2176
drivers	240	32 839	0.0073	5 373 230	0.0240
fs	24	3220	0.0075	691 789	0.0346
kernel	6	1688	0.0035	123 748	0.0484
mm	3	806	0.0037	56 446	0.0531
net	1	3714	0.0002	512 910	0.0019
sound	7	3690	0.0018	503 548	0.0139

Table 3.3 – The table contains the amount of *missing defects* per subsystem, the amount of `#ifdef` blocks, the amount of defects per `#ifdef` block, the source lines of code, and the amount of defects per one thousand source lines of code. In general, `arch` shows the highest amount of *missing defects* and also the highest rate of defects per `#ifdef` block and per one thousand SLOC. We can see that `arch` contains 17 times more *missing defects* per one thousand SLOC (0.4149) than `drivers` (0.0240).

[11]Personal correspondence with Tartler et al. [2010].

3.3 Logic Integrity Violations

Logic integrity violations stem from contradictions and tautologies of a block's precondition or contradictory constraints in KCONFIG itself. As a matter of fact, logic defects are (a) harder to detect and (b) entail a more complex analysis than *symbolic defects*. If you want to detect a missing item, it is sufficient enough to search literals in the boolean formula that are not defined in KCONFIG and, as a consequence, are absent in the extracted variability model. In contrary, *logic integrity violations* require a *semantic analysis* as the feature constraints of each referenced item need to be taken into account to detect potential contradictions (dead blocks) and tautologies (undead blocks). In general, there are two classifications of logic defects, *kconfig defects* and *code defects* which I describe separately in the following two sections.

3.3.1 Kconfig Defects

Kconfig defects are caused by contradictory boolean formulas, which stem from contradictory constraints in KCONFIG as well as from conflicting preconditions of `#ifdef` blocks. As a consequence, *kconfig defects* are hard to detect since the final boolean formulas may contain hundreds of thousands of literals which deludes human understanding. Furthermore, the analysis of this logic defect class requires domain specific knowledge and understanding in the feature model of the Linux kernel.

3.3.1.1 Causes of Kconfig Defects

How are kconfig defects introduced?

In general, there are two cases of *kconfig defect* that we need to discuss separately. First, a *kconfig defect* can be caused by conflicting preconditions of `#ifdef` blocks in the source code. This case can be explained best, if we have a look at a real defect. Listing 3.3.1 shows the precondition of the CPP block 19 of `/arch/x86/kernel/apic/apic.c` in Linux v3.2. Block 19 is an `#else` branch of block 18, which depends on `CONFIG_INITR_REMAP`. Block 18 is further enclosed by block 17 that references the KCONFIG feature `X86_X2APIC`. Block 18 is *undead*, block 19 is accordingly *dead*. The KCONFIG feature `X86_X2APIC` depends on `INITR_REMAP` so that `INITR_REMAP` is always true in the enclosing block 17, and thereby causes a tautology in the precondition of block 18 (undead) and a contradiction in the precondition of block 19 (dead). These defects occur in the interaction of the logic CPP structure with constraints of KCONFIG. Hence, I recommend to check the CPP structure of the source file first, before analyzing the corresponding minimally unsatisfiable subset. I analyzed 53 of the 101 tracked *kconfig defects*; 23 of them are attributed to a contradictory usage of KCONFIG items.

```
B19
&& ( B19 <-> B17 && ( ! (B18) ) )
&& ( B18 <-> B17 && CONFIG_INTR_REMAP )
&& ( B17 <-> CONFIG_X86_X2APIC )
&& B00
```

Listing 3.3.1 – *Kconfig defect* in `./arch/x86/kernel/apic/apic.c` (Linux v2.6.30). Block 19 is an `#else` branch of block 18, which depends on `CONFIG_INITR_REMAP`. Block 18 is further enclosed by block 17 that references the KCONFIG feature `X86_X2APIC`. Block 18 is *undead*, block 19 is accordingly *dead*. The KCONFIG feature `X86_X2APIC` depends on `INITR_REMAP` so that `INITR_REMAP` is always true in the enclosing block 17, and thereby causes a tautology in the precondition of block 18 (undead) and a contradiction in the precondition of block 19 (dead).

The second identified cause of *kconfig defects* is related to items that are *always on* (true) causing tautologies or contradictions, depending on their occurrence in the corresponding formula. You can examine such items most easily by checking the extracted variability models, which mark such features as "`ALWAYS_ON`". In principle, these features relate to functionalities that are mandatory to the system, for instance `MMU` (memory management unit) or `SMP` (symmetric multiprocessing). 14 of the 53 analyzed *kconfig defects* are attributed to references on always on items.

The remaining *kconfig defects* are directly caused by conflicts in the feature model that cannot be resolved and thereby produce dead and undead blocks that reference affected KCONFIG features. In general, this issue affects any constraint-defining statement of the KCONFIG language. The following example was present for *15 Linux versions* (v2.6.37 – v3.11) in `IA64` and directly affects the architecture's root feature `CONFIG_IA64`. The issue can be described best, when we a look at the feature's definition in KCONFIG, illustrated in Listing 3.3.2. `IA64` conditionally selects the features `PCI, ACPI and PM` and unconditionally selects `ARCH_SUPPORTS_MSI` which, due to some constraints, sets `IA64_HP_SIM` to false. Consequently, all features are selected (evaluate true) since the condition of the if statements are met. As `IA64` is the root feature of the `IA64` architecture (and is thereby always selected), the listed features consistently evaluate true (always on). As a consequence, the intentionally conditional KCONFIG feature constraints turn to constants.

```
config IA64
   bool
   select PCI~  if (!IA64_HP_SIM)
   select ACPI~ if (!IA64_HP_SIM)
   select PM    if (!IA64_HP_SIM)
   select ARCH_SUPPORTS_MSI
```

Listing 3.3.2 – An excerpt of the `IA64` KCONFIG file (v2.6.37 – v3.11). `CONFIG_IA64` conditionally selects the features `PCI, ACPI and PM` and unconditionally selects `ARCH_SUPPORTS_MSI` which, due to some constraints, sets `IA64_HP_SIM` to false. Consequently, all features are selected (evaluate true) since the condition of the if statements are met. As `IA64` is the root feature of the `IA64` architecture (and is thereby always selected), the listed features consistently evaluate true (always on). As a consequence, the intentionally conditional KCONFIG feature constraints turn to constants.

Table 3.4 shows the identified cases of how *kconfig defects* are introduced. A defect's cause can stem from various changes to the *implementation* and *configuration space*. Consequently, a *kconfig defect* can be caused by adding a new block (14) or renaming a referenced CPP item (7), as well as by changes to KCONFIG such as a feature's type (10), a select (1) or a dependency (1).

Much to my surprise, *kconfig defects* can also ground in removing a KCONFIG feature (and all references in the KCONFIG space). 10 defects have been caused by removing a KCONFIG feature that was selected by two others which were located in the same choice. A choice is used to *exclusively select* contained features and thereby forbids to select more than one feature of the same choice; dependencies on features in the same choice are generally contradictory. However, there is a special way in KCONFIG to break its own rules. Choice features can depend on each other if both select a feature *outside* the choice. Consider two KCONFIG features A and B. Both are defined in the same choice and select another feature C, which is defined outside the choice. This special case allows dependencies from A on B, and vice versa. The exclusive select turns out to be non-exclusive. This was the case for 10 of the analyzed defects which are caused by the removal of the non-choice feature and corresponding selects from within the choice. Figure 3.7 illustrates the overall progress of *kconfig defects* in the analyzed period of time.

Cause	# Defects
Add `#ifdef` block	14
Change KCONFIG type	10
Remove KCONFIG feature	10
Add file	10
Rename CPP item	7
Add KCONFIG select	1
Change KCONFIG dependency	1

Table 3.4 – Identified causes of ***kconfig defects*** between Linux v2.6.29 and v3.12

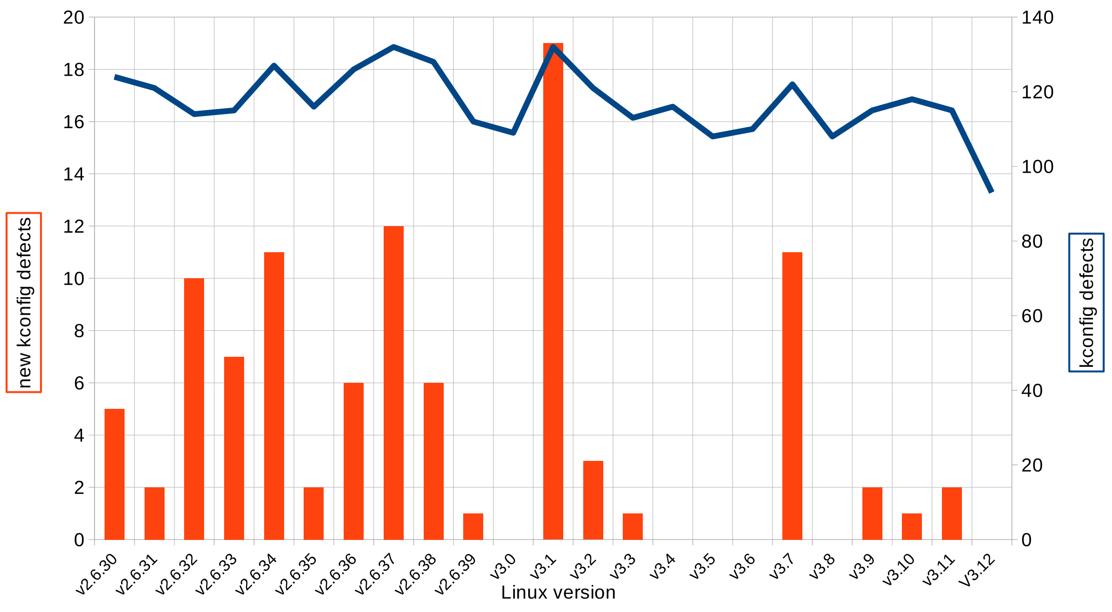


Figure 3.7 – Evolution of introduced kconfig defects in the Linux kernel correlated to the total progression of kconfig defects between v2.6.29 and v3.12.

3.3.1.2 Fixes of Kconfig Defects

How are kconfig defects repaired?

Kconfig defects can stem from various changes to the logic C preprocessor structure of the source code as well as from changes to the *configuration space*. A block's precondition may reference an always on item, it may conflict with the precondition of the enclosing block, or contain a contradictory formula due to constraints in KCONFIG. This variety of changes also applies to the issue of fixing kconfig defects. In stark contrast to *missing defects*, some *kconfig defects* can only be fixed either in KCONFIG or in the source code, depending on a defect's cause. A contradictory dependency, for instance, can only be fixed by changes to KCONFIG itself, whereas a conflict in two block preconditions may be fixed by renaming one of the referenced KCONFIG features in a source file, or by resolving the contradiction in the constraints of KCONFIG. However, I identify that most fixes (see Table 3.5) happen in the source code by removing entire `#ifdef` blocks (14), removing C preprocessor macros (13) or by renaming the referenced CPP item (5). Four conflicts are resolved by removing the `#ifdef` block whereas the code of the `#else` block remains untouched. The problem with dependencies within the same choice (Figure 3.3.1.1) has been fixed by changing the choice to a KCONFIG menu, which allows such dependencies as the included options can be non-exclusively selected. Figure 3.8 on page 41 shows the overall progression line correlated to the identified fixes in the analyzed period of time. The tracked *kconfig defects* have an *average life span of 3.31*.

Cause	**# Defects**
Remove `#ifdef` block	14
Remove CPP macro	13
Change KCONFIG dependency	6
Rename CPP item	5
Change KCONFIG choice to menu	4
Remove if, keep else	4
Re-factor KCONFIG file	2
Remove KCONFIG select	2
Remove file	2

Table 3.5 – Identified fixes of *kconfig defects* between Linux v2.6.29 and v3.12

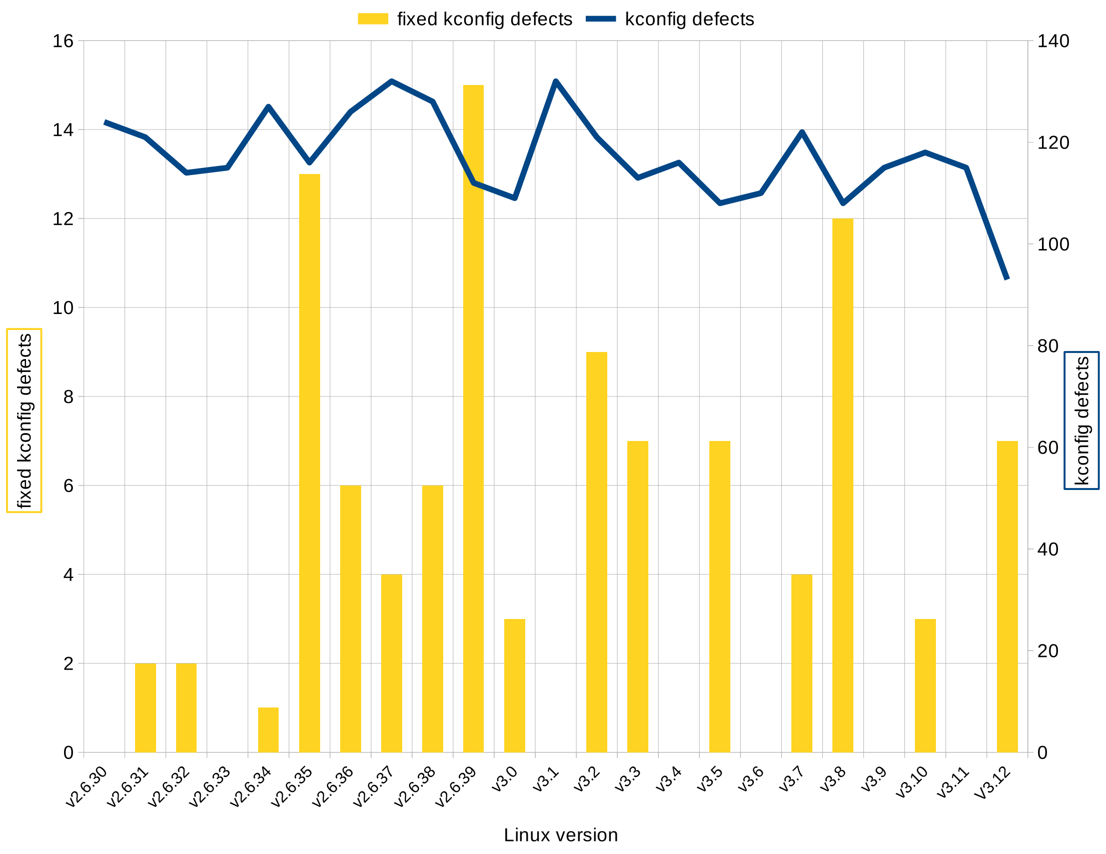


Figure 3.8 – Evolution of fixed *kconfig defects* in the Linux kernel correlated to the total progression of *kconfig defects* between v2.6.29 and v3.12.

3.3.2 Code Defects

Code defects stem from contradictions or tautologies in the source code itself, for instance, from double checks in successive C preprocessor macros, or from references on previously defined CPP items. As a consequence, the analysis of such defects does not require any variability model as they solely affect the *implementation space*. I analyzed all 54 tracked *code defects* with the following results.

3.3.2.1 Causes of Code Defects

How are code defects introduced?

In principle, you can analyze *code defects* in the same way as *kconfig defects*, namely by checking the precondition of the defect affected block first. The block's precondition is a reliable index if the defect is caused by double checked items or not. The second identified cause of *code defects* is previously defined or undefined CPP items in the source code. Such defines are overwriting the user-selected value from KCONFIG or even define a new “CONFIG_” prefixed item. Both cases are breaking the integrity of the *configuration space* of the Linux kernel as KCONFIG features should only and only be defined and evaluated in KCONFIG itself. However, these violations regularly occur in development; I attribute 37 of 54 analyzed code defects to this case. However, such defects are the consequence of insufficient reviews, as all the identified defines are leftovers of debugging and testing processes of the affected module. This fact enforces the need to check GIT commits before submission. Table 3.6 shows the identified cases of how code defects are introduced to the Linux kernel.

Cause	# Defects
Add file	32
Rename CPP item	12
Add block	6
Add `#undef`	4

Table 3.6 – Identified causes of *code defects* between Linux v2.6.29 and v3.12

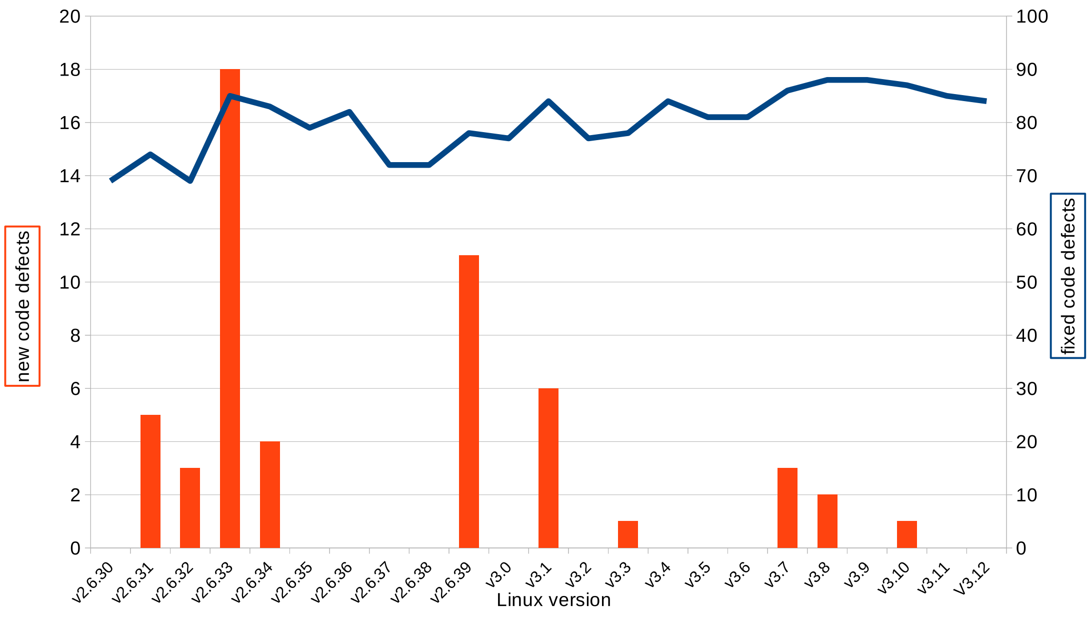


Figure 3.9 – Evolution of introduced *code defects* in the Linux kernel correlated to the total progression of code defects between v2.6.29 and v3.12.

3.3.2.2 Fixes of Code Defects

How are code defects repaired?

68.5 percent of *code defects* are caused by defining or undefining referenced KCONFIG features in source files. Such defects can only be fixed by removing the corresponding defines and undefines from the source code, and optional changes to the CPP macros and KCONFIG files. The second cause of *code defects* are double checks of KCONFIG features in successive CPP macros. You can fix such bugs in various ways, as the unconditional blocks can be made conditional by (a) renaming a referenced KCONFIG item or (b) by changing the logic structure of `#ifdef` blocks. Consequently, automated fixes of such defects are barely possible since a deeper understanding of a defect's context and of the source code is required. The identified cases of how code defects are fixed are shown in Table 3.5. Most of the defects are fixed by removing the file from the Linux source (31). The remaining fixes happen in the C preprocessor macros, for instance by renaming referenced items (14) or removing superfluous macros (7). The overall evolution of fixed code defects is illustrated in Table 3.7. The tracked and analyzed code defects have an *average life span of 7.59*. Although *code defects* are easier to detect than *missing defects* (life span of 3.65) and *kconfig defects* (life span of 3.31) they remain unfixed for a longer time. 41 of the 54 analyzed *code defects* occur in the `drivers` subsystem, so that I assume that most of the affected files are less important, and therefore less maintained than others, what would align with the fact that most defects are fixed by removing the entire file.

Fix	**# Defects**
Remove file	31
Rename CPP item	14
Remove CPP macro	7
Remove `#undef`	4
Remove `#ifdef` block	3
Remove `#def`	2

Table 3.7 – Identified fixes of ***code defects*** between Linux v2.6.29 and v3.12

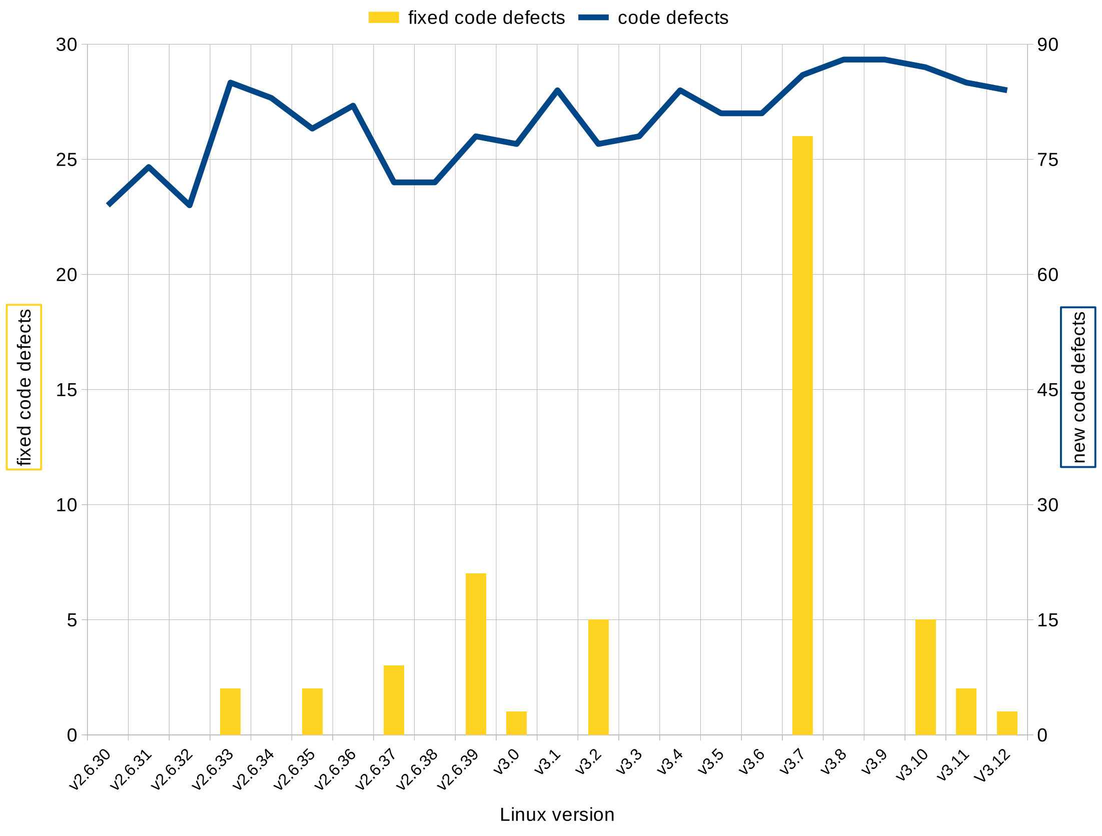


Figure 3.10 – Evolution of fixed code defects in the Linux kernel correlated to the total progression of code defects between v2.6.29 and v3.12.

3.4 Summary

Variability related defects present a delicate research field. The analysis of such defects requires a thorough understanding of the source code and the feature model of the Linux kernel and is barely possible without proper tool support. In my work, I make use of previous research work, namely the UNDERTAKER tools, to extract variability models from KCONFIG and to reveal variability defects in the source code. In total, I analyzed 24 versions of the Linux kernel which contain 34 724 variability defects. In a following step, I track the defects over all analyzed Linux versions to filter which defects correspond between different versions. Thereby, the previously generated 34 724 defects are reduced to 4727 *unique* defects of which 923 are introduced and fixed in the analyzed time interval (Linux v2.6.29 – v.3.12).

Missing defects constitute the biggest share of tracked defects. 768 of 923 defects (82.4 percent) are *symbolic integrity violations*, of which I manually analyzed 277 defects. In comparison to previous research [Nadi et al., 2013], I present a very different image of symbolic defects, as such may also affect KCONFIG itself with significant impact on both, the *configuration space* and the *implementation space*. In my results, symbolic defects show an average life span of 3.65, whereas previous research claims that such defects remain present for 6 Linux versions. I can prove that this difference is due to (a) inaccuracies in the approach of Nadi et al. [2013], and (b) due to the inclusion of *false positives* in previous research studies [Dietrich et al., 2012, Nadi et al., 2013, Tartler et al., 2011, 2012]. Hence, I introduced a new defect class to the UNDERTAKER tool, to filter such defects that are not related to KCONFIG (no KCONFIG defects). However, previous studies include these false defects and thereby distort *all* presented results.

Furthermore, my work constitutes the first empirical case study of logic defects. I analyzed 53 of 101 tracked *kconfig defects* and all 54 *code defects*. In this context, I co-developed a functionality of the UNDERTAKER tool, which is now able to generate a minimally unsatisfiable subset to simplify the analysis of contradictory boolean formulas of variability defects.

The final step of my analysis is the manual mapping of analyzed defects to GIT commits, in order to generate meaningful data of how each defect class is introduced and repaired in daily kernel development. Nadi et al. [2013] present an automated approach to find GIT commits that remove or rename a KCONFIG feature without propagating the change to the source code. However, this approach results in a high rate of up to 73 percent of false positives. My approach differs in various ways. First, Nadi et al. [2013] do not describe that *missing defects* can also be caused by changes to *source files*. Second, by using the HERODOTOS tool [Palix et al., 2009], they are not able to track variability related defects. HERODOTOS bases on COCCINELLE[12] to detect bugs in a semantic context [Padioleau et al., 2006], which does not cover the *configuration space* (KCONFIG). Thereby Nadi et al. [2013] are able to check how long a certain code pattern remained in the Linux source, but are not able to cross-check changes with the underlying variability models. By the design of my approach, I avoid such problems as (a) I use a self developed script to track defects over time, and (b) I manually map the *384* analyzed defects to the introducing and fixing GIT commits. During the manual analysis, I gained in depth knowledge of variability defects, how they occur and how they can be repaired. By doing so, I developed several scripts that help analyzing a defects. In the following subsection, I shall summarize all these insights which influenced the final tool, as described in Chapter 4.

[12] `http://coccinelle.lip6.fr/`

3.4.1 Lessons Learned

All the data and insights of this empirical analysis of variability related defects shall be used to implement heuristics, metrics and algorithms for my tool, UNDERTAKER–CHECKPATCH. In the following, I summarize the key points of my analysis and the challenges the tool needs to address to properly detect and analyze variability defects in GIT commits:

- We can detect variability related defects with the UNDERTAKER tool, which generates defect reports containing the contradictory boolean formula of each defect.
- *Missing defects* and *kconfig defects* can be caused and fixed by changes to KCONFIG and by changes to the source code. *Code defects* can only be caused and fixed by changes to the source code, whereas the reparation of such may include optional changes to KCONFIG.
- The causes of *missing defects* can be detected by parsing the defect report and by checking which of the referenced KCONFIG features is not defined in KCONFIG and thereby absent in the variability models.
- If a KCONFIG feature A depends on an undefined feature B, feature A can always evaluate true or false. Moreover, all references on feature A cause *missing defects* (*symbolic integrity violations*). Consequently and in addition to `#ifdef` blocks, UNDERTAKER–CHECKPATCH needs to analyze and track changes to KCONFIG files as well. Thereby, all changes to KCONFIG files need to be cross-checked with the *implementation space* and the *configuration space* in order to detect potential violations, such as broken references.
- The causes of *kconfig defects* cannot be automatically detected. However, we can simplify the contradictory boolean formula of the defect report. The UNDERTAKER generates a minimally unsatisfiable subset, which reduces the search space from hundreds of thousands of literals (KCONFIG features) to just a few (two to ten).
- A similar problems applies for *code defects*. But as *code defects* are independent from the underlying variability model, the analysis of such can be simplified by checking the CPP structure of the affected `#ifdef` block.

In the following chapter, I will describe how my tool, UNDERTAKER–CHECKPATCH, addresses the mentioned issues.

Chapter 4

Implementation

The UNDERTAKER–CHECKPATCH tool ships the basic functionality to examine a specified PATCH file for introducing, changing or fixing variability related defects. In addition to that, the tool can further analyze the causes of defects, parse KCONFIG files and correlate those changes to variability defects. This chapter outlines each functionality of the tool and describes the implemented algorithms.

4.1 Reporting of Defects

I instrument the aforementioned UNDERTAKER tool to detect variability bugs. As the functionality is related to PATCH files, there are two states of the affected files:

- File state **A**: before applying the PATCH
- File state **B**: after applying the PATCH

In order to find changes related to variability anomalies, I need to treat both file states separately, to (a) detect defects, and (b) correlate both versions. The entire process requires the following successive steps:

❶ Dead analysis of file state **A**: I utilize the tool `lsdiff` [13] to get a list of files that are changed by the specified PATCH. Depending if a model is specified or not, UNDERTAKER–CHECKPATCH then automatically extracts a variability model from the given Linux source tree. Afterwards, I trigger the UNDERTAKER to report variability defects of each file and accordingly assign them to the `#ifdef` blocks.

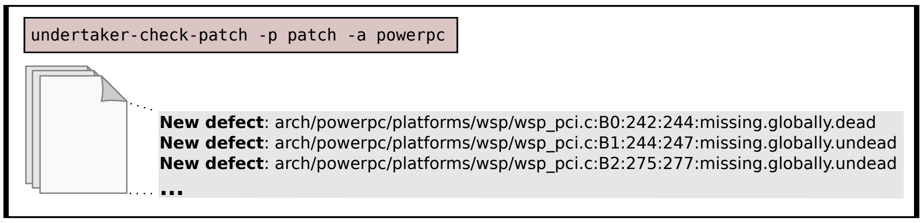
❷ Parsing the PATCH file: As `#ifdef` blocks are potentially affected by changes to the source files, UNDERTAKER–CHECKPATCH parses the PATCH file in a second step to update each block's line ranges. This step is required to have an exact mapping from blocks of file state **A** to the blocks of file state **B**.

[13]http://linux.die.net/man/1/lsdiff

❸ Applying the PATCH, and dead analysis of file state **B**: As soon as the PATCH file is applied, the dead analysis on file state **B** can take place. In case the PATCH file changes KCONFIG files, UNDERTAKER–CHECKPATCH extracts new models in order to propagate the changes made to the *configuration space*. The defects are accordingly assigned to a new set of blocks.

❹ Reporting Defects: At this point, the two sets of blocks can be compared to check for changes in the blocks. After the blocks of both file states are mapped, the tool reports all changes to defects such as the introduction, the reparation of the defect, or changes of the defect's classification.

To conclude. UNDERTAKER–CHECKPATCH ships the functionality to examine the changes to variability related defects triggered by a specified PATCH file. It reports which defects are introduced, changed and repaired at specific source file locations. Figure 4.1 shows an exemplary defect report of the UNDERTAKER–CHECKPATCH tool. It checks a GIT commit that introduces *missing defects*. The output displays that defects are introduced, with the exact source file locations (including the ranges of the `#ifdef` block), and shows the corresponding defect class. The flag "`-a powerpc`" specifies that only models for the `PowerPC` architecture need to be considered; the specification of an architecture further reduces the execution time of the tool as less variability models need to be extracted.



```
undertaker-check-patch -p patch -a powerpc
```

```
New defect: arch/powerpc/platforms/wsp/wsp_pci.c:B0:242:244:missing.globally.dead
New defect: arch/powerpc/platforms/wsp/wsp_pci.c:B1:244:247:missing.globally.undead
New defect: arch/powerpc/platforms/wsp/wsp_pci.c:B2:275:277:missing.globally.undead
...
```

Figure 4.1 – Sample report of the UNDERTAKER–CHECKPATCH tool for GIT commit `f352c7251255effe6c2326190f1378adbd142aa3`

4.2 Analyzing Defects

Besides the basic functionality to report changes of variability defects by a specified PATCH file, the UNDERTAKER–CHECKPATCH tool empowers the user to further *analyze the detected defects*. When you enable this functionality (`--check`), the tool analyzes each defect class separately, and reports the defects' causes as far as possible.

4.2.1 Analysis of Symbolic Defects

The analysis of *referential integrity violations* is comparatively fast. A report for this defect class needs to display all KCONFIG items that are causing the *missing defect* – such are not defined in KCONFIG and thereby not defined in the extracted variability model. Consequently, the tool cross checks *missing defects* with the extracted variability model of file state B. The introduced *missing defects* of Figure 4.1 on page 48 are additionally reported as illustrated in Listing 4.2.1; UNDERTAKER–CHECKPATCH reports that the referenced KCONFIG feature CONFIG_WSP_DD1_WORKAROUND_DD1_TCE_BUGS“ is referenced, but not defined”.

```
 Reporting defects:
New defect: ↘
    arch/powerpc/platforms/wsp/wsp_pci.c:B0:242:244: ↘
    missing.globally.dead
New defect: ↘
    arch/powerpc/platforms/wsp/wsp_pci.c:B1:244:247: ↘
    missing.globally.undead
...
  Analyzing defects:
arch/powerpc/platforms/wsp/wsp_pci.c:B0:242:244: ↘
    missing.globally.dead: ↘
    CONFIG_WSP_DD1_WORKAROUND_DD1_TCE_BUGS referenced but ↘
    not defined
arch/powerpc/platforms/wsp/wsp_pci.c:B1:244:247: ↘
    missing.globally.undead: ↘
    CONFIG_WSP_DD1_WORKAROUND_DD1_TCE_BUGS referenced but ↘
    not defined
...
```

Listing 4.2.1 – Text snippet of UNDERTAKER–CHECKPATCH output for GIT commit f352c7251255effe6c2326190f1378adbd142aa3. After the reporting of defects, UNDERTAKER–CHECKPATCH further analyzes the defects (--check) and reveals the defect causing KCONFIG features.

4.2.2 Analysis of KCONFIG files

Changes to KCONFIG files have a direct impact on the variability model of the Linux kernel. In Chapter 3 we discussed that such changes are likely to cause *referential integrity violations*; a feature may be removed or renamed, or depend on a another feature that is absent in the *configuration space*. In most cases, the causing commits did not propagate the changes made in the KCONFIG space to the rest of the system, and thereby manifest in dead and undead `#ifdef` blocks and KCONFIG features. Each development cycle of Linux, averagely 600 GIT commits touch KCONFIG files, and potentially violate the referential integrity of the system.

In this context I implemented the functionality to parse and detect changes in KCONFIG. Hence, and in addition to the previously mentioned functionality, UNDERTAKER–CHECKPATCH reports changes to KCONFIG files that cause *referential integrity violations*. The reports cover the renaming and removal of entire KCONFIG features, as well as changes to feature constraints (i.e., dependencies and selects). In case a feature is renamed or removed, the tool searches for references in file state B, and reports such to the users. In order to avoid information overload, I decide to only report files that contain broken references without displaying additional information (e.g., source lines). By combining the information, which KCONFIG features are referenced in the contradictory formulae, and which features are removed from KCONFIG, UNDERTAKER–CHECKPATCH can report the exact cause and effect of changes to the *configuration space*. In case a referential integrity violation is met, the tool reports the affected KCONFIG identifier and the affected files (see Listing 4.2.2). In case a PATCH renames or adds a dependency or select statement in KCONFIG, UNDERTAKER–CHECKPATCH checks if the feature is defined in the extracted variability model of file state B.

```
 Analyzing defects:
Patch removes item CONFIG_USB_MUSB_HOST which is still ↘
   referenced in:
arch/arm/mach-ux500/usb.c
arch/arm/mach-davinci/board-da830-evm.c
arch/arm/mach-davinci/usb.c
arch/arm/mach-omap2/board-n8x0.c
```

Listing 4.2.2 – Text snippet of UNDERTAKER–CHECKPATCH output for GIT commit 622859634a663c5e55d0e2a2cdbb55ac058d97b3, which removes several features from KCONFIG but does not propagte the changes to the rest of the system, such as source files. UNDERTAKER–CHECKPATCH alerts the user which removed KCONFIG features are still referenced in which files.

4.2.3 Analysis of KCONFIG defects

Kconfig defects occur due to contradictions in the *configuration space* (e.g., broken dependencies in KCONFIG), as well as due to contradictory preconditions in the *implementation space*. In general, it is beyond our power to reduce a boolean formula to single literals in order to correct the contradiction or tautology. This fact has a significant impact on the feasibility of the analysis of this defect class. Hence, UNDERTAKER–CHECKPATCH ships the functionality to generate a minimally unsatisfiable subset of the contradictory boolean formula. This function reduces the actual expenditure to analyze a KCONFIG defect. However, the user needs to study the MUS formula by her own in order to further analyze the problem. Since the generation of MUS formulae increases the execution time of the tool, this functionality needs to be enabled via an additional parameter. Depending on the amount of #ifdef blocks in the file and the size of the underlying variability models, the MUS generation may take up to 50 seconds per file.

4.2.4 Analysis of Code Defects

The analysis of code defects needs to cover two cases. First, a code defect can be caused by previously defined features in a source file. Second, code defects may also stem from double checked KCONFIG features. Hence, UNDERTAKER–CHECKPATCH first checks if KCONFIG features in the precondition of the defect affected block are previously (un)defined in the file. In this case, the tool reports the issue. Otherwise, the precondition of the defect affected block is printed. Again, the user needs to check the formula by her own. However, such preconditions are human-understandable and easy to read so that the analysis of the respective defect is simplified in every case. Listing 4.2.3 shows an exemplary output of a precondition of a *code defect* in Linux v3.10; arbitrary complex CPP structures can thereby be reduced to a fairly small formula that represents the precondition of the defect affected #ifdef block. The displayed precondition of block B9 indicates that (a) block B9 references CONFIG_PM and (b) that B9 is enclosed by block B8, which references the same feature.

```
 Analyzing defects:
...
drivers/usb/core/hub.c:B9:2830:3250:code.globally.undead: ↘
    there is a tautology in the block’s precondition
B9
&& ( B9 <-> B8 && CONFIG_PM )
&& ( B8 <-> CONFIG_PM )
&& B00
```

Listing 4.2.3 – Text snippet of UNDERTAKER–CHECKPATCH output for GIT commit 84ebc10294a3d7be4c66f51070b7aedbaa24de9b, which renames block B8 and B9 to the same identifier and thereby causes a *code defect*. The displayed precondition of block B9 indicates that (a) block B9 references CONFIG_PM and (b) that B9 is enclosed by block B8, which references the same feature.

4.3 Summary

I implemented a tool, UNDERTAKER–CHECKPATCH, which analyzes specified PATCH files and accordingly reports changes to defects such as newly introduced or fixed defects. Defects can also be correlated to changes in KCONFIG and vice versa. Additionally, the tool ships the functionality to further analyze the causes of defects, and displays missing KCONFIG items, a block's precondition or the defect causing formula. The tool's reports cover all defects that can be detected by the UNDERTAKER tool and reveals changes to defect affected blocks as well as the introduction and reparation of defects. The tool empowers the user to get a clear overview of how a PATCH file changes and potentially violates the variability of the Linux kernel. The additional defect analysis has *100 percent accuracy* for *symbolic integrity violations*. All missing items can be detected and are reported with care to allow a precise understanding of a defect's cause. Furthermore, changes to KCONFIG files are detected and cross checked with the *implementation* and *configuration space* in order to alert referential violations caused by the respective change to KCONFIG. In case of *logic integrity violations*, the tool is checking the aforementioned trivial cases and otherwise displays a minimally unsatisfiable subset or the block's precondition, to simplify further examinations of the problem.

The implementation is influenced by and bases on the insights of my empirical case study, as described in Chapter 3. Furthermore, the UNDERTAKER–CHECKPATCH tool extends the state-of-the-art detection of such defects by further analyzing a defect's cause. UNDERTAKER–CHECKPATCH is implemented in the PYTHON programming language.

Chapter 5

Evaluation

The evaluation of the UNDERTAKER–CHECKPATCH tool (see Chapter 4) is split into two phases, on two independent data sets. In Chapter 3, I manually mapped variability related defects to GIT commits which I will use in the first evaluation. I will manually validate all reports made by UNDERTAKER–CHECKPATCH and describe the results. Second, I run the tool on the current development branch of Linux v3.16 to demonstrate how the tool can be used in daily Linux development.

5.1 Evaluation of Previously Matched GIT commits

Chapter 3 describes the processes of how we can detect, track and analyze variability related bugs in order to find the defect causing and defect fixing GIT commits. I identified *197 unique* GIT *commits* to cause and fix 277 of 768 *missing defects*, 53 of 101 *kconfig defects* and 54 *code defects* which I detected between Linux v2.6.29 and Linux v3.12. In this evaluation I generate a mapping in the other direction, *from* GIT *commits to variability related defects*, by running UNDERTAKER–CHECKPATCH on this set of GIT commits.

I manually checked the validity of each defect report of UNDERTAKER–CHECKPATCH to be sure that the tool works correctly and that it does not report false positives. Hence, UNDERTAKER–CHECKPATCH reports that the 199 GIT commits of Chapter 3 cause 404 *missing*, 63 *kconfig*, and 100 *code defects* (Table 5.1), whereas 505 *missing*, 73 *kconfig*, and 74 *code defects* are repaired. The results show that the 199 commits cause much more defects than I analyzed before (denoted in brackets in the Table 5.1). I explain this observation with the fact (a) that I did not analyze all detected defects (277 of 768 *missing*, 53 of 101 *kconfig* and 54 of 54 *code defects*), and (b) that some of the *additional defects* exceed the interval of analyzed Linux versions (v2.6.29 – v3.12), so that they are excluded from previous data. A total sum of 837 *missing defects* is reported as *unchanged* by the tool; such are present before and after applying the

GIT commit. I explain this number with multiple GIT commits touching the same files which contain many defects. Especially the `blackfin` architecture shows a high defect rate from Linux v2.6.34 until v3.3, where some of the analyzed GIT commits include more than 100 unchanged missing defects.

Moreover, by using the optional functionality to further analyze reported defects, the UNDERTAKER–CHECKPATCH tool parses KCONFIG files. In this case, the tool searches (a) for changes that remove KCONFIG features without propagating the change to other files (source and KCONFIG), or (b) changes that edit a *select or dependency statement* of a KCONFIG feature definition. Newly referenced features, which are not defined in KCONFIG, constitute *referential integrity violations*. UNDERTAKER–CHECKPATCH reports 451 *referential integrity violations*, 23 broken dependencies and three broken selects.

Defect Class	Introduced	Repaired	Changed To	Unchanged
Missing	404 (127)	505 (348)	43	837
Kconfig	62 (9)	73 (20)	1	29
Code	110 (56)	74 (20)	29	38

Table 5.1 – Data of how many *additional* variability related defects the UNDERTAKER–CHECK–PATCH tool reveals by checking the previously matched GIT commits from Section 3.1.2.

Table 5.2 on page 55 shows the benchmarks of UNDERTAKER–CHECKPATCH on a workstation with an Intel Core i7 Quad-Core and 16 GB of RAM, what I consider to be a conventional setup of a Linux developer. In all cases, the tool is triggered with the optional functionality to further analyze detected defects (`--check`). As a matter of fact, the runs for GIT commits related to *symbolic integrity violations* are more time consuming since they can be completely analyzed to detect the causes of defects. In contrast to that, commits related to *logic integrity violations* are faster to analyze since only the preconditions of the affected `#ifdef` blocks or previously (un)defined features are displayed by the tool. However, you may be surprised by the difference of more than a double of execution time between both defect classes. I found out that there are many GIT commits in the set of symbolic defects that are touching a huge amount of files (>100 files) or introduce many defects, and thereby result in higher execution times. As a result, the measured run-times range from less than a second to over 20 minutes. The longest measured run of UNDERTAKER–CHECKPATCH is on a GIT commit that is merging two subdirectories in the `drivers` subsystem and thereby touches 571 files, taking almost 21 minutes of execution time.

GIT Commit of Defect Class	Average Run Time
Symbolic Defects	40.50 seconds
Kconfig Defects	18.65 seconds
Code Defects	14.90 seconds

Table 5.2 – Performance measurement of the UNDERTAKER–CHECKPATCH tool on previously matched GIT commits from Section 3.1.2.

Note, that the benchmarks also include such GIT commits that are changing KCONFIG files and thereby require the extraction of new variability models from the source. This process takes two minutes on average but can be significantly improved when you specify a main architecture (`--arch`). With a specified main architecture UNDERTAKER–CHECKPATCH extracts a variability model only for this architecture, which (a) improves the process of extracting models and (b) reduces the time to detect variability related defects as the feature constraints of only one model need to be loaded. However, I recommend to use this functionality only if the specified PATCH is changing files of one and the same architecture, or if you wish to forcefully check a single variability model. As a consequence, I decide to leave this choice to the user to (a) provide a clear and understandable interface and (b) to generate models for each architecture by default as this avoids reports of false positives; only such blocks are reported that are dead or undead for all architectures.

5.2 Evaluation on Linux Development Branch

To evaluate the UNDERTAKER–CHECKPATCH tool on an independent data set, I trigger the tool on the current Linux kernel development branch, Linux v3.16-rc1. The two data sets are independent from another, since the sets of GIT commits do not intersect ({v2.6.29–v3.12} and {v3.16-rc1}). For this evaluation, I developed a PYTHON script that runs the tool on each of the 12 078 commits that have been merged into the main Linux GIT repository of Linus Torvalds between Linux v3.15 and v3.16-rc1. I want to prove, (a) that the tool is working on a data set that has not been subject of my analysis, and (b) that the tool performs well and that it can be used in daily Linux development.

Table 5.3 contains the result of this evaluation process that ran 39.2 hours (11.68 seconds per commit) on a machine with an Intel Core i7 Quad-Core processor and 16 GB of RAM. 18 *missing* and one *kconfig defect* are introduced, whereas 325 *missing*, 18 *kconfig*, and 6 *code defects* are fixed. I consider this to be a positive tendency as significantly more defects are repaired than introduced. However, the number of *unchanged defects* is striking the eye: UNDERTAKER–CHECKPATCH identifies 2434 defects that remain unchanged in the analyzed set of GIT commits.

Defect Class	**Introduced**	**Repaired**	**Unchanged**
Missing	18	325	1607
Kconfig	1	18	152
Code	0	6	675

Table 5.3 – Data of how many changes to variability related defects the UNDERTAKER–CHECKPATCH tool reveals by checking the Linux development between Linux v3.15 and v3.16-rc1.

Although the number of introduced defects is minute, there is another case of GIT commits causing variability bugs: *changes to the configuration space*. Between Linux v3.15 and v3.16-rc1 590 of 12 078 commits touch KCONFIG files of which 20 commits cause at least 212 *referential integrity violations*. As aforementioned, UNDERTAKER–CHECKPATCH is only reporting affected files, so that the number of actual defects may be higher.

5.3 Summary

I evaluated the UNDERTAKER–CHECKPATCH tool on two independent, intersection free data sets. First, the tool reveals hundreds of additional variability related defects and broken references for the set of previously analyzed GIT commits from Chapter 3. I outline, that the tool's performance is determined by the size and complexity of a specified PATCH file. The second data set contains 12 078 GIT commits in the development cycle from Linux v3.15 to v3.16-rc1. UNDERTAKER–CHECKPATCH reveals more than 2400 unchanged defects, and reports changes such as newly introduced or fixed defects. Additionally, the tool identifies that 20 GIT commits are critical to the referential integrity of the system, as they change the *configuration space* without properly propagating the changes to the rest of the kernel's source code.

Chapter 6

Discussion

The evaluation yields that the UNDERTAKER–CHECKPATCH tool is able to detect, report, and further analyze variability related defects that are triggered by PATCH files. In this chapter, I will discuss the *usability* of this tool, and how I improved the quality of the underlying UNDERTAKER tool.

6.1 Use Cases

Developer Tool

First, the tool can be used by a Linux developer to check a PATCH file before sending it to the maintainer or a specific mailing list, and before integrating the PATCH into the GIT repository. UNDERTAKER–CHECKPATCH empowers the user to check her code for variability related defects, which are barely possible to detect without proper tool support. By doing so, the tool additionally reveals such defects that are present before and after the PATCH. These unchanged defects can then be reported to the maintainer in charge of the source code or posted in the bug tracking system of the Linux kernel[14]. In all cases, UNDERTAKER–CHECKPATCH helps developers to (a) avoid the introduction of variability defects, (b) it reveals existing defects in the code, and (c) it helps to analyze and further fix the respective defects.

In personal correspondence via email, Greg Kroah-Hartman[15] confirmed the benefit of tools, such as UNDERTAKER–CHECKPATCH: “We do have people going over the tree and removing unused config options (options that are never set and the code that never got built from them.) So that is good work to do, and if you have a tool to do it, that’s even better.” This issue affects both, *referential* and *logic defects*.

[14]https://bugzilla.kernel.org/

[15]http://en.wikipedia.org/wiki/Greg_Kroah-Hartman

Automated Testing Systems

In Section 5.2, I evaluated UNDERTAKER–CHECKPATCH on the current Linux development branch (Linux v3.16-rc1) with the help of a PYTHON script, which first extracts desired GIT commits and then runs UNDERTAKER–CHECKPATCH. The usage of comparable scripts suggest the *second use case*: the tool can run on dedicated build and test servers and send reports to the authors of GIT commits and the maintainers of affected source files. The realization of such systems puts the quality management of the Linux kernel on the level of system configurability. Thereby, developers can be alerted and fix the defects before releasing new versions. My evaluations show an execution time of 19.5 seconds on average per GIT commit, which I consider to be sufficiently fast to run on a server. Greg Kroah-Hartman states that "if it is something to check our code base, sure, we [the Linux developers] should use it, we have automated testing systems it could be put into."

6.2 Contributions to the UNDERTAKER

UNDERTAKER–CHECKPATCH is determined by previous research work and tools, mainly by the UNDERTAKER tool. As a consequence, I manually validated all data that is generated by external tools, and thereby submitted several bug fixes and bug reports to the UNDERTAKER tool, described as follows:

- I introduced a new defect class to the UNDERTAKER. This class avoids the inclusion of such defects, that cannot be related to KCONFIG, and thereby constitute false positives (see *"no* KCONFIG *defects"* in 3.1.2).
- One bug is related to the detection of undead blocks. In this case, the UNDERTAKER did not report certain undead blocks. Such blocks are located in the first indentation level and are thereby not nested in other `#ifdef` blocks.
- Another bug is related to the variability models. During manual analysis of missing defects, I discovered cases of false positives where an inconsistent mapping of default values for boolean KCONFIG features caused false positives.
- Furthermore, several models of many Linux versions were incomplete or even absent due to a bug in the variability extractor of the UNDERTAKER; the extraction terminated while parsing some statements of the KCONFIG language. The consequence are hundreds of false positives in affected Linux versions.

In addition to previously mentioned issues, I developed an approach to *increase the number of detected defects*. In order to detect dead and undead `#ifdef` blocks, the UNDERTAKER cross checks the CPP constraints of each source file with the constraints of *each* extracted variability model. Hence, only blocks that are defect for all architectures are then considered to be a real bug. However, this approach is not practical for all source files, especially for those in the hardware abstraction layer (e.g., `arch/x86` for the `x86` architecture). Such files are *architectural dependent* and, as a consequence, shall not be cross checked with other variability models. By changing this calling convention of the UNDERTAKER, I (a) managed to increase the number of variability defects, and (b) correct some wrong classifications of defects.

Chapter 7

Conclusion

Variability related defects present a delicate and error-prone issue to the Linux community: "The tricky parts these days are configuration issues, i.e. code that fails to build for certain configurations, due to various reasons (forgot to handle a case in another `#ifdef` branch, code inside vs. outside `#ifdef`, different indirect includes on different architectures [...]" [16]. Such bugs require a tool-based analysis and oftentimes manifest in critical errors to the system, such as `NULL` pointer dereferences, buffer overflows, or API violations [Abal et al., 2014].

In this thesis I presented a tool, UNDERTAKER–CHECKPATCH, which addresses the issue of variability related defects in the Linux kernel. The tool checks PATCH files for variaiblity related defects, and further analyzes the causes and effects of detected defects. In addition to that, UNDERTAKER–CHECKPATCH parses changes of KCONFIG files and reveals potential *symbolic integrity violations*. The tool is easy to use and it integrates seamlessly into a Linux developer's work flow. UNDERTAKER–CHECKPATCH can be used in the context of continuous integration and quality management approaches by the Linux community, as it can be used to automatically report variability defects to the authors of affected GIT commits and the responsible maintainers.

The implementation as well as the evaluation of the tool are influenced by my empirical case study of feature consistency and variability related bugs on 24 versions of the Linux kernel. The case study constitutes the first research work on both, *referential* and *logic integrity violations*. My results present a very different image of the quality and the quantity of the problem than previous research work, and enforce the need of a tool that can actually be used by Linux developers. As a consequence, I assume that UNDERTAKER–CHECKPATCH will be gladly accepted by the Linux community.

[16]`http://lists.linuxfoundation.org/pipermail/ksummit-discuss/2014-June/001010.html`

List of Acronyms

CPP	C preprocessor
GCC	GNU C compiler
LKM	loadable kernel module
MUS	minimally unsatisfiable subset
SLOC	source lines of code
API	application programming interface

List of Figures

List of Tables

Bibliography

Iago Abal, Claus Brabrand, and Andrzej Wasowski. 40 variability bugs in the linux kernel. Technical report, IT University of Copenhagen, 2014.

Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8. doi: 10.1145/502034.502042. URL `http://doi.acm.org/10.1145/502034.502042`.

Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. A Robust Approach for Variability Extraction from the Linux Build System. In ACM Press, editor, *Proceedings of the 16th International Software Product Line Conference*, volume 1, pages 21–30, New York, 2012. ISBN 978-1-4503-1094-9. doi: 10.1145/1966445.1966451. URL `http://www4.cs.fau.de/Publications/2012/dietrich_12_splc.pdf`.

Mark H. Liffiton and Karem A. Sakallah. On finding all minimally unsatisfiable subformulas. In *in Int'l Conf. on Theory and Applications of Satisfiability Testing*, pages 173–186. Springer-Verlag, 2005.

Roberto E. Lopez-herrejon. Understanding feature modularity in feature oriented programming and its implications to aspect oriented. In *Programming'. ECOOP2005 PhDOOS Workshop and Doctoral Symposium*, 2005.

Sarah Nadi, Christian Dietrich, Reinhard Tartler, Ric Holt, and Daniel Lohmann. Linux Variability Anomalies: What Causes Them and How Do They Get Fixed? In Thomas Zimmermann, Massimiliano Di Penta, and Kim Sung, editors, *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 111–120, Los Alamitos, CA, USA, 2013. ISBN 978-1-4673-2936-1. URL `http://www4.cs.fau.de/Publications/2013/nadi_13_msr.pdf`.

Yoann Padioleau, Julia L. Lawall, and Gilles Muller. SmPL: A domain-specific language for specifying collateral evolutions in Linux device drivers. In

International ERCIM Workshop on Software Evolution (2006), Lille, France, apr 2006.

Nicolas Palix, Julia Lawall, and Gilles Muller. Herodotos: A Tool to Expose Bugs' Lives. Rapport de recherche RR-6984, INRIA, 2009. URL `http://hal.inria.fr/inria-00406306`.

Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Gilles Muller, and Julia Lawall. Faults in linux 2.6. *ACM Trans. Comput. Syst.*, 32(2):4:1–4:40, June 2014. ISSN 0734-2071. doi: 10.1145/2619090. URL `http://doi.acm.org/10.1145/2619090`.

D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972. ISSN 0001-0782. doi: 10.1145/361598.361623. URL `http://doi.acm.org/10.1145/361598.361623`.

Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is The Linux Kernel a Software Product Line? In Frank van der Linden and Björn Lundell, editors, *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, 2007.

Julio Sincero, Reinhard Tartler, Christoph Egger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Facing the Linux 8000 Feature Nightmare. In ACM SIGOPS, editor, *Proceedings of ACM European Conference on Computer Systems (EuroSys 2010), Best Posters and Demos Session*, 2010. URL `http://www4.informatik.uni-erlangen.de/Publications/2010/sincero_tartler_eurosys-life.pdf`.

Reinhard Tartler, Julio Sincero, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Configurability Bugs in Linux: The 10000 Feature Challenge. In USENIX Association, editor, *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10), Poster Session*, 2010. URL `http://www4.informatik.uni-erlangen.de/Publications/2010/tartler_sincero_osdi-life.pdf`.

Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time Configurable System Software. In Gernoth Heiser and Christoph Kirsch, editors, *Proceedings of the EuroSys 2011 Conference (EuroSys '11)*, pages 47–60, New York, NY, USA, 2011. ISBN 978-1-4503-0634-8. doi: 10.1145/1966445.1966451. URL `http://www4.informatik.uni-erlangen.de/Publications/2011/tartler_11_eurosys.pdf`.

Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Revealing and Repairing Configuration Inconsistencies in

Large-Scale System Software. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(225):531–551, 2012. doi: 10.1007/s10009-012-0225-2. URL `http://www4.cs.fau.de/Publications/2012/tartler_12_sttt.pdf`.