FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Stefan Hengelein

Analyzing the Internal Consistency of the Linux KConfig Model

Masterarbeit im Fach Informatik

27. Juli 2015

Please cite as:
Stefan Hengelein, “Analyzing the Internal Consistency of the Linux KConfig Model” Master’s thesis, University of Erlangen, Dept. of Computer Science, July 2015.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

Analyzing the Internal Consistency of the Linux KConfig Model

Masterarbeit im Fach Informatik

vorgelegt von

Stefan Hengelein

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuender Hochschullehrer: **PD Dr.-Ing. habil. Daniel Lohmann**

Beginn der Arbeit: **7. Januar 2015**
Abgabe der Arbeit: **7. Juli 2015**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.

Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Alle URL-basierten Quellen wurden, soweit nicht anders angegeben, am 04. Juli 2015 auf ihre Gültigkeit geprüft.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.

I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly. Unless stated otherwise, all URL-based references were checked for validity on July 04th, 2015.

(Stefan Hengelein)
Erlangen, 7. Juli 2015

ABSTRACT

LINUX, as well as other system software, offers a great deal of configurability that allows users to tailor the kernel to their specific needs. Linux/`v4.0` provides more than fourteen thousand configurable options which are declared in the configuration system KCONFIG and implements those options distributed over multiple build phases (KCONFIG, the build system KBUILD and source files). Managing this variability has become increasingly hard and is barely solvable by hand nowadays. To help developers, it is necessary to provide comprehensive tool support to identify problems. Previous approaches already identified many of those defects but did not analyse if there are defects in the declared configurability and therefore did not do a holistic analysis.

In my thesis I close that gap and provide a detailed analysis solely of the in KCONFIG modelled configurability. I provide several approaches that identify new defects, help to identify more pitfalls in the variability of the kernel and use the newly found defects to enrich previous analyses. To tackle these analyses, I re-use existing snapshots of the modelled variability, extract and analyse the relevant information and interactions of options.

With this investigation, I have identified more defects and potentially harmful or gratuitous attributes of options. Furthermore I studied the impact of my findings to other analyses and integrated my approaches partially into the tools that perform these analyses. However, the number of defective KCONFIG options is small in relation to the total number of declared options.

KURZFASSUNG

Ebenso wie andere Betriebssystemkerne, bietet LINUX seinen Nutzern eine große Anzahl an konfigurierbaren Optionen um den Kern nach ihren eigenen Vorstellungen maßzuschneidern. Linux/v4.0 bietet hierbei mehr als vierzehn tausend solcher in KCONFIG deklarierten Optionen an. Die Implementierung dieser Variabilität ist jedoch über mehrere Phasen des Erstellungsprozesses des Kerns verteilt (KCONFIG selbst, das Build-System KBUILD und in Quelldateien). Diese Variabilität handzuhaben wurde im Laufe der Jahre immer komplexer und eine Prüfung der Konsistenz über diese Implementierungsebenen ist heutzutage kaum noch von Hand möglich. Deswegen ist eine umfassende Unterstützung für Entwickler durch Werkzeuge nötig. Vorherige Ansätze waren in der Lage viele Variabilitätsdefekte aufzudecken, haben jedoch nicht in der modellierten Konfigurierbarkeit nach Fehlern gesucht, sondern Quelldateien in Kombination mit extrahierten KCONFIG und KBUILD Bedingungen. Dieses Vorgehen hat jedoch zu keiner ganzheitlichen Analyse geführt.

In dieser Arbeit präsentiere ich daher, wie ich diese Lücke schließe und zeige eine detailierte Analyse der modellierten Konfigurierbarkeit auf. Ich verwende verschiedene Ansätze, um Probleme zu identifizieren, die spätere Analysen bereichern und auf weitere problematische Punkte in der Variabilität aufmerksam machen. Um diese Analysen anzugehen, verwende ich bestehende Abbilder der in KCONFIG modellierten Variabilität wieder und extrahiere und analysiere die relevanten Informationen und Interaktionen dieser Optionen.

Mit dieser Untersuchung habe ich weitere Defekte und potentiell schädliche oder unnötige Attribute von Optionen identifiziert und habe zusätzlich den Einfluss dieser Entdeckungen auf weitere Analysen späterer Phasen des Erstellungsprozesses des Kernel-Abbilds untersucht und habe meine Ansätze partiell in die Werkzeuge integriert, die diese Analysen durchführen. Verglichen mit der Gesamtanzahl der Optionen in KCONFIG, ist jedoch ist die Anzahl an fehlerhaften KCONFIG Optionen gering.

CONTENTS

1 INTRODUCTION

Software is nowadays often configurable. Different customers might demand similar but not identical products. Companies then are faced with the decision to redevelop a product from scratch or to carve out the differences in the requirements and employ means of configurability to re-use as much code as possible in order to minimize development costs. Other firms start directly with domain analysis and engineering to create a configurable product to improve the development process and product quality, reduce the time-to-market and achieve mass customization [Sin+07] to reach a broad range of customers. This configurability enables developers to tailor software to the needs of their customers.

System software is no exception to these procedures as Parnas [Par72] stated in 1972: "operating systems and system software in general demand a highly modularized design". This kind of software often provides support for heterogeneous hardware and different use cases ranging from small embedded devices to desktop machines or nodes in clusters. LINUX is a prime example for this design. In over ten years of development the number of configurable options has more than tripled as displayed in Figure 1.1 and the figures keep rising. Linux/`v4.1`, for instance, offers more than fourteen thousand five hundred configurable options provided by the supplied KCONFIG tools and models. The implementation of these options is distributed over multiple levels of the software generation process, which are namely the configuration system KCONFIG, the build system KBUILD and the C preprocessor (CPP). Independent of the KCONFIG controlled variability, branches in LINUX's linker scripts render the kernel variable at link-time and modules can be loaded or unloaded at runtime. In practise, this enormous variability is difficult and error-prone to manage and has already lead to thousands of variability defects in LINUX [Tar+11b]. These defects manifest in seemingly configuration-conditional `#ifdef`-blocks that can never be selected (dead) or de-selected (undead), files that can never be compiled or even defects that are only exposed at compile time [Tar+14] or lead to bugs at runtime (e.g. null pointer dereference, use after free, buffer overflows, reads outside of array boundaries, etc.) [ABW14].

In their dissertations in the context of the VAMOS-project [Web], Sincero [Sin13] and Tartler [Tar13] investigated scalable methods and tools to identify variability related bugs. They implemented the UNDERTAKER-toolchain as a key element for their analyses. This toolchain analyses source files and employs an image of the in KCONFIG modelled configurability and build system

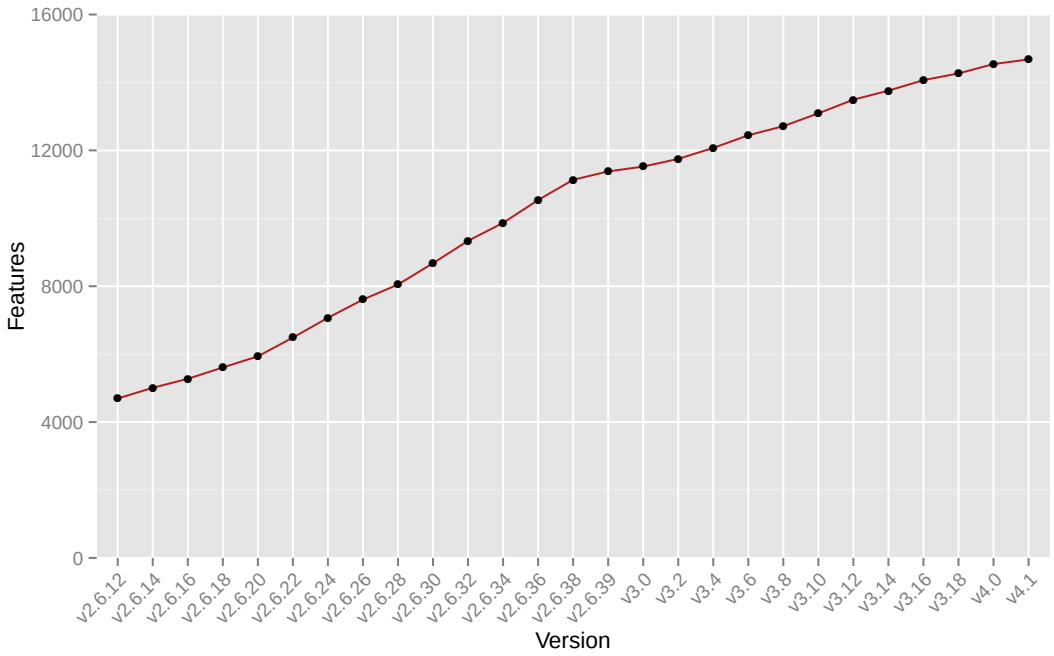

Figure 1.1 – The increase of KCONFIG features from v2.6.12 (Jun '05) to v4.1 (Jun '15).

constraints – a variability model – to identify dead or undead CPP blocks or whole files. Later, they extended their toolchain to support and enhance the test coverage of static checkers [Tar+14]. They utilise a holistic view of the static variability in LINUX, but did not reach a holistic analysis of this static variability. Therefore it was not investigated if some of the defects they have identified originate purely in defective declarations of options in KCONFIG or if the modelled variability lead to other, not yet discovered, defects.

In my thesis I therefore analyse the modelled variability, the configuration space, which is implemented in KCONFIG in the LINUX kernel, to complete previous approaches and achieve a holistic analysis of the static variability in LINUX. This kernel provides a wide variety of options and the number of those features continuously increases and reached 14679 for version `v4.1`. A scalable approach is therefore of utmost importance and techniques like the ones used by Rincón et al. [Rin+14], which scale up to 150 features, are not feasible for LINUX.

In this thesis, I present the extension of the UNDERTAKER to analyse the snapshot of the modelled configurability and also present a tool to analyse feature interactions to identify defective or potentially superfluous options or attributes of those options.

First I describe the concepts and tools used to implement the configurability in the LINUX kernel and how each step of the build process leads to a operational kernel image in Chapter 2, followed by a more detailed elaboration about how the UNDERTAKER toolsuite investigates and classifies variability defects and approaches of other researchers. In Chapter 3 I explain how I tackle those

two approaches and how my approach completes the analyses done by Sincero [Sin13] and Tartler [Tar13] to achieve a truely holistic analysis of the modelled and implemented variability in LINUX. Furthermore, I present insights about the implementation of those approaches in Chapter 4, continue with a detailed evaluation of the expanded and created tools in Chapter 5 and finish with a discussion about the implications of the results of my approaches in Chapter 6.

2 FUNDAMENTALS

System software is often highly configurable. LINUX is one of the most prominent examples where this can be observed. With its support of 30 different architectures (x86, arm, arm64, mips, ...) in Linux/`v4.0`, LINUX can be run on small embedded devices but also on more complex desktop PCs or on nodes in huge clusters. Each architecture might require individual code for the startup of the kernel, for memory management or how interrupts are handled by processors. One source of variability is therefore the architecture. These architectures are placed in the `arch/` subdirectory and some of these architectures also contain several *sub-architectures*. These sub-architectures might add support for different instruction sets (e.g. i386 and AMD64) or for different versions of hardware specifications (e.g. ARMv7-A, ARMv7-R).

On top of the abstraction layer for architectures, LINUX supports many different subsystems that implement (mostly) architecture independent code. These subsystems include networking, filesystems, memory management, sound, kernel, libraries and drivers, while the code in the `drivers/` subsystem is up to 60 percent of the overall source code. However, some functionality is mutually exclusive, not every piece of code is always needed and some parts of the code require the presence of other parts of code. To tackle these different requirements, LINUX provides the necessary means to configure the kernel and allow users to tailor the kernel to their individual needs. Embedded systems might require a kernel providing only the needed functionality and a small size of the kernel, while a kernel for desktop machines should support a broad range of drivers and the size of the kernel is not an issue. To provide support for a more generic range of hardware, LINUX allows users to choose how parts of the kernel are compiled. These parts can be either compiled directly in the kernel or as a loadable kernel module (LKM), which the kernel can load on demand at runtime.

The configurability in LINUX is modelled in KCONFIG. This configurability is then exposed as options in configuration tools to users who can decide how they want to compile the kernel and which functionality it should contain. The build process – which follows the choice of configuration – is organised in several phases, which I describe in the following Section 2.1. The implementation of every phase decides which source code artefacts are compiled, depending on the configuration.

However, the implementation process of this variability is not perfect, understanding every aspect of this configurability is hard and developers easily make mistakes that lead to functional bugs or

simply dead code. These issues are aided by the dispersion of the variability implementation across different build phases, the large code base of LINUX and constantly rising number of configuration options (v2.6.12 has 4.696 total KCONFIG options in June '05, v4.0 has 14.679 in June '15).

Sincero [Sin13] and Tartler [Tar13] researched defects that originate in the implementation of variability in source code and in interactions of this variability with the modelled configurability in KCONFIG or other build phases. I describe their work and the work of other researchers briefly in Section 2.6. However, Sincero and Tartler limited their research to variability defects originating in source code or in the interaction with other build phases and did not investigate if there are defects in the modelled configurability in LINUX. This left some questions unanswered:

- Are there options in KCONFIG that can never be chosen but cause defects in later build phases? (dead options)
- Are there options that are always enabled but appear configurable? (undead or false optional [Rin+14])
- Are there other defects that originate in KCONFIG that influence the variability in later stages?

I focus on answering those questions in this thesis.

To present a holistic view of the variability in LINUX, I describe how it is implemented in Section 2.1. It is necessary to acquire a deep understanding of KCONFIG to be able to identify defects in the modelled configurability in LINUX. I provide a detailed explanation of KCONFIG in Section 2.5.

2.1 Static Configurability and Build Process

The configuration and build process in LINUX is organised in several interconnected phases [Tar+14] as seen in Figure 2.1. *static configurability* means, that configuration has to be done before the compilation process is started. As a first step ❶, users have to choose the architecture by setting the environment variable `ARCH` to the intended architecture. For step ❷, the desired configuration has to be created. Users can create this configuration by using the command-line tool CONF or slightly more advanced tools with a graphical interface (MCONF, GCONF or QCONF), provided by KCONFIG. These tools parse and evaluate the architecture specific KCONFIG files, in which the variability is modelled. After the configuration is done, the KCONFIG tools create a `.config` file in the root directory of the LINUX source, to represent the chosen configuration in the format shown in Listing 2.1. Every active option is represented in this file.

```
1 # CONFIG_GLOB_SELFTEST is not set
2 CONFIG_RD_LZ4=y
3 CONFIG_MMU=y
4 CONFIG_DVB_CX24123=m
```

Listing 2.1 – Example of a `.config` file. `CONFIG_GLOB_SELFTEST` is disabled, `CONFIG_RD_LZ4` and `CONFIG_MMU` are enabled and `CONFIG_DVB_CX24123` will be a LKM.

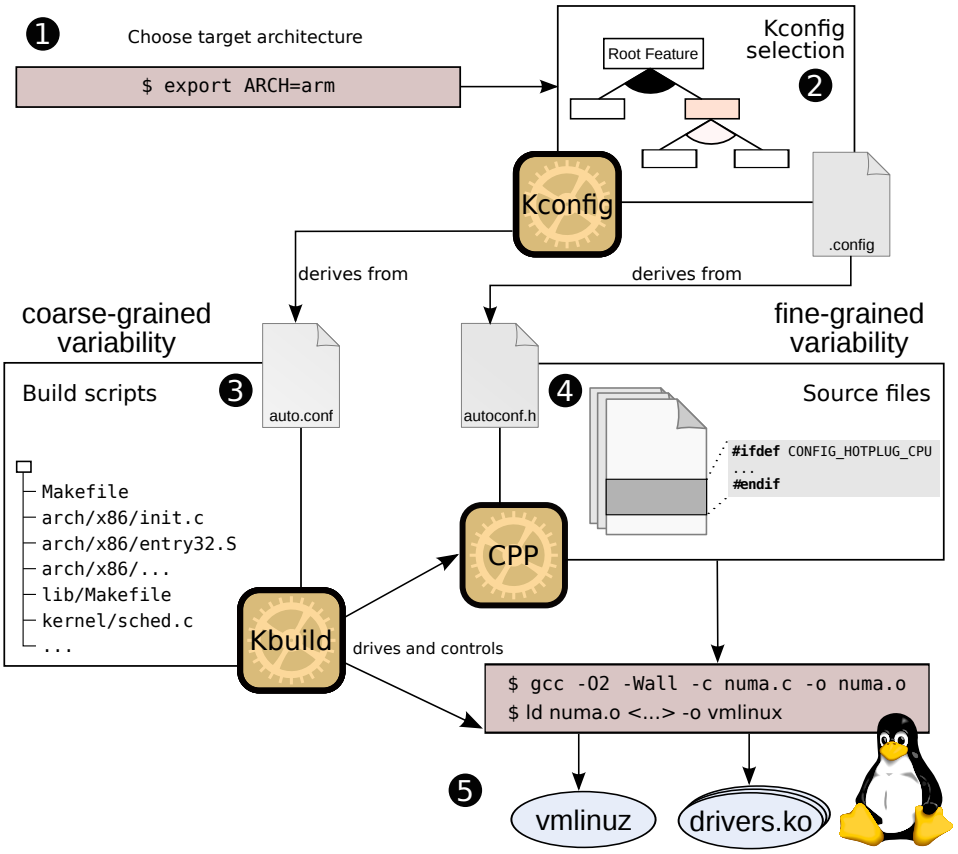

Figure 2.1 – Fine-Grained and Coarse-Grained Variability Implementation in Linux. Figure taken from [Tar+14].

When the build process has started, KBUILD– LINUX's own build system – derives two files of the previously created `.config` file. The first file, `auto.conf`, located in `include/config`, is used by KBUILD in step ❸ to determine which files will be compiled. Listing 2.2 shows an excerpt of this file. In contrast to the `.config` file, `auto.conf` does not contain any disabled symbols and comments. Since KBUILD decides if a folder is included in the traversion of the directory structure and a file is compiled or not, this type of variability is called *coarse-grained-variability*.

```
1 CONFIG_RD_LZ4=y
2 CONFIG_MMU=y
3 CONFIG_DVB_CX24123=m
```

Listing 2.2 – Example of a `auto.conf` file. No disabled options are mentioned here and no comments are included.

The build system includes the second generated file, `include/generated/autoconf.h`, into every compilation unit. Within this file, the relevant bits of the `.config` are translated into CPP

`#define` statements, as shown in Listing 2.3. Developers then are able to use CPP `#ifdef` or `#if defined` statements in source files to decide on a fine-grained level, which part of the file should be compiled, depending on the user selection. This kind of variability is also called *fine-grained variability* [Tar+14].

```
1 #define CONFIG_RD_LZ4 1
2 #define CONFIG_MMU 1
3 #define CONFIG_DVB_CX24123_MODULE 1
```

Listing 2.3 – Example of a `autoconf.h` file, where the options mentioned in `.config` are translated into #define Statements. Options set to "m" get the KCONFIG-conventional _MODULE-suffix for the definition.

After the preprocessor decided in step ❹ which parts of the code are kept, the compiler translates the compilation units into object files and the linker creates in step ❺ a bootable `vmlinuz` kernel image, using the built-in compilation units, and `kernel objects` (.ko) for the linux kernel modules (LKM).

To show how these phases of configurability are interconnected, Tartler [Tar13] used Figure 2.2 in his dissertation. He states the earlier phases dominate later phases and therefore impose a hierarchy of variability on later phases. He sees each phase as a separate layer of variability. Each decisions on an upper layer has a direct impact on the inclusion of *variability points* on lower layers. Variability points are code points in KBUILD or in source files where the decision is made if code artefacts or files are compiled or not.

The first layer l_0 shows the declaration of the formal model, which is implemented within the definitions of KCONFIG options in KCONFIG files with their dependencies. The plain text model is realised in the derived `auto.conf` and `autoconf.h` files, which I specified before. The layers l_1 and l_2 realise the previously mentioned ❸ and ❹ build phases. However, Tartler [Tar13] describes l_3 as run-time variability, which does not fit the Figure 2.2, where l_3 is declared as *Compiler/Language* with `if(CONFIG_F1)` as code example. When the source file with this `if(CONFIG_F1)`-statement is compiled, the CPP will replace `CONFIG_F1` with 1 if it was enabled by the user and therefore defined in `autoconf.h`. If `F1` was not chosen by the user, `CONFIG_F1` will not be defined in `autoconf.h` and the compilation will break with an "undeclared identifier" error. There is currently no mechanism to change the configuration at runtime in LINUX.

Since all configuration happens on l_0 in KCONFIG, this layer is also called *configuration space*. The later layers where the variability is actually implemented in every variability point can also be summarised as *implementation space*.

After this description of the build system, I continue with brief descriptions of each phase of this system.

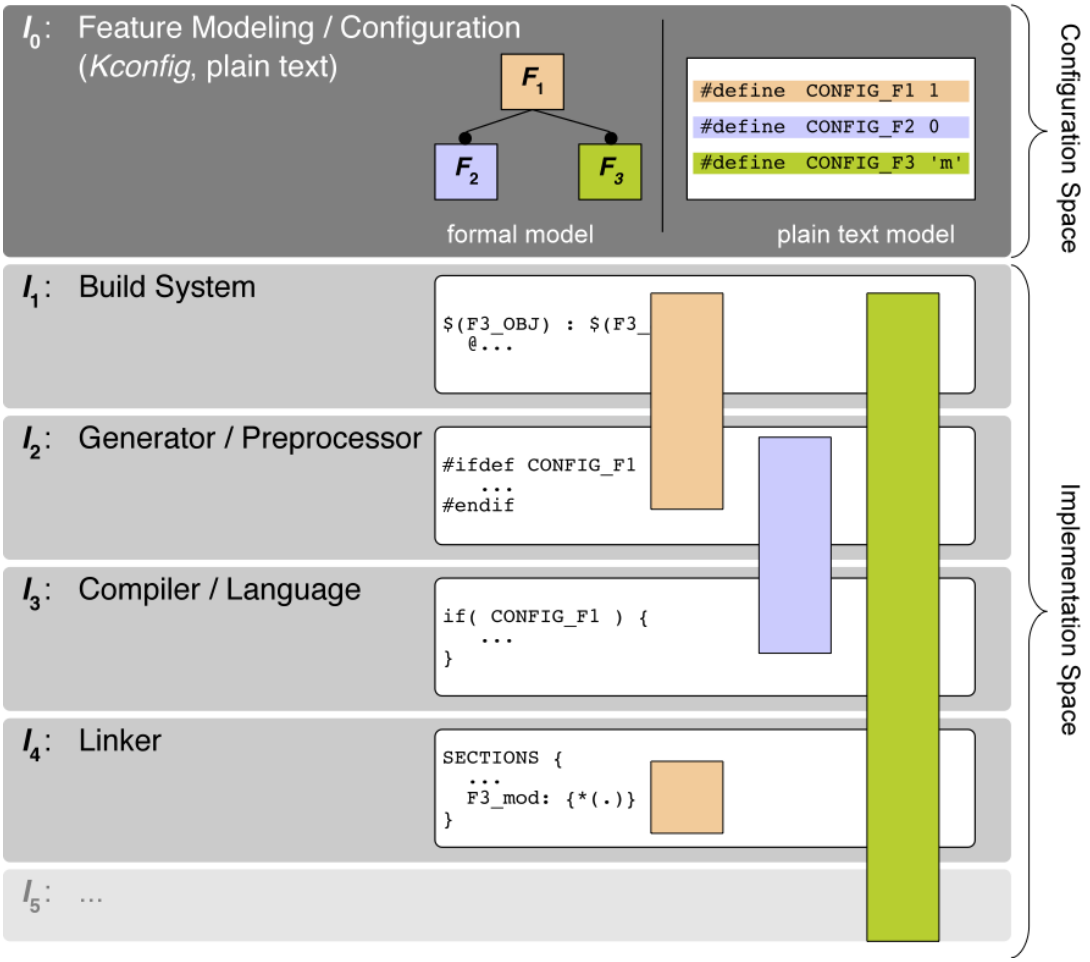

Figure 2.2 – The layers of configurability.

2.2 KCONFIG: Decide Which Options are Enabled

KCONFIG is a domain specific language (DSL) to specify in which way a piece of software is configurable: declare which properties the software has and how these properties interact with or depend on each other. These declarations are usually done in files named Kconfig. Every entry can have its own dependencies or attributes, which are used to determine its visibility, enable other entries and to determine where the entry is placed in the tree structure in which these options are organised.

KCONFIG and its tools were originally developed for the *Linux kernel* and they have been adopted for a wide range of other projects [Ber+12] since then. The KCONFIG-tools employ the declarations in KCONFIG-files to provide a command-line or graphical interface for users to choose their desired configuration.

An example for such a declaration is displayed in Listing 2.4 which shows the entry for option `VM86`.

```
config VM86
    bool "Enable VM86 support" if EXPERT
    default y
    depends on X86_32
    ---help---
      This option is required by programs like DOSEMU to run
      16-bit real mode legacy code on x86 processors. It also may
      be needed by software like XFree86 to initialize some video
      cards via BIOS. Disabling this option saves about 6K.

      If unsure, say Y.
```

Listing 2.4 – Example of an entry in Kconfig.

The first line represents the definition of the option's name `VM86`. The `config` keyword indicates the start of a new config entry, while the following lines define the attributes of that config entry. The attributes in this case are the type of the config option, followed by an input prompt, a condition for this prompt, a default value, dependencies and the help text that is displayed to the user if necessary.

Every line starts with a keyword except after a help keyword started a help text. The parser uses the indentation to recognise which text belongs to a block of help text. The first line after the keyword establishes the start indentation of the help text and the block ends when a line is found that has a smaller indentation.

2.3 KBUILD: Decide Which Files are Compiled

KBUILD is the systematic extension of the GNU MAKE system in LINUX to separate the *what to compile* from *how to compile it*. This separation was first proposed by Michael Elizabeth Chastain in the year 1997[1] who named the pattern "dancing makefiles".

With this pattern, the central build rules are mostly defined in various Makefiles in the `scripts/` directory, while small Makefiles that determine which files will be compiled, are scattered across the source tree. An example for these small files can be seen in Listing 2.5. These Makefiles are traversed recursively. This traversion sometimes happens unconditionally and sometimes depending on the configurability information. Users determine how they want to configure their kernel and KBUILD derives a `auto.conf` file - in MAKE syntax - of the `.config`, which is created by the configuration process. A more detailed description of KBUILD and how the variability is implemented there, can be found at the Master's thesis of Ruprecht [Rup15], who implemented a parser of Makefiles to extract presence conditions for files.

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

```
# unconditional traversal of a directory
obj-y        += irqchip/
# include the zorro/ directory in Kbuild traversal
obj-$(CONFIG_ZORRO)         += zorro/
# compile the sourcefiles in this directory
obj-$(CONFIG_ZORRO)    += zorro.o zorro-driver.o zorro-sysfs.o
```

Listing 2.5 – Example of the implementation of variability in LINUX's Makefiles.

2.4 The C Preprocessor: Decide Which Code is Compiled

The C preprocessor is used to process the source code before the compiler sees the source. The CPP can be used by developers to decide on a *fine grained* level which code will be compiled with the usage of `#if` or `#ifdef` statements (see an example in Listing 2.6) or to expand macros which create source code to simplify the development process for developers or reduce redundant code. When the expression of `#if` or `#ifdef` statement evaluates to false, the compiler will not see the contents of the block.

To determine which source code will be compiled, KBUILD uses the `.config` to derive a `autoconf.h` header which will be included by the build system into every compilation unit to define the respective configuration symbols.

```
1 #ifdef CONFIG_ZORRO
2 // code
3 #else
4 // more code
5 #endif
```

Listing 2.6 – Example for an ifdef-block which implements variability in source files in LINUX.

To avoid the compilation errors in C if-statements, as described in Section 2.1, the KCONFIG maintainer Michal Marek introduced a series of CPP macros to determine the value of a `CONFIG_`-variable with the types **`bool`** or **`tristate`** (which I explain in Section 2.5.4), even if the variable is disabled – thus not defined – without causing a compilation error. These macros are:

- `IS_BUILTIN()` evaluates to 1 if the argument is set to "y", 0 otherwise.
- `IS_MODULE()` evaluates to 1 if the argument is set to "m", 0 otherwise.
- `IS_ENABLED()` evaluates to 1 if the argument is set to "y" or "m", 0 otherwise.
- `IS_REACHABLE()` is similar to `IS_ENABLED()`, but evaluates to 0 when invoked from built-in code when the symbol argument is set to "m".

These macros can be used in C if-statements as well as in CPP `#if`-Statements.

After this short explanation of each build phase, I continue with the detailed explanation of KCONFIG.

2.5 KCONFIG in Detail

The main focus of this thesis is to identify potential errors and dead options in the configuration space, which is implemented in KCONFIG in LINUX. To achieve this goal, it is necessary to acquire a deep understanding about the semantics of KCONFIG. Therefore I continue with a detailed explanation of the configuration space in LINUX.

I included some advanced information about the internal mechanics of KCONFIG in this Section that are not relevant for the understanding of this thesis but might be interesting for a holistic understanding of KCONFIG. This advanced information is shown in boxes with the headline "Additional Information".

Note for the following syntax examples:

- Every term in angle bracket (<>) has to be replaced with the content described in these brackets.
- The meaning of expressions (**<expr>**) is described in Section 2.5.4.1.
- Everything in square brackets ([]) is optional

A more complete example of a kconfig file is shown in Listing 2.7. For the sake of simplicity, I have omitted the help texts of the listed options.

```
mainmenu "Linux KERNEL Configuration"

menuconfig MODULES
        bool "Enable loadable module support"
        option modules

config ARCH
        string
        option env="ARCH"

# Select 32 or 64 bit
config 64BIT
        bool "64-bit kernel" if ARCH = "x86"
        default ARCH != "i386"

menuconfig EXPERT
        bool "Configure expert kernel features"

config X86_32
        def_bool y
        depends on !64BIT
        select HAVE_UID16

config HAVE_UID16
        bool

config OUTPUT_FORMAT
        string
        default "elf32-i386" if X86_32
        default "elf64-x86-64" if X86_64

config PGTABLE_LEVELS
        int
        default 4 if X86_64
        default 3 if X86_PAE
        default 2

source "init/Kconfig"

menu "Processor type and features"

config ZONE_DMA
        bool "DMA mem alloc." if EXPERT
        default y

config SMP
        bool "Symmetric multi-processing support"

config X86_FEATURE_NAMES
        bool "readable names" if EMBEDDED
        default y

endmenu

if X86_32
config X86_EXTENDED_PLATFORM
        bool "extended (non-PC) x86 platforms"
        default y
endif

config PCI
        bool "PCI support"

config X86_INTEL_CE
        bool "CE4100 TV platform"
        depends on PCI

choice
        prompt "High Memory Support"
        default HIGHMEM4G
        depends on X86_32

config NOHIGHMEM
        bool "off"

config HIGHMEM4G
        bool "4GB"

config HIGHMEM64G
        bool "64GB"
        depends on !M486
        select X86_PAE

endchoice

choice
        tristate "USB Gadget Drivers"
[...]

config USB_CONFIGFS
        tristate "USB functions through configfs"

config USB_CONFIGFS_SERIAL
        bool "Generic serial bulk in/out"
        depends on USB_CONFIGFS

config USB_CONFIGFS_ACM
        bool "Abstract Control Model (CDC ACM)"
        depends on USB_CONFIGFS

config USB_AUDIO
        tristate "Audio Gadget"

[...]
endchoice

showif EXPERT
config RANDOMIZE_BASE_MAX_OFFSET
        hex "Maximum kASLR offset allowed"
        range 0x0 0x20000000 if X86_32

config FILE_LOCKING
        bool "Enable POSIX file locking API"

endif
[...]
```

Listing 2.7 – Example of a KConfig file.

KCONFIG programs would create the following tree structure out of this file to show it to users:

```
1  Linux KERNEL Configuration
2
3  +- Enable loadable module support
4  +- 64-bit kernel
5  +- Configure expert kernel features
6  |  +- Maximum kASLR offset allowed
7  |  +- Enable POSIX file locking API
8  +- Processor type and features
9  |  +- DMA memory allocation support
10 |  +- Symmetric multi-processing support
11 |  +- Processor feature human-readable names
12 +- extended (non-PC) x86 platforms
13 +- PCI support
14 |  +- CE4100 TV platform
15 +- High Memory Support
16 |  - off
17 |  - 4GB
18 |  - 64GB
19 +- USB Gadget Drivers
20 |  - USB_CONFIGFS
21 |    +- Generic serial bulk in/out
22 |    +- Abstract Control Model (CDC ACM)
23 |  - Audio Gadget
24 +- ...
```

Listing 2.1 – The corresponding KCONFIG structure to the previous example KCONFIG file.

Each line represents an KCONFIG entry with its own dependencies. These dependencies are used to determine the visibility of an entry. Entries are only visible if the its parent is also visible [Doc], each node with a parent becomes a sub-menu of its parent (lines 5–7). Notice that several options are missing because they do not have an prompt (which I describe in Section 2.5.4).

The example starts in the first line with a `mainmenu` entry, which determines which text should be displayed in the configuration program's title bar, if the program chooses to use it.

Additional Information:

This entry is not relevant for this thesis but for completeness I explain its functionality briefly.

A `mainmenu` is usually the first entry in a structure of Kconfig-files. Older versions of LINUX used this entry more often and sometimes did not define it as first entry (e.g. v2.6.32, architecture `m68k`). Current versions of the KCONFIG parser (v4.0) crash with a "Segmentation Fault" if there is a `mainmenu` entry and it is not the very first entry and is therefore backwards incompatible.

In the following sections I describe the specific language elements of KCONFIG and their meaning.

2.5.1 Menu Entries

Configuration entries, often referred to as menu entries, come in several forms. The most common entry is the definition of an config option which starts with the `config` keyword and is followed by a symbol which explicitly identifies this option. Another entry is `menuconfig` which is similar to `config` except that the declaration of a `menuconfig` also gives a hint to frontends that all suboptions should be displayed as a separate list of options [Doc]. Other entries like `menu` or `choice` help to organise them. These entries will be described in Section 2.5.2 and Section 2.5.3.

Symbols

The identifier which follows a `config` or `menuconfig` declaration is called a symbol. In KCONFIG, symbols can be used to reference the option in dependencies or other interactions like default or select attributes (explained in Section 2.5.4). In later build stages, KBUILD or CPP, symbols are not used in the form they are declared in KCONFIG, but are referenced with the reserved `CONFIG_`[2] prefix added. This prefix is added by the transformations described in Section 2.1. Programming conventions in LINUX reserve this prefix to denote a KCONFIG controlled namespace [Tar13] with which developers should not tamper. However, this convention is violated 184 times in Linux/`v4.0` source files. Options with the `tristate` type get the `_MODULE` suffix in later build stages, if they are set to be built as module.

Internally, KCONFIG sees everything that is part of an expression and that is not an operator, as a symbol. There are two types of symbols: constant and non-constant symbols. The previously mentioned identifier which follows a `config` or `menuconfig` declaration is a non-constant symbol. Non-constant symbols are the most common ones. They solely consist of alphanumeric characters and underscores (e.g. `X86_CPUID`) and are called *non-constant* because they identify a configuration entry that gets a value during the configuration process.

Constant symbols can only be part of expressions and are always surrounded by single or double quotes and therefore look like strings in other languages (e.g. `"0xdeadbeef"`).

Multiple Definitions of KCONFIG Options

Additional Information:

KCONFIG allows developers to to define configuration options – often also called features – multiple times. An option is defined multiple times if multiple options have the same symbol. The type of those definitions must not conflict and only one input prompt per definition is allowed. KCONFIG does not combine conditions of multiple definitions. These definitions are independent of each other, except they influence the value of same symbol and only the last settled value is utilised for the generation of the resulting `.config`. They can be defined

[2] Which prefix is added can be controlled by users by changing the environment variable `CONFIG_`. If this variable has a value, this value will be used as a prefix.

in different files and may appear in different places in the tree structure of menus if their dependencies are satisfied and they are visible. Users of configuration tools might be confused when they search for options that are defined multiple times. KCONFIG does not show a clear distinction between multiple definitions in the search function and users may not be aware that multiple definitions are more alternative definitions than a definition of additional conditions, thus the search result for such options looks broken to them.

2.5.2 Menus

A `menu` explicitly specifies the structure of a number of configuration entries. The syntax for such a structure group is shown in this example:

```
menu "<categorial description>"
    <menu options>
    <menu entries>
endmenu
```

All entries within the menu block become subentries of this menu and inherit all its dependencies. Listing 2.7 shows a `menu` structure in the lines 40–53, while the resulting `menu` structure is displayed in Listing 2.1 in line 8 to 11.

2.5.3 Choices

Choices allow developers to declare groups of options where the number of concurrently active choice values can be limited depending on the type of the choice and an additional option `optional`. Choices can only have two types, either boolean or tristate. If no type is chosen and only a prompt attribute is present, KCONFIG will infer the type of the choice on the base of the type of the choice values. If there are tristate choice values, the choice will be tristate, otherwise the choice is boolean. The correct syntax of choice declarations is displayed in the following example:

```
choice [symbol]
    tristate, bool or prompt <text>
    <optional or other attributes>
    <multiple choice values>
endchoice
```

A boolean choice describes a *exclusive or* (XOR) relationship between the choice values. Exactly one entry can be activated at the same time. An example for a boolean choice can be seen in Listing 2.7 in the lines 68–84 with the corresponding illustration of the structure in the lines 15–18 in Listing 2.1.

If the type of the choice is tristate though, users will get a prompt to choose the value of the choice when they configure the kernel. When they choose “y”, the choice will behave exactly like a normal boolean choice and only exactly one value can be selected. If they choose “m” the choice will allow users to set 1..*n* tristate choice values to “m”, boolean choice values will not be shown to users in that case. This semantic is reasonable for LINUX, since it will ensure only one driver will be compiled into the kernel or multiple drivers are available but will only be loaded at runtime if they are needed. Listing 2.7 displays an example of a `tristate` choice from line 86 to 105.

There is a special option for choices which is marked by the `optional` keyword. With this keyword it is possible to leave a choice *disabled*. This changes the behaviour of a boolean choice to a mutually exclusive 0..1 choice and the range of possible activated module choice values to 0..*n* [Ber+12]. However, there are very few `optional` choices in the LINUX kernel.

```
1  choice
2      tristate "USB Gadget Drivers"
3  [...]
4
5  config USB_CONFIGFS
6      tristate "USB functions configurable through configfs"
7
8  config USB_CONFIGFS_SERIAL
9      bool "Generic serial bulk in/out"
10     depends on USB_CONFIGFS
11
12 config USB_CONFIGFS_ACM
13     bool "Abstract Control Model (CDC ACM)"
14     depends on USB_CONFIGFS
15
16 [...]
17 endchoice
```

Listing 2.8 – An abbreviated example for a choice where not all config declarations in a choice block are choice values. `USB_CONFIGFS` is the only choice value, while `USB_CONFIGFS_SERIAL` and `USB_CONFIGFS_ACM` are only submenus of `USB_CONFIGFS`.

Additional Information:

Developers might be surprised by the fact, that not all config declarations in a choice block are eventually choice values and thus presented to users. There is one choice with a lot of choice values in LINUX where this happens, as shown in the abbreviated Listing 2.8. `USB_CONFIGFS` is the only displayed valid choice value. The other config entries `USB_CONFIGFS_SERIAL` and `USB_CONFIGFS_ACM` have a dependency on `USB_CONFIGFS`. As I explain in Section 2.5.4.1, KCONFIG analyses the dependencies and tries to create a submenu structure on the basis of these dependencies. This leads KCONFIG to see `USB_CONFIGFS_SERIAL` and `USB_CONFIGFS_ACM` as submenus of `USB_CONFIGFS` and not as choice values as developers might expect. The

definition of both submenus would be semantically equivalent to Listing 2.8 if they were defined after the `endchoice` statement.

2.5.4 Menu Attributes

Menu entries can have a number of attributes. These attributes can be used to establish the type of a config option, declare dependencies, input prompts, help texts, default values or limit the interval of possible values [Doc]. Many of those attributes accept an optional `if <expr>` enquiry. Attributes with these enquiries are only active when the expression is satisfied, according to the definition I provide in Subsection 2.5.4.1.

There are five possible types for config options: `bool`, `tristate`, `string`, `int`, `hex`. The `boolean` type has two possible values; "y" (yes) and "n" (no). A `tristate` symbol is similar to a boolean symbol, except it has a third state; "m" (module). The value "y" indicates that code will be compiled into the kernel image and the value "m" indicates code will be built as LKM.

When the build system derives the configuration to the MAKE and CPP representations (as described in Section 2.1), it uses *two* symbols to represent the *three* states of a `tristate` symbol as displayed in Table 2.1. For example, when users assign the value "y" to the `tristate` configuration symbol `X`, the transformation process will define the macro `CONFIG_X`. When the assigned value is "m", the macro `CONFIG_X_MODULE` will be defined and none of both macros will be defined for the value "n".

X	CONFIG_X	CONFIG_X_MODULE
y	1	0
m	0	1
n	0	0

Table 2.1 – An example how the build system represents tristate values. The option `X` has three possible values: "y", "m" and "n", which are represented by defining either `CONFIG_X` or `CONFIG_X_MODULE` or none of them.

Additional Information:

Internally, KCONFIG only has two basic types for options; `tristate` and `string`. All other types are based on them. The `boolean` type is based on the `tristate` type and the numeric types are based on `string`.

The type attribute also accepts an input prompt definition with the following syntax:

```
<type> "<prompt text>"
```

This definition is semantically equivalent to the following definition with the `prompt` keyword:

```
<type>
prompt "<prompt text>"
```

An input prompt defines the displayed text in configuration tools. Every option can have at most one input prompt. It is possible to add a condition that has to be satisfied before the prompt of this option is exposed to the user:

```
prompt "<prompt text>" [ if <expr> ]
```

An option can still be active through default values, if it is invisible; otherwise, the default value will be presented to users and they can accept or override it.

A default attribute (Syntax: **`default <expr1> [if <expr2>]`**) can be used to set a default value to **`<expr1>`** for an option. Only the first visible default attribute is active if there are multiple defaults defined, even if the config option is defined multiple times and default attributes are spread over these definitions. This means, defaults can be overridden by earlier definitions. The visibility can be limited with conditions in **`<expr2>`**. There is a shorthand notation for the declaration of `bool` or `tristate` options without a prompt and a default value with the syntax:

```
def_{bool, tristate} <expr1> [ if <expr2> ]
```

The condition in **`<expr2>`** is applied to the declared default value **`<expr1>`**.

2.5.4.1 Dependencies

The dependency attribute can be added with a **`depends on <expr>`** statement. If this attribute is given multiple times for an option, KCONFIG joins the expressions with the "&&" operator.

The expression is applied to all other attributes of a configuration entry, which accept an **`if`**-statement in their definition. This means the declaration:

```
bool "foo" if <expr>
default y if <expr>
```

is semantically equivalent to:

```
bool "foo"
default y
depends on <expr>
```

KCONFIG tries to derive a menu structure on the basis of these dependencies. If option B depends on option A, it will try to show B as a submenu of A. This process seems easy for this trivial example but will get complicated if more configuration options are involved. These submenus can be seen as a tree structure as seen in the following example:

```
+- A
|  +- B
+- Kernel Hacking
|  +- Kernel debugging
|  +- Memory debugging
|    +- Debug VM
+- ...
```

Dependencies enforce an upper limit on the symbol which defines them. This means the input range for tristate symbols is reduced. An example for this case are the declarations in Listing 2.9. When users choose A to be built as module, B can be at most module or be disabled.

```
1 config A
2     tristate "A"
3
4 config B
5     tristate "B"
6     depends on A
```

Listing 2.9 – Example for the implications of an upper limit enforced by dependencies.

However, if B would have the type `bool` instead of `tristate` and users would choose A to be built as module, KCONFIG will round this limit up to "y" and the dependency for B would be satisfied. Users then could choose between either "y" or "n" for B.

Tristate options can have three possible values: "n", "m" or "y". To represent these values in calculations, KCONFIG uses the numbers 0, 1 and 2. Configuration options become visible when its dependency evaluates to "m" or "y".

Dependency Expressions

A dependency expression (previously described as **<expr>**) is *true* when all of its sub expressions evaluate to either "y" or "m". Expressions can be combined and linked according to the syntax rules defined in the KCONFIG documentation [Doc]:

$$\texttt{<expr>} ::= \begin{cases} \texttt{<symbol>} & (1) \\ \texttt{<symbol>} = \texttt{<symbol>} & (2) \\ \texttt{<symbol>} \; \texttt{!=} \; \texttt{<symbol>} & (3) \\ \texttt{(} \; \texttt{<expr>} \; \texttt{)} & (4) \\ \texttt{!} \; \texttt{<expr>} & (5) \\ \texttt{<expr1>} \; \&\& \; \texttt{<expr2>} & (6) \\ \texttt{<expr1>} \; || \; \texttt{<expr2>} & (7) \end{cases}$$

(1) Convert the symbol to an expression. Returns the value for boolean and tristate symbols and "n" for other symbol types.

(2) If the values of both symbols are equal, it returns "y", otherwise "n"

(3) If the values of both symbols are equal, it returns "n", otherwise "y"

(4) Returns the value of the expression. Used to override precedence.

(5) Returns the result of (2 - **<expr>**)

(6) Returns the result of min(**<expr1>**, **<expr2>**)

(7) Returns the result of max(**<expr1>**, **<expr2>**)

When a value-like symbol (`string`, `hex`, `int`) is used directly in an expression, this expression always evaluates to "n" (e.g. **`depends on Z`** when Z has the type string) as a direct result of (1). This means, value-like symbols can only be used with comparison operators.

Less obvious is the behaviour of the comparison operators (2) and (3) if one of the symbols is not defined. As a consequence of definition (1), users could expect an undefined symbol to always evaluate to "n" and a comparison of an undefined symbol and "n" to be always true. However, this is not the case. KCONFIG sees the type of undefined symbols as *unknown* and the value of these symbols as *indeterminate*. Consequently is a comparison with an undefined symbol in case (2) always evaluated to false. For the non-equality check (3) the opposite happens: a comparison with an undefined symbol is always true.

Additional Information:

Tristate logic is in two cases unapparent to developers:

- The first unobvious case occurs for the negation operator (5) in combination with a `tristates` symbol `B` which is set to "m" (internally interpreted as 1). This operator returns the result of `(2 - <expr>)` where **<expr>** is the value of the tristate symbol. This means, a `tristate` option `A` with a dependency on `!B`, has its input range limited to [0,1] or "n" and "m" when `B` is set to "m". When `A` has the type `bool`, the input range is limited in the same way but KCONFIG will round the 1 up to 2 and the allowed inputs are "n" and "y" or {0,2}.
- When an option `A` has the dependency **`depends on RFKILL || !RFKILL`**, the dependency is always true and does not impose an upper limit on `A` if `A` has the type `bool`. When `RFKILL` and `A` have the type `tristate`, this dependency does not impose an upper limit on `A` when `RFKILL` has the values "n" or "y", but limits the highest possible value

of A to “m” when RFKILL has the value “m”. In contrast to these *two* previous cases is a dependency on **A && !A** always false when A is a bool symbol and equivalent to a comparison with “m” (**A="m"**) when A has the type tristate.

Implementation of Value-like Symbol Comparisons

As I have explained in Section 2.5.1, there are two types of symbols: constant and non-constant ones. However, KCONFIG and its users in LINUX do not use these symbols in a disciplined way in comparisons (2) and (3) in dependencies. Constant symbols are easy to recognise due to their string-like notation (e.g. **"0xdeadbeef"**), but developers often use them without quotes (e.g. **0xdeadbeef**) in comparisons. KCONFIG recognises the second usage as a non-constant symbol and will set the internal type to **unknown** since it is not defined anywhere. When such a symbol is used in a comparison with a symbol with int or hex type, KCONFIG tries to parse the name of the **unknown** symbol accordingly to the type of the symbol on the other side of the comparison. If the name of that symbol does not match a certain pattern for the type, it will be rejected and the comparison is always false or true, as I have explained before.

Additional Information:

Names for hex types must start with “0x” or “0X” and all following characters must match the regular expression (RegExp) [0-9a-fA-F]. Names for int symbols must not start with a zero - a zero would indicate an octal number - and might start with a minus character and the following characters must be in the range [0-9].

A Comparison between a non-constant string symbol and something else is only valid if the other side of the comparison has either the string type itself or is a constant symbol. The comparison is then done char-wise.

2.5.4.2 Select Statements

The **select**-statement (Syntax: **select <symbol> [if <expr>]**) realises reverse dependencies for boolean and tristate options. In contrast to the previously mentioned dependency attribute, reverse dependencies enforce a lower limit of another symbol (notice the order I mentioned in Section 2.5.4.1: “n”, “m” and “y” are internally represented as 0, 1 and 2 in KCONFIG). The value of the current symbol is used as a minimal value for the selected **<symbol>**. If **<symbol>** is selected multiple times, the largest selection is used as a limit [Doc]. This means if A is a tristate option which selects B, B at least has to have the value “m” if A is set to “m”.

The KCONFIG documentation defines several rules for the usage of these statements. However, KCONFIG does not enforce these rules and they are often ignored or abused. According to these rules, select statements should only be used on non-visible symbols, which means symbols without a prompt in any definition, and on symbols with no dependencies.

Nevertheless, select statements in KCONFIG will force a symbol to a value, even if their dependencies are not satisfied. The only case where this is not true is a select on a choice value. Select

statements on symbols that are choice values are silently ignored. KCONFIG will print a warning to the console if a selected symbol has unmet dependencies.

2.5.4.3 Ranges for `int` or `hex` Options

Configuration options with the type `int` or `hex` can have a special attribute to limit the valid range of an option to a closed interval between two symbols:

```
range <symbol1> <symbol2> [ if <expr> ]
```

2.5.4.4 Other Option Attributes

There is a number of attributes which are only used a few times due to their special purpose. All of these attributes begin with **`option`** followed by one of four valid options.

- The **`modules`** option declares the symbol to be used as `MODULES` symbol, which enables the support for the third state of tristate symbols. KCONFIG allows at most one menu entry to have this modules option. If there is no modules option, or the one available is set to "n", KCONFIG will treat all tristate symbols as boolean symbols.
- With **`env=<value>`** developers can import environment variables into a menu entry with the type string. This attribute behaves like a default if the symbol has a prompt, except the value is read from the environment. The behaviour of this option is undefined if it is mixed with default values.
- **`defconfig_list`** marks the option to have a list of string default values with a path to default configs, which can be used as a default when there is no `.config`.
- Another less common option is **`allnoconfig_y`** which can be used to indicate to KCONFIG that it has to set the value this entry to "y" when **`allnoconfig`** is run.

Another seldomly used attribute is used for the restriction of the visibility of `menu` entries (see Subsection 2.5.2). Since `menus` do not have an explicit prompt, apart from the description string, it is not possible to restrict the visibility of that menu through a condition for that prompt. However, there is a special attribute for menus to limit the visibility: **`visible if <expr>`**. If the expression is false, the menu and its submenu entries will not be visible to users.

2.5.5 Structure in KCONFIG

To declare the relevant menu entries in the same (sub)directories where source files or Makefiles are placed, KCONFIG has the **`source "<path>"`** statement to include other KCONFIG files. This statement leads the KCONFIG parser to read the KCONFIG file refrenced by <path> and will always be evaluated. Each architecture has a root KCONFIG file in its subdirectory in `arch/`[3], which includes other KCONFIG files through source statements. This means each architecture has its own KCONFIG

[3] Except for the User Mode Linux (UML) architecture, whose root kconfig file is located in `arch/x86/um/` in current versions.

tree of options. Developers can re-use KCONFIG declarations in different architectures and do not have to re-declare them for every architecture.

2.5.5.1 If Statements

If a developer wants to add the same expression to the dependencies of a number of consecutive config options, this can be done with an if-block, as shown in the following example:

```
if <expr>
    <config entries>
endif
```

In contrast to if-statements in imperative languages, an if-statement in KCONFIG does not mean the contents of the if-block will not be evaluated if the expression evaluates to false.

Another construct **showif <expr>**, similiar to **if <expr>**, is currently discussed on the mail-inglist[4]. Contrary to if-statements, the expression will be added to the condition of the prompt and not to the dependencies.

With the proposed patch, developers try to solve a problem of the automatic creation of submenus. When a consecutive group of options has the same condition for prompts, e.g. **if EXPERT**, KCONFIG will try to add all options of this group as a submenu of EXPERT. If one of these options inside the group, not at the borders, loses this condition or another symbol without the same prompt condition is inserted into this group, KCONFIG breaks the submenu at this position and the options after the offending one, will be displayed in the parent menu of EXPERT ("General Setup") and thus might be overlooked by users.

2.5.6 Common Idioms in KCONFIG

The KCONFIG documentation[Doc] provides two hints how to implement common idioms.

The first idiom is for options that are defined for some architectures but not all of them. This idiom is based on the idea to introduce an intermediate option, or bridge option, that is selected by the architecture that supports the option in question. These intermediate options usually are named after the basic option, but usually add a prefix HAVE_, MIGHT_HAVE_ or HAS_.

Listing 2.10 shows an example of this idiom, where HAVE_GENERIC_IOMAP is an option that provides a bridge between the architectures main-option, X86, and the actual option GENERIC_-IOMAP. These bridge options help developers to easily learn which architectures support which features without a need to search where X86 is mentioned in the dependencies.

Another way to implement this functionality would be to add the architectures main-options to the dependencies, but if an option is implemented by many architectures, the dependencies would be cluttered an it would be increasingly difficult to parse the relevant bits.

[4] https://lkml.org/lkml/2015/5/14/450

```
config X86
    def_bool y
    select HAVE_GENERIC_IOMAP


config HAVE_GENERIC_IOMAP
    bool

config GENERIC_IOMAP
    bool "foo"
    depends on HAVE_GENERIC_IOMAP && FOO
```

Listing 2.10 – An example of the usage of a KCONFIG-hint

Another idiom is the restriction of a component to be built as module only. This can be done by qualifying the symbol declaration with a `depends on m` statement, which limits the value of this symbol to module "m" or disabled "n". Due to the transitivity of dependencies, all options that have a symbol with a `depends on m` attribute in their dependencies, will also be at most be defined as module "m".

2.6 Related Work

In this section I describe related approaches of other researchers. The research of previous VAMOS/CADOS research is the most relevant for this thesis, therefore I start with a overview of the research of Sincero [Sin13] and Tartler [Tar13]. The ontological approach to identify dead and false optional features by Rincón et al. [Rin+14] is discussed at the end of this section.

2.6.1 Previous Work in the VAMOS/CADOS Projects

In 2007 Sincero et al. [Sin+07] investigated if the LINUX kernel is a Software Product Line (SPL), where he describes a product line as a set of software components that can be assembled together in order to deliver different products (kernel images in the case of LINUX) in a specific domain. They state that the scientific SPL guidelines aim at several goals like automatic product generation (see Section 2.1), high configurability (14.541 features in Linux/`v4.0`), code re-use (subsystems are re-used for different architectures), flexible architecture, no overhead for unrequested features (CPP and whole files may be excluded), etc. While LINUX accomplishes most of these goals, they *dismiss* the kernel as a SPL, because the guidelines also aim at optimisations of the development process through domain engineering, which are not met by LINUX. These optimisations should improve the time-to-market, the product quality and mass customisation. Sincero et al. [Sin+07] state that development of new features in LINUX is not a controlled planing process and the evolution of the code base is based on the work of several thousands of volunteers and a few paid engineers who peer review and test code changes, implement new features or alter kernel-wide interfaces. They

later conclude that the LINUX kernel fits the requirements for a SPL only partially and it can only be *partially* considered a SPL by the definition of Northrop and Clements [NC01], which Sincero used.

Sincero et al. [Sin+10] later analysed the l_2 layer (see Figure 2.2) of the implementation space variability in LINUX to identify variability related bugs. He extracted presence conditions[5] for CPP blocks in source files and transformed them to propositional formulas and asked a checker function, if the extracted conditions are satisfiable. Listing 2.11 shows an example for the defects he identified. The first block B0 requires `CONFIG_A` to be enabled and the second block B1 requires `CONFIG_A` to be disabled. This is a contradiction and block B1 is therefore dead. As a side effect of block B1 being dead, the else block B2 becomes undead and cannot be deselected. To check if arbitrary formulas are

```
1 #ifdef CONFIG_A
2 // B0
3 #ifndef CONFIG_A
4 // B1   dead
5 #else
6 // B2   undead
7 #endif
8 #endif
```

Listing 2.11 – An example of a code defect. Block B0 requires `CONFIG_A` to be set and B1 requires `CONFIG_A` to be disabled, which is a contradiction and B1 is dead and cannot be enabled. Since B1 is dead, B2 is undead and cannot be disabled.

satisfiable is a (boolean) satisfiability problem (SAT) problem, which is NP-complete. The practical implementation of the checker function is realised with a SAT-checker. He named these defects `code` defects, since the only relevant conditions are extracted from the code layer of variability.

As an extension of the work done by Sincero, Tartler et al. [Tar+11b] analysed the configuration space (layer l_0) of LINUX. He extracted the variability information from KCONFIG and transformed them to a feature model. The combination of code conditions extracted by Sincero et al. [Sin+10] (l_2) and the *relevant* conditions from KCONFIG (l_0) lead to the identification of two additional defect types: *logical* and *referential*. The tool to identify the defects previously described as `code` defects by Sincero, the *logical* and *referential* defects by Tartler et al. [Tar+11b] is called UNDERTAKER[6].

An example for a logical defect is displayed in Listing 2.12. To compile block B1 it is necessary to enable both options `A` and `B`. If the conditions of KCONFIG are considered and `B` has a dependency on `!A`, both options cannot be enabled at the same time and block B1 is dead.

The third identified defect type is *referential*. This type of defect happens because options are used that are never defined in KCONFIG and thus can never be enabled (example in Listing 2.13).

After the analyses of Sincero et al. [Sin+10] and Tartler et al. [Tar+11b] where they focused on the layers l_2 and l_0 (see Figure 2.2) of variability, Dietrich et al. [Die+12] evaluated the KBUILD layer (l_1). Figure 2.3 shows how many KCONFIG symbols are used in which build phase. Dietrich

[5] The CPP block is active if these conditions are satisfied. If blocks are nested, this condition includes the merged conditions of the block itself and of *parent* blocks.

[6] `https://undertaker.cs.fau.de`

```
1 #ifdef CONFIG_A
2 // B0
3 #ifdef CONFIG_B
4 // B1   dead
5 #endif
6 #endif
```

Listing 2.12 – An example of a logical defect. Block B1 requires `CONFIG_A` and `CONFIG_B` to be enabled. When `B` depends on `!A` in KCONFIG, the UNDERTAKER tool finds a contradiction and describes the defect as KCONFIG or logical defect.

```
1 #ifdef CONFIG_CPU_HOTPLUG
2 // B0   dead
3 #endif
```

Listing 2.13 – An example of a *referential* defect. Block B1 requires `CONFIG_CPU_HOTPLUG` to be enabled. However, `CPU_HOTPLUG` is never defined in KCONFIG and thus can never be enabled and the block is dead. Developers accidentally interchanged the option, the correct one would have been `HOTPLUG_CPU`.

et al. [Die+12] noticed a huge fraction of KCONFIG symbols is used in KBUILD and a large portion of these symbols is only used in KBUILD. Berger et al. [Ber+10] proposed a (fuzzy) parser for the extraction of KBUILD constraints but Dietrich et al. [Die+12] did not consider this parser robust enough over different versions of LINUX, since some Makefiles had to be altered, and implemented a probing based extraction of these constraints to increase the accuracy of the previous explained defect analyses by combining the constraints of the layers l_0 to l_2.

The previous examples always combined the layer l_2 with other layers to search for defect CPP blocks in source files. However, with the extracted presence conditions for whole source files, the question whether there are dead or even undead source files suggests itself. To tackle that problem, Tartler introduced a virtual block around the whole file to represent the source file. This virtual block gets the file precondition as block condition and the dead analysis is able to check if a file is dead but not if a file is undead. The dead analysis combines the layers l_1 and l_0.

However, the probing based extraction proved to be expensive [Rup15]. While an extraction of the Linux/`v2.6.36` constraints took around 30 minutes on a Core i5-4590 processor with 3.3 GHz, 16 GiB RAM, the extraction of the Linux/`v3.16` constraints took over 3 hours on the same machine for a *single* architecture. To tackle these costs, Ruprecht [Rup15] implemented an highly accurate parser for KBUILD constraints which extracts the same conditions in just over a second. He also introduced a new defect type (KBUILD logical) to distinguish between an ordinary KCONFIG logical defect and a defect that is caused by the addition of KBUILD constraints. This faster approach allowed the inclusion of file preconditions in the analysis in UNDERTAKER–CHECKPATCH.

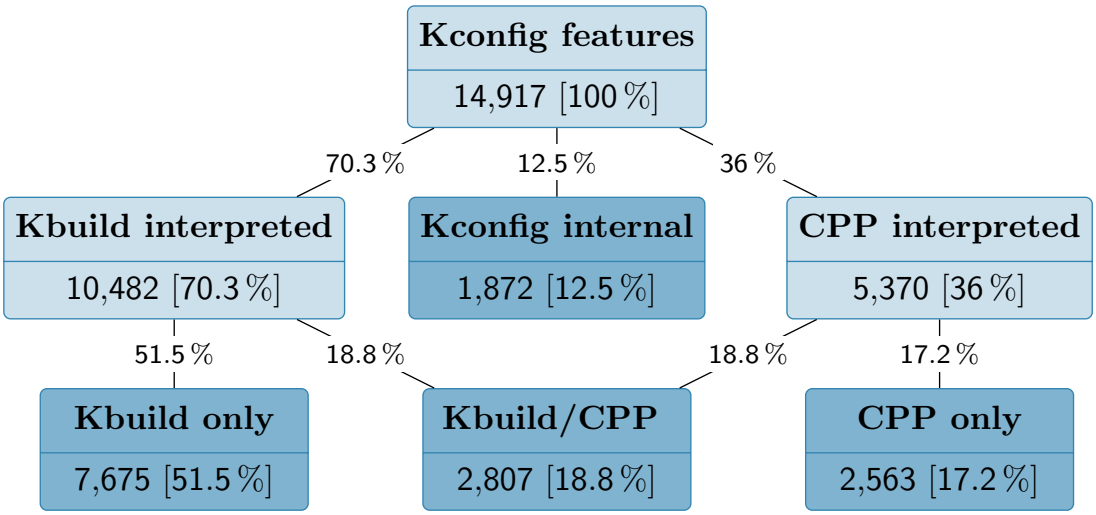

Figure 2.3 – The distribution of the uses of KCONFIG features in Linux/`v4.1-rc7` in the different build phases. The percentages on the edges are calculated relative to the total amount of KCONFIG features.

UNDERTAKER–CHECKPATCH

The analyses I previously presented investigated a number of files or the whole source tree of LINUX for variability defects. This procedure does not fit to the workflow of LINUX developers where code improvements are done incrementally by sending patches to subsystem maintainers. To simplify the usability of the UNDERTAKER tool, Rothberg [Rot14] implemented a small wrapper tool, that parses patches sent to maintainers and employs the UNDERTAKER toolchain to check if a patch introduces a new variability defect, fixes an old one or leaves an old defect unchanged. This is achieved by comparing the state *before* the patch is applied to the state *after* the patch is applied and checking for variability defects in each step. In addition to this functionality, UNDERTAKER–CHECKPATCH resolves *referential* problems in order to provide hints for developers about the origins of newly introduced defects.

Additionally, this tools enabled Ruprecht [Rup15] to perform the analyses done by the UNDERTAKER toolchain on a daily basis, where all new patches in a `linux-next` version are analysed. Since this tool integrates well with the workflow of LINUX developers, it is reasonable to expand this tool with new checks to be able to promptly warn developers about defects they have introduced.

Configuration Coverage

Configuration coverage is one of the major problems for static analysis tools or a reliable testing environment of variable software. Research for configuration coverage is driven by the question how to achieve thorough code coverage, while expenses and resource costs are reduced [WP01; Tar+11a].

With the rising number of commits introduced into the LINUX kernel, it is considered increasingly difficult to at least compile test each part of the code. Consider Listing 2.14. To compile both

```
1 #ifdef CONFIG_A
2 // code
3 #else
4 // more code
5 #endif
```

Listing 2.14 – An example why one configuration does not suffice to achieve a high code coverage.

blocks, developers need two configurations, one with `CONFIG_A` enabled and one where that option is disabled. However, maintainers often test their code only with *one* `allyesconfig`[7] or *one* `allmodconfig`[8], which both could not cover the simple example above.

To attain viable metrics for configuration coverage for linux source files, Tartler [Tar13] introduced the following theorem:

Theorem 2.6.1. *The configuration coverage of a given configuration is the fraction of the number of selected blocks divided by the total number of available blocks.*

$$\mathcal{CC} := \frac{\text{selected blocks}}{\text{available blocks}} \tag{I}$$

Earlier I have described that there are a number of blocks that can never be selected and others that can never be de-selected. These blocks have to be singled out of the configuration coverage because undead blocks are covered by every configuration and dead blocks cannot be covered by any configuration. As a consequence, Tartler [Tar13] introduced an additional metric for a *normalised* configuration coverage:

$$\mathcal{CC}_N := \frac{\text{selected blocks} - \text{undead blocks}}{\text{available blocks} - \text{undead blocks} - \text{dead blocks}} \tag{II}$$

This means, that the possible code coverage changes when more defects are recognised. Therefore, I do a comparative analysis of the configuration coverage metrics $\mathcal{CC}$ and $\mathcal{CC}_N$ in LINUX in Subsection 5.1.1.1.

Tartler [Tar13] states, a holistic variability model [...] is useful to improve the effectiveness of static analysis tools. With the extracted variability information taken from the layers l_0 and l_1, the UNDERTAKER toolchain is able to create a minimal set of maximising configurations for single files to increase the overall configuration coverage of tests or static analysis tools. These configurations are called *partial configurations*.

In their Configuration Coverage (CC) paper, Tartler et al. [Tar+12b] discuss two different algorithms to maximise the coverage of a set of configurations. Their *greedy* approach is more

[7] KCONFIG tries to set all options defined to "y". If it notices conflicting dependencies when trying to enable an option, this option is simply not enabled. The result of `make allyesconfig` therefore depends on the traversion order of those options.
[8] KCONFIG tries to set all options defined to "m". The same limitation apply as for `allyesconfig`

expensive to calculate but delivers better results for a lower number of configurations then the *naïve* approach. The algorithms described in this paper implement *statement coverage*[9]; higher coverage criteria such as *decision coverage* or *path coverage* are deemed as too expensive, although the UNDERTAKER toolchain has been expanded with a *decision coverage* algorithm later.

Later Tartler et al. [Tar+14] used their previous work on CC to implement a driver for configurability-aware static analysis tools called VAMPYR and used this tool to find additional compilation errors and warnings.

2.6.2 Analysing Defects in Feature Models

Rincón et al. [Rin+14] introduces the concept of an "ontological rule-based approach for analysing dead and false optional features in feature models". Their main contribution is, apart from the identification of dead and false optional features, the explanation of the cause of the defect in natural language. For the defect identification, they first transform the Feature Model (FM) to a *Feature Model Ontology*, which is a enriched version of the FM with the relevant information for the dead or false optional analysis. They define an ontology as a "[...] formal explicit specification for a shared conceptualisation" [Rin+14].

After the transformation step they use a set of rules to identify the defects and print a description of the defect according to those rules in the last step. Their analysis scales for up to 150 features. For 100 features this analysis takes around 5sec to finish and for 150 features their analysis already takes 2 minutes of analysis time. This exponential growth proves this approach impracticable for highly configurable systems like LINUX where the six year old version Linux/`v2.6.31` already has 8988 features.

Another limitation is that they use a rather formal feature model which is difficult to get with the complex semantics of the three valued logic through tristates and the select semantics (see Section 2.5 for details).

2.7 Summary

In this chapter I have shown how the build system implements the variability in LINUX, followed by a short description of all steps (KCONFIG, KBUILD, CPP) of the build sequence and then described every language aspect and behaviour of KCONFIG in detail. After that, I continued with the explanation of the approaches of other researchers. I started with a discussion if LINUX can be considered a SPL and continued with a description about the work of Tartler [Tar13] and Sincero [Sin13], followed by an explanation about how they have identified many defects (directly in the code, with the help of a model of LINUX's variability, with KBUILD constraints and how they find blocks that are defect because of missing symbols in the presence condition) in the variability in LINUX and introduced the tools they provide to support static analysis tools to maximise the *configuration coverage* but also the metrics they define to understand this *configuration coverage*. I also presented a short introduction

[9] Visit `https://en.wikipedia.org/wiki/Code_coverage` for details.

into the new UNDERTAKER–CHECKPATCH tool that employs the tools created by Tartler and Sincero to analyse patches for variability defects.

Furthermore, I produced a illustration of the analyses of feature models done by Rincón et al. [Rin+14], which defects they identified and how they tackled the identification.

2.8 Problem Statement

Previous approaches by Sincero [Sin13] and Tartler [Tar13] have analysed the variability in LINUX. They focused on the variability in source files and expanded later the presence conditions of CPP blocks (φ_{CPP}, conditions from layer l_2) with information extracted from KBUILD (φ_{KBUILD} from l_1 conditions) and KCONFIG (φ_{KCONFIG} from l_0 conditions). With this combined variability information they achieved a holistic view of the variability in LINUX:

$$\varphi_{\text{LINUX}} = \varphi_{\text{CPP}} + \varphi_{\text{KBUILD}} + \varphi_{\text{KCONFIG}}$$

This view enabled them to identify a number of defects as shown in Section 2.6.1. However, their research was limited to variability defects that originate in code conditions (φ_{CPP}), in a combination of the code conditions and preconditions extracted from KCONFIG ($\varphi_{\text{CPP}} + \varphi_{\text{KCONFIG}}$), in a combination of the former in addition with constraints from the build system KBUILD ($\varphi_{\text{CPP}} + \varphi_{\text{KCONFIG}} + \varphi_{\text{KBUILD}}$) and, on a per file base, if there are defects in file preconditions from KBUILD in combination with conditions from KCONFIG ($\varphi_{\text{KBUILD}} + \varphi_{\text{KCONFIG}}$). Still, they did not investigate if there are defects in the modelled configurability φ_{KCONFIG} in LINUX.

The configurability in LINUX is organised in a hierarchy of layers (see Figure 2.2) where each layer dominates the layers beneath itself. This means changes on an upper layer have an impact on lower layers. With that in mind, the configuration space (l_0) is the most important layer in the build process of LINUX, because it dominates all other layers and defects on this level have an impact on all later build phases. It is therefore surprising that defects in the *configuration space* are not well researched. With this thesis, I fill this gap and provide tools to analyse the configuration space in LINUX. I use and expand the tools created by Sincero [Sin13] and Tartler [Tar13] as a basis for my work and analyse and improve their reverse-engineered model of the FM of LINUX. Finally, I will answer the following, up to now unanswered, questions:

- Are there options in KCONFIG that can never be chosen but cause defects in later build phases? (dead options)
- Are there options that are always enabled but appear configurable? (undead or false optional [Rin+14])
- Are there other defects that originate in KCONFIG that influence the variability in later stages?

In the next chapter I will focus on a detailed description of the potential problems in LINUX and how I identify the problems mentioned in the introduction of this chapter.

3 DESIGN

The variability in LINUX is implemented in several interconnected phases. The configurability is implemented in a hierarchy of KCONFIG files and the behaviour of the KBUILD phase is implemented in a hierarchy of Makefiles and KBUILD files, where *coarse grained* decisions are made if whole files are compiled or not. The *fine grained* implementation of the variability in source files is implemented in CPP blocks that can be nested. All these steps have one thing in common: when the files are too large or the implementation is spread across different files, it is hard to keep an overview and avoid mistakes. Various combinations of these phases were already analysed by other researchers, but the phase that influences all other phases in the build system was up to now not properly researched. In this chapter, I describe several defect types that I have identified and how I tackle their identification.

The reason Sincero et al. [Sin+07] only partially considered LINUX as a SPL is the fact, that it is not developed according to the SPL guidelines with a controlled planning and domain engineering process (as discussed in Section 2.6.1). This lack of planning is compensated by very large manpower [Sin+07] and even small changes and new features are peer-reviewed. Therefore, KCONFIG also does not need and provide a formal specification for their configurability implementation; KCONFIG is a pragmatic approach to solve the configurability problem, it is fast and battle tested through large code base of LINUX. Because of this lack of formal specification of the KCONFIG language, it was necessary for Tartler [Tar13] to reverse-engineer a model of the FM by studying the implementation. His model is created in two steps:

❶ dump all relevant information (symbolname, type, dependencies, prompt, selects, defaults) for each option to an intermediate file

❷ use the parsed information from the intermediate file to generate a model with propositional logic as intermediate language [Tar13]

The first step ❶ dumps a serialised version of the KCONFIG parse tree to an intermediate file. Step ❷ parses and aggregates the information from the intermediate file to create a formula – in propositional logic – for each symbol. The generated model of the FM is therefore a set of propositional implications, one or two implications I_n per symbol n in the *configuration space* $\Phi_{\text{configuration}}$ [Tar13]. As described in Section 2.5.4 on Page 18, the assigned values of `tristate` symbols are represented by *two* macros in configuration files, which can be later referenced in the build system or in CPP statements. Due to the definition of this mapping it is not possible to enable both macros at the same time and

due to the peculiarities of the `tristate` logic, both macros have different presence conditions. As a consequence of these facts, `tristate` symbols have *two* implications for their presence, which have to be connected together with a logical disjunction for the full variability model. The full variability model can be described as the conjunction:

$$\varphi_{\text{KCONFIG}} := \bigwedge_{n} I_{n_1} \vee I_{n_2} \qquad \forall n \in \Phi_{\text{configuration}}$$

However, this variability model does not contain information like the type of a symbol, if a symbol has a prompt, which symbols are selected under which conditions, information about default values or where symbols are defined, but only the aggregated presence conditions for a symbol. This means that the intermediate model can still provide a valuable contribution to analyses of the configuration space as I explain in Section 3.2 and is also used by the UNDERTAKER toolchain to retrieve type information.

Nevertheless, the extraction process created by Tartler [Tar13] is not exact. He considered some language aspects like "symbol visibility" and "default values" technically harder to implement and left them unimplemented. The impact of this approximated rather than exact model is discussed in Section 5.2.4.

In order to get a *exact* representation of the variability in LINUX, I could have extracted a snapshot of KCONFIG by myself, but that would have gone beyond the scope of this thesis. Though, the UNDERTAKER toolchain offers another image of LINUX's FM s. Hackner [Hac13] implemented an extractor that directly transfers the declared variability to a model in `DIMACS` cnf format[10], the format that SAT checkers use. This model does not offer the possibility to employ the slicing algorithm (which I describe in Section 3.3) and, due to the nature of the implementation of the extractor, the number of clauses ascends exponentially with every added feature. Both arguments lead me to believe that a consistency analysis with this model would be too slow and would not scale. Furthermore, this model does not contain information about type, presence of a prompt, which symbols are selected under which conditions or any information about default values. Therefore, I would need to either implement a parser of KCONFIG to get these information or use the intermediate model of the extraction done by Tartler.

While architectures in LINUX share common code in the subsystems, they have specific code in their subfolder in `arch/` that no other architecture uses. It is no surprise that different architectures do or do not provide support for different features (e.g. MMU, KVM support, ...). To model this behaviour correctly, each architecture has its own structure of KCONFIG files that later include the configuration files in the subsystems and has the possibility to provide alternative definitions for options that are defined in the subsystems to enforce a certain behaviour. Therefore the modelled variability is different for each architecture and it is necessary to extract a model of the FM for each one of them; even features defined in subsystems might have different conditions on different architectures. These architectures can be seen as sub product lines of LINUX because the architecture related code evolves independently from other architectures. When users choose an architecture for which they want to build LINUX, the view of variability is limited to this architecture.

[10] `http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf`

In the following sections I describe how I analyse the configuration space of LINUX with the usage of both the intermediate models and the feature models for each architecture. Through this analyses I was able to identify defects in KCONFIG and inaccuracies of the extractor written by Tartler [Tar13] which creates a reverse-engineered image of LINUX's FM s.

3.1 Feature Models

In this Section I want to provide a short introduction of feature models in general.

Kang et al. [Kan+90] first defined a *feature* in feature-oriented domain analysis (FODA) as a prominent or distinctive user-visible aspect, quality, or characteristic of a software system. Others described a feature more briefly as an increment in program functionality [Bat05]. On the other hand, *feature models* define features and their usage constraints in product lines [Bat05]. These constraints are often visualised in tree structures showing the relationships between a *parent* feature and its *child* features (or subfeatures). According to Batory [Bat05], these relationships incorporate the following denotations:

- **Mandatory:**
  when the parent feature is enabled, the child feature is required to be enabled
- **Optional:**
  when the parent feature is enabled, selecting the child feature is optional
- **And:**
  all subfeatures connected with this relationship must be selected
- **Or:**
  one or more subfeatures connected with this relationship have to be enabled (**n:m** cardinalities are possible, at least **n** but at most **m** features have to be enabled)
- **alternative:**
  only one of the connected subfeatures can be selected

In the context of KCONFIG, an *or* or an *alternative* relationship are the different occurrences of a choice group. An *alternative* is a `boolean` choice, while an *or* group is a `tristate` choice without the possibility to enforce additional cardinality constraints. When a parent selects a child, the child is mandatory when the parent is enabled. The other way around, when a child has a dependency on the parent, the parent has to be enabled to be able to choose a value for the child and the child is optional at this point.

Rincón et al. [Rin+14] describe two defects directly related to the semantics of feature models. They claim that creating feature models correctly representing the domain is not trivial and defects might be unintentionally introduced while constructing a FM. These defects are often only recognised with extensive tool support.

The first defect which Rincón et al. [Rin+14] describe are dead features. These features cannot appear in any product of the product line, because they require a mandatory feature to be disabled or they have a contradiction in their dependencies. Secondly, Rincón et al. [Rin+14] describe false

optional features. These features are declared optional but appear in all products of the product line. This happens when a feature is required by a mandatory feature, when it takes part in a cardinality group with several dead features or when it is required by another false optional feature (transitive false optional).

Apart from the defects Rincón et al. [Rin+14] describe, there is another factor that has to be considered when searching for defects in LINUX: each architecture in this kernel has its own view of the variability (therefore, I also call them a sub product line) and a great deal of code in the subsystems is used by multiple architectures. However, not every architecture supports all features (e.g `PCI` or `IOMMU`) and some features are therefore intentionally dead for some architectures, but not for all. Another story is the configurability only seen by the architecture itself (i.e. defined in the corresponding `arch/` subdirectory), were other architectures do not have to be considered. In the following, I distinguish between global defects and architecture-internal defects. Global defects occur in the subsystems and on all architectures, where features that are architecture-internal defect are only seen by one architecture. A feature is only truely defect when it is either globally or architecture-internal defect. This insight suggests the need to analyse all architectures and infer which features are globally defect and look at the location of the definition to decide if a defect is architecture-internal.

Nevertheless, there is a special case for architecture-internal defects: a symbol can be architecture-internal defect on one architecture but not defect for another. While the underlying code can not be seen as defect in that case, it is still possible to simplify the FM where the defect occurs by solving the issue.

In the following two Sections I describe how I identified those defects in LINUX with two different model-types as a base for my analyses.

3.2 A Rule-Based Approach for the Intermediate Model

There are different forms of potential defects in feature models and these defects might have different origins. While features can be dead because of contradictions in their dependencies, features can also be dead because users or the configuration system cannot enable them in any case. Other defects are unnecessary instructions that prevent developers from grasping the declared variability more quickly.

3.2.1 Unreachable Symbols

There are three possibilities to set the value of a symbol in KCONFIG.

First ❶, users can set a value of an option. However, KCONFIG has the possibility to declare options that do not appear in the configuration menu but have an influence on other options or can be influenced by others. These options do not have an input prompt, are therefore invisible to users and are called KCONFIG-internal in Figure 2.3.

The second ❷ possibility is that a feature requires the presence of another feature. This functionality is modelled in KCONFIG through a select statement that force-enables another option, regardless of the dependencies of this other option.

As a third ❸ possibility, KCONFIG offers the possibility to declare default values setting the value of an option when the conditions for the default value are satisfied.

When a symbol is not enabled by at least one of those possibilities, it is dead and cannot be part of any product of a product line. I call this defect *unreachable* dead. The described conditions lead to three rules that have to apply for the identification of *unreachable* dead symbols:

❶ The symbol is invisible (does not have a prompt)

❷ The symbol is not selected by another symbol

❸ The symbol does not have a default value (or just default values with the value "n")

However, as I have described in Section 3.1, it is possible that an option is unreachable on one architecture (e.g. a feature in the subsystems) but accessible on another architecture. Therefore it is necessary to analyse all feature models and take the intersection of all identified unreachable symbols to get the set of global unreachable symbols. Architecture-internal *unreachable* symbols are identified by looking at the set of unreachable symbols and checking if they are defined in a KCONFIG-file within the hardware abstraction beyond `arch/`.

When an unreachable symbol is part of a propositional formula, the SAT checker does not know that this symbol cannot be enabled in practise and is free to set it to any value that satisfies the formula. With the knowledge that these options can never be part of any products of a product line, it is possible to force disable the unreachable symbols for any sub product line where they are identified. This can be done by adding the unreachable symbols to the blacklist of each model when the model is created from the intermediate representation. UNDERTAKER can later use these new entries in the blacklist to identify files or blocks that depend on these options and recognise them as defect.

3.2.2 Unnecessary Selects on Choice Values

A peculiarity in the semantics of KCONFIG is that selects ignore the dependencies of the selected symbol and the selected symbols are enabled in any case. However, the "any case" statement is not entirely true. When a symbol is selected that is part of a cardinality group (i.e. a choice value), the select statement is *silently* ignored by the configuration tool. These statements are unnecessary and can be deleted to improve the comprehensibility for developers.

This leads to two rules for the identification:

❶ The Symbol is a choice value

❷ The Symbol is selected by another symbol

The identification of these symbols is done in two steps. First, collect all symbols that are choice values and then iterate over all select statements in the intermediate model and search for these choice values as selected symbols.

3.2.3 Selects on Symbols with Dependencies

The KCONFIG guidelines suggest that selects should only be used on invisible options, i.e. options without a prompt, and on options without dependencies. However, this guideline is often violated and with the information from the intermediate model, these options can easily be identified by constructing a knowledge base about which features have dependencies and then analysing if there are features that select options with dependencies.

This leads to two rules for the identification:

❶ The Symbol has dependencies

❷ The Symbol is selected by another symbol

With the knowledge which symbols select other symbols with dependencies, developers get a chance to evaluate if the violation of the dependencies result in bugs.

3.3 Model Consistency Analysis

As a second step of my consistency analysis of the LINUX KCONFIG model I analyse the full variability model to identify features that are defect because of their aggregated conditions. This analysis is an extension of the UNDERTAKER tool. With the full variability model it is possible to identify *dead* features or features that are *false optional*. Features are dead when they cannot be part of any product of a product line or false optional when they are part in every product of a product line, but how does this translate into KCONFIG semantics?

There are two kinds of mandatory features in KCONFIG. I have described the first possibility in Section 3.1; a feature is mandatory because it is required (i.e. selected) by another feature. The other possibility is that a feature is “always on” for a certain architecture, where the architecture is the parent and the feature is always enabled because it has no prompt, no dependencies and an unconditional default that will always enable this option.

A feature is optional when it is possible to set a value to that option through defaults, manual user input or select statements. This means, every option that is not mandatory for a sub product line, is optional because of the semantics of the select statement.

Following the definition of a *false optional* by Rincón et al. [Rin+14] (described in Section 3.1), a false optional feature in KCONFIG is a feature that is selected by another feature that is *always on* or selected by a feature that is false optional itself (I also call them *transitive always on*). Dead features have contradictions in their dependencies or depend on features that are not defined in the current view of the configurability.

Since each architecture has its own view of the variability, it is necessary to analyse all sub product lines and infer if a found defects is global or evaluate if it is architecture-internal (see Section 3.1 for details).

Feature Defects

There are two potential reasons why features are dead. They can be dead because of contradictions in their dependencies or because features in the dependencies are not defined in KCONFIG. Dependencies on missing features always evaluate to "n", comparisons with missing features always evaluate to false for equality comparisons or always to true for non-equality comparisons.

```
1  menu "TI OMAP/AM/DM/DRA Family"
2      depends on ARCH_MULTI_V6
3
4  config OMAP4_ERRATA_I688
5      bool "OMAP4 errata: Async Bridge Corruption"
6      depends on !ARCH_MULTIPLATFORM
7
8  endmenu
9
10 menu "Multiple platform selection"
11    depends on ARCH_MULTIPLATFORM
12
13 config ARCH_MULTI_V6
14     bool "ARMv6 based platforms (ARM11)"
15
16 endmenu
```

Listing 3.15 – Example of a *logical* defect in a KCONFIG file.

Listing 3.15 show an example for a *logical* defect that results in the dead feature `OMAP4_ERRATA_I688`, which is dead because of a logical contradiction. This feature has a dependency on `!ARCH_MULTIPLATFORM` and the surrounding menu "TI OMAP/AM/DM/DRA Family" adds another dependency on `ARCH_MULTI_V6` to `OMAP4_ERRATA_I688`. However, the definition of `ARCH_MULTI_V6` gets a dependency on `ARCH_MULTIPLATFORM` through the surrounding menu "Multiple platform selection". This is a contradiction and `OMAP4_ERRATA_I688` can never be enabled.

```
1  config GPIO_MB86S7X
2          bool "GPIO support for MB86S7x Platforms"
3          depends on ARCH_MB86S7X
```

Listing 3.16 – Example of a *missing* defect in a KCONFIG file.

`GPIO_MB86S7X` in Listing 3.16 is dead because of the dependency on `ARCH_MB86S7X` which is never defined in KCONFIG.

Two *false optional* features are displayed in Listing 3.17. `X86` is always on for the X86 architecture and is therefore mandatory. This option selects another feature, `GENERIC_IOMAP`, which is consequently *false optional*. The lines 9–10 show the transitive false optional feature `GENERIC_PCI_IOMAP` which is selected by `GENERIC_IOMAP`.

```
config X86
        def_bool y
        select GENERIC_IOMAP

config GENERIC_IOMAP
        bool
        select GENERIC_PCI_IOMAP

config GENERIC_PCI_IOMAP
        bool
```

Listing 3.17 – Example of *false optional* defects in a KCONFIG file.

For the analysing process, the conditions for each option are translated into a model, which is displayed in Listing 3.1. Each line represents an implication I_n. The left side of the implication is the feature name and the right side is the condition for that feature.

The implication for `CONFIG_OMAP4_ERRATA_I688`:

```
CONFIG_OMAP4_ERRATA_I688 → CONFIG_MULTI_V6 && !CONFIG_ARCH_MULTIPLATFORM
```

is represented in the form of:

```
CONFIG_OMAP4_ERRATA_I688 "CONFIG_MULTI_V6 && !CONFIG_ARCH_MULTIPLATFORM"
```

in the model.

```
❶ CONFIG_OMAP4_ERRATA_I688 "CONFIG_MULTI_V6 && !CONFIG_ARCH_MULTIPLATFORM"
❷ CONFIG_ARCH_MULTI_V6 "CONFIG_ARCH_MULTIPLATFORM"
❸ CONFIG_GPIO_MB86S7X "CONFIG_ARCH_MB86S7X"
[...]
❹ CONFIG_X86 "CONFIG_GENERIC_IOMAP && !CONFIG_ALPHA && !CONFIG_ARM && [...]"
❹ CONFIG_GENERIC_IOMAP "(CONFIG_X86)"

❹ UNDERTAKER_SET ALWAYS_ON "CONFIG_X86"
UNDERTAKER_SET ALWAYS_OFF "CONFIG_COMPARE_WITH_NONEXISTENT"
```

Listing 3.1 – The model representation to the defects in a KCONFIG file.

I analyse the representation of every feature for a sub product line in the model representation in order to find defects. This analysis is done in several steps to be able to pinpoint the origin of a defect more precisely. The first two steps investigate the conditions for logical defects. The third step adds a negated condition for all *undefined* features. The last step checks if a feature is undead. I will explain each step with the help of the model in Listing 3.1:

❶ For the analysis of `OMAP4_ERRATA_I688`, I start with the presence condition for this feature: check if “CONFIG_MULTI_V6 && !CONFIG_ARCH_MULTIPLATFORM” is satisfiable.

❷ Since the presence condition was satisfiable, I conjugate the dependencies of the features referenced in the presence condition and their transitive dependencies to the formula and check again for satisfiability. This step adds the implication for `CONFIG_ARCH_MULTI_V6` to the formula. `ARCH_MULTIPLATFORM` cannot be enabled and disabled at the same time, therefore a *logical* defect is found.

❸ The expanded formula can then be analysed for undefined features. When a feature does not have an entry in the model (and thus in KCONFIG) it is undefined. These undefined features are then conjugated to the formula, with every feature negated. The example `CONFIG_GPIO_MB86S7X` has the feature `CONFIG_ARCH_MB86S7X` in its dependency, which is undefined. The first two steps of the analysis created the formula: "`CONFIG_ARCH_MB86S7X`". The third step recognises the option `CONFIG_ARCH_MB86S7X` is undefined and appends "`&& !CONFIG_ARCH_MB86S7X`" to the formula. An analysis of this formula will realise there is a contradiction and tag this option as *missing* dead.

❹ To identify *false optional* features, I use the negated presence condition of a feature and expand the formula with the relevant symbols of the KCONFIG model. Then I add all symbols that are marked to be "always on" in the model. The entry that identifies the relevant symbols as "always on" (or mandatory) is shown in line 9 in Listing 3.1. The blacklist, or features that are always disabled are shown in line 10. The addition of the "always on" items forces the SAT checker to enable these options during the analysis. The negation of the presence condition sets `!X86`; the addition of the always on symbols forces `X86` to be enabled, which is a contradiction and a false optional feature is identified.

The expanded formulas of the steps ❷–❹ can contain many implications. A high number of clauses in a condition makes it hard to spot the exact cause of a contradiction. To simplify this analysis process for developers who read the reports generated by the UNDERTAKER toolchain, I implemented an additional step to minimise contradictory formulas on the basis of the PICOMUS tool created by Lagniez and Biere [LB13], which is part of the PICOSAT toolchain[11]. I will show more details in Subsection 4.2.3.

With a small adjustment of the algorithm, it is also possible to search for dead or undead[12] files, when file preconditions are present in the model. File preconditions are displayed similarly to configuration options in models. Each file gets an implication in the model with the normalised filename on the left side and the presence condition for that file on the right side.

```
FILE_sound_soc_au1x_i2sc.c "... && CONFIG_SND_SOC_AU1XI2SC"
```

[11] `http://fmv.t/picosat/`

[12] A undead file is a file that is present in every product of a product line, similar to a false optional feature.

3.4 Integration into UNDERTAKER–CHECKPATCH

Rothberg [Rot14] extended the UNDERTAKER toolchain by the UNDERTAKER–CHECKPATCH tool. With this tool it is possible to analyse the state before a patch is applied and after a patch is applied and compare the results. This tool integrates well with the workflow of LINUX developers. Therefore, it is desirable to add support of new insights to the sequence of checks within this tools.

UNDERTAKER–CHECKPATCH recognises if there are changes to KCONFIG and will generate new models for the analysis when there are changes. With the information of both model states, it is in principle possible to integrate the previously described checks into UNDERTAKER–CHECKPATCH.

3.4.1 The Rule-Based Approach

With the addition of the arch-specific `unreachable` symbols to the blacklist[13] of each model, new `unreachable` symbols are identified by the comparison of the symbols in this list before and after this patch is applied. However, UNDERTAKER–CHECKPATCH did previously not read the models for the *before* state, since the presence conditions were only needed later. To identify `unreachable` symbols in UNDERTAKER–CHECKPATCH it is necessary to read these models, but this leads to an increase of the analysis time per patch.

Symbols can become newly unreachable when, for instance, a select on a symbol is removed by that patch and the symbol was only enabled by that select. With the knowledge of newly unreachable items, developers then can decide if they want to keep the symbol and the code depending on it or delete it.

The integration of the identification of selects on choice values or selects on features with dependencies is also integrated into the sequence of checks by comparing both states.

3.4.2 Model Consistency Analysis

The model consistency analysis is able to find dead files and dead or undead features in KCONFIG. With a specialised mode it is possible to analyse if a symbol touched by a patch became dead. This analysis is nonetheless not able to identify interactions with other symbols that might lead to dead symbols or files. However, due to the calculation time of the consistency analysis (explained in Subsection 5.2.5 or to search for dead files, an analysis of all models per patch is not feasible.

3.5 Changes to the Model of LINUX's Variability and the Model Extraction Process

During my analysis of KCONFIG and the feature representation, I identified and solved several problems in the model of LINUX's FM. Some of the inaccuracies were simpler to solve, others were more complicated to identify and to solve. Three of the simpler issues were:

[13] The UNDERTAKER toolchain will ensure items on the blacklist are always disabled during analyses.

- The extractor for the intermediate model completely ignored information about value-like features beyond the name of that feature.
- The intermediate model did not contain an accurate location for the definition of an option, this accurate location is essential for the classification of defects as architecture-internal for the *unreachable* or *consistency analysis*.
- The transformation from the intermediate model to the model ignored a quirk in the `tristate` logic. The negation in KCONFIG returns the result of $(2-v)$ where v is the value of the feature and `tristate` features can have the possible values "y" (2), "m" (1) or "n" (0). This means the negation of a `tristate` feature is true when a feature is either "n" (0) or "m" (1).

I explain the more complicated issues in the following paragraphs.

3.5.1 Comparisons with Undefined Symbols

As I have mentioned in Section 2.5.4.1, KCONFIG evaluates equality comparisons with undefined features always to false and inequality comparisons always to true. This behaviour was not modelled in the feature model. I replace the inequality comparison with a feature that has no representation in KCONFIG with a free variable which the SAT checker can enable when it is necessary to satisfy a condition. Since the equality comparison is always false, it is necessary to replace this comparison a contradiction like `FREE_1 && !FREE_1` but that would complicate the identification of the defect (Why is there a contradiction?). Instead of the obvious contradiction, I replaced the comparison with another term and introduced a flag symbol `CONFIG_COMPARE_WITH_NONEXISTENT` and added this flag symbol to the blacklist (or "always off" list) of the model (see Listing 3.1). For the term to replace the comparison I use the missing symbol and the flag symbol (e.g. "`CONFIG_MISSING && CONFIG_COMPARE_WITH_NONEXISTENT`"). The UNDERTAKER toolchain will ensure items on the blacklist will always be disabled. This will create a contradiction and when developers see the flag symbol in a minimal unsatisfiable subset (MUS) formula, they can look at the unminimised formula and search for the symbol that is responsible for the contradiction because it is not defined.

3.5.2 Choice Handling

The choice "USB Gadget Drivers" shown in Listing 2.7 confused the extractor of the intermediate model. As I have explained in Section 2.5.3, not every option between the `choice` and `endchoice` statements is a choice value. The extractor spuriously declared `USB_CONFIGFS_ACM` and `USB_CONFIGFS_SERIAL` as choice values and everything after these options was not recognised as choice value anymore, even though many other *valid* choice values were declared after that point. Another problem was, that the extractor sometimes did not assign the right choice values to the right choice group or the declaration of a choice group within another choice group.

The reason for these problems originated in the way the extraction tool traversed the underlying datastructure of the options in KCONFIG. I replaced the iterative traversion of that datastructure with a recursive traversion and solved the previously described problems.

3.5.3 Non-Constant and Constant Symbols

Developers do not always use constant and non-constant symbols (described in Subsection 2.5.1 on Page 15) in comparisons within dependencies in a disciplined way. KCONFIG *does* support comparisons with non-constant symbols (e.g. `0xdeadbeef`) and tries to parse the name of this non-constant symbol if this symbol is never defined. Nonetheless, the extractor for the intermediate model did not mark those symbols as constant symbols and they were seen as ordinary non-constant symbols in later analysis steps of the UNDERTAKER toolchain. This lead to the reporting of several referential false positive defects, which I repaired by marking those symbols as constants and ignoring them in the UNDERTAKER toolchain when a formula is scanned for undefined symbols.

3.5.4 Select and Default Statements, Input Prompt Conditions

Tartler [Tar13] described the language features "default values" and "symbol visibility" (conditions for input prompts) as harder to implement and left them unconsidered in his implementation of the model transformation. During my analysis I have identified another impreciseness in this transformation: select statements. To get a precise mapping for the exact semantics of the select statement is hard because of the ignoring of the dependencies of the selected symbol. These selects on symbols with dependencies *can* cause more false positives. Furthermore, the current implementation of the transformation from the intermediate model to the model ignores selects on `tristate` symbols. Unfortunately, I was not able to completely solve these problems because of time constraints.

3.6 Summary

In this chapter I have explained how I tackled the identification of dead and false optional features in LINUX. I started with a description of the formal semantics of feature models and explained how this translates into KCONFIG semantics. This section is followed by my rule-based approach to identify defects in the intermediate model. With this rule-based approach I was able to identify symbols that are *unreachable*, unnecessary selects on choice values and warn when there are selects on features with dependencies. I described how I extended the UNDERTAKER with one new mode for a *model consistency* analysis and another new mode to identify *defect files* through the file presence conditions in the model. Furthermore I extended the undertaker with a flag to create a minimised unsatisfiable subset of a contradictory formula and how I extended UNDERTAKER–CHECKPATCH with the analyses of the rule-based approach. Lastly I have discussed improvements to the mapping of the actual behaviour of KCONFIG to the extracted (intermediate) model. The rule-based approach is implemented in a script in the `Python` language.

In the next Chapter I evaluate the results of the tools whose implementation I described in this Chapter.

4 IMPLEMENTATION

In this chapter, I provide insights into how I engaged the issues described in the last chapter, how many lines of code were written and which languages were employed.

4.1 A Rule-Based Approach for the Intermediate Model

The rule-based approach is based on the information of the intermediate model. To perform the transformation from the intermediate model to the variability model, Tartler implemented a class in `python` that parses and preprocesses the intermediate model. Since this class already does sophisticated preprocessing and caches intermediate data in order to eliminate multiple calculations of the same result, I decided to re-use this class and implemented the script to perform the analyses of my rule-based approach in a `python` script with 323 lines of code. All rule-based analyses build on the aggregation and inferring the relevant data to identify defects.

4.1.1 Unreachable Symbols

Listing 4.18 shows the algorithm to identify *unreachable* symbols. To identify these symbols, I first collect the set of all symbols that are selected, the set of all symbols that have a prompt and finally the set of all symbols that have at least one default attribute with a value other than "n". After these sets are extracted, I iterate over all defined symbols and add a symbol to the result set `unreachable_items`, when it is not contained in one of the previously mentioned sets.

This step extracts all *unreachable* symbols of a single architecture. However, as I have described in Section 3.1, the fact that a symbol is *unreachable* on one architecture does not mean it really is a defect, since it could be reachable on another architecture. I identify the global *unreachable* symbols by forming the intersection set of the *unreachable* symbols of all architectures $\Phi_{\text{architectures}}$.

$$\varphi_{\text{globally unreachable}} := \bigcap_{n} U_n \qquad \forall n \in \Phi_{\text{architectures}}$$

Architecture-internal *unreachable* symbols are identified by iterating over the set of *unreachable* symbols for each architecture and checking with a RegExp if the location of the definition of a particular symbol starts with the corresponding architecture path (e.g. `arch/x86/`).

```
def unreachableItems(rsf):
    items = set(rsf.collect("Item", 0, True).keys())
    # select infos
    selectedSymbols = collectSelectedSymbols(rsf)
    # prompts
    promptSymbols = collectPromptSymbols(rsf)
    # defaults
    defaultSymbols = collectDefaultSymbols(rsf)

    # use information
    unreachable_items = set()
    for item in items:
        if item in selectedSymbols or item in promptSymbols:
            continue
        if not item in defaultSymbols:
            unreachable_items.add(item)

    return unreachable_items
```

Listing 4.18 – The algorithm to identify *unreachable* symbols.

4.1.2 Unnecessary Selects on Choice Values

The algorithm to identify unnecessary selects on choice values is displayed in Listing 4.19. First I collect a dictionary with all choice items and another dictionary with all select statements on an architecture and then iterate over all select statements and add the symbol to the result dictionary `foundSelects` when it is contained in the `choiceItems` dictionary. The resulting dictionary is a mapping of a selected symbol to a list of symbols that select that symbol. An example for the output of this analysis will be displayed in Section 5.1.2.

```
def identifyChoiceItemSelects(rsf):
    # key is the item, value is the choice it belongs to
    choiceItems = rsf.collect("ChoiceItem")
    selects = rsf.collect("ItemSelects", 0, True)

    foundSelects = dict()
    for (selecting, selected_list) in selects.items():
        for selected in selected_list:
            selected_item = selected[0]
            if selected_item in choiceItems:
                if not selected_item in foundSelects:
                    foundSelects[selected_item] = set([selecting])

                foundSelects[selected_item].add(selecting)

    return foundSelects
```

Listing 4.19 – The algorithm to identify selects on choice values.

4.1.3 Selects on Symbols with Dependencies

Selects on symbols with dependencies can be identified with the algorithm shown in Listing 4.20. I start with the extraction of a dictionary with symbols as key and their dependencies as values and then iterate over all select statements and add a symbol to the result dictionary `selectedBy` when the selected symbol is contained in the set of symbols with dependencies. An example for the output of this analysis will be displayed in Section 5.1.3.

```
1  # returns a dict with selected items as key and the selecting items as a set as value
2  def identifySelectWDeps(rsf):
3      selectedBy = dict()
4      deps = rsf.depends()
5      for (item, select_set) in rsf.collect("ItemSelects", 0, True).items():
6          for select in select_set:
7              (selected, expr) = select
8              if selected in deps:
9                  if not selected in selectedBy:
10                     selectedBy[selected] = set()
11
12                 selectedBy[selected].add(item)
13
14     return selectedBy
```

Listing 4.20 – The algorithm to identify selects on symbols with dependencies.

4.1.4 Impact Analysis

To measure the impact of *unreachable* symbols on the dead and coverage analysis in the UNDERTAKER toolchain, I created a module that provides the functions I used in Section 4.1.1. The module to identify *unreachable* symbols has 68 lines of code and is used in the transformation tool to add these symbols to the blacklist of the model. The adaption of the transformation tool required the addition two lines of code. However, all *unreachable* symbols are added to the blacklist and not just the architecture-internal or global *unreachables*. Identifying the global defects in the transformation tool would be a lot more effort, since this tool only has the information of *one* architecture and the UNDERTAKER toolchain already picks the right model when a file beyond `arch/` is analysed, which means, the right *unreachable* symbols are also used for analyses. To identify a global defect, the UNDERTAKER toolchain already cross checks all architectures. This implies that newly identified defects through *unreachable* symbols also have to be defect on *all* architectures, otherwise they are just tagged as locally defect.

4.1.5 Integration into UNDERTAKER–CHECKPATCH

The tool UNDERTAKER–CHECKPATCH is written in `python`. For the integration of the *unreachable* analysis, I added a new option `-c` or `–CHECK-UNREACHABLE`. When users activate this option,

UNDERTAKER–CHECKPATCH loads the model in the before state and creates a copy of the set of options on the blacklist. Later, after the tool analysed the before state and continues with the after state, UNDERTAKER–CHECKPATCH creates a copy of the set of blacklist options of the after state and forms the difference quantity of both sets to learn if new *unreachable* symbols have been introduced and warn the users accordingly. The adoption into UNDERTAKER–CHECKPATCH required 42 additional lines of code in this tool. There is currently no other way to add symbols to the blacklist of a model than with the *unreachable* analysis.

4.2 Model Consistency Analysis

The model consistency analysis extends the UNDERTAKER, which is written in `c++`, with an additional `ConsistencyAnalyzer` module. This module contains 172 lines of code, while other source files of the UNDERTAKER toolchain were expanded with 57 lines of code. The choice to expand the UNDERTAKER was obvious since the core functionalities I needed were already implemented there. These functionalities were the slicing algorithm defined by Tartler [Tar13] (which I describe in Section 4.2.2), the necessary application programming interface (API) to use a SAT checker.

The results of this analysis are dumped to one defect report per architecture and these defect reports are later evaluated with a separate `python` script with 114 lines of code. This script aggregates the results of the analysis and identifies architecture-internal and global defects by parsing the defect reports. Defects that are *undead* on one or more architectures and logically or missing dead on others are excluded from the global defects.

4.2.1 Feature Defects

The defect analysis itself is constructed as described in Section 3.3. When a defect is found, I use the MUS functionality of the UNDERTAKER toolchain to minimize the defect formula and print this minimized formula to the defect report. However, I needed to adapt the algorithm in the UNDERTAKER to be able to extract the minimal unsatisfiable formula for every SAT checker object. The algorithm operates with the following steps for each symbol X in the model:

❶ Check the presence condition of the symbol for satisfiability ($X_1 \vee X_2$)

❷ Check expanded the presence condition of the symbol with the transitive dependencies for satisfiability. I perform the identification of the transitive dependencies of feature X with the with the help of the slicing algorithm described by Tartler et al. [Tar+11b], which I explain briefly in Subsection 4.2.2. To identify the set of relevant features Φ_{slice_X}, I use the slicing algorithm with the features mentioned in the presence conditions $X_1 \vee X_2$ of the analysed feature X. This algorithm identifies a set of relevant features Φ_{slice_X} that can be used to create a *slice* $\varphi_{\text{KCONFIG slice}_X}$ of the model.

$$\varphi_{\text{KCONFIG slice}_X} := \bigwedge_n I_{n_1} \vee I_{n_2} \qquad \forall n \in \Phi_{\text{slice}_X}$$

This slice $\varphi_{\text{KCONFIG slice}_X}$ is then combined with the presence condition of feature X and checked for satisfiability:

$$SAT\left((X_1 \vee X_2) \wedge \varphi_{\text{KCONFIG slice}_X}\right)$$

If the formula is not satisfiable, I have identified a defect.

The identification of the relevant features and the usage of a *slice* of the model instead of the whole variability model optimises the calculation time of the SAT checker, which I use to check a formula for satisfiability.

❸ To check for *missing* defects, it is necessary to identify all undefined options $\Phi_{\text{undefined}}$ in $\varphi_{\text{KCONFIG slice}}$ and conjugate them negated to the formula and check this new formula for satisfiability to identify a *missing* or *referential* defect.

$$\varphi_{\text{unidentified}} := \bigvee_n \quad u_n \qquad \forall n \in \Phi_{\text{undefined}}$$
$$SAT\left((X_1 \vee X_2) \wedge \varphi_{\text{KCONFIG slice}_X} \wedge !\left(\varphi_{\text{unidentified}}\right)\right)$$

❹ False optional or undead features are identified with the usage of the negated presence condition of the feature X, conjugated with the transitive conditions of this feature and then checked for satisfiability.

$$SAT\left(!(X_1 \vee X_2) \wedge \varphi_{\text{KCONFIG slice}_X}\right)$$

4.2.2 Configuration Model Slicing

To get a formula with only the *relevant* configuration constraints, Tartler et al. [Tar+11b] employed the slicing algorithm displayed in Algorithm 4.1.

```
Require: S initialised with an initial set of items
  R = S
  while S ≠ ∅ do
    item = S.pop()
    PC = presenceCondition(item)
    for all i such that i ∈ PC do
      if i ∉ R then
        S.push(i)
        R.push(i)
      end if
    end for
  end while
  return R
```

Algorithm 4.1 – Algorithm for configuration model slicing. Algorithm taken from [Tar+11b].

This algorithm starts with worklist S initialised with the items mentioned in the presence condition of the analysed symbol. The list R is initialised with the contents of the worklist S and stores the result list. The algorithm iterates until the working stack S is empty. Each iteration removes an

entry from the working stack, asks the model for the presence condition of this entry through the function `presenceCondition(item)` and then analyses every configuration item in that presence condition. When the configuration item is not already present in the result set R it is added to R and the working stack S.

4.2.3 The Minimal Unsatisfiable Subset (MUS)

The problem of identifying the minimal subset of unsatisfiable clauses is called MUS in literature [BSW03; Bru03; BS05; LS08]. Biere [Bie08] created tools to solve SAT problems; his PICOSAT tool is used by the UNDERTAKER toolchain to check propositional formula for satisfiability. Later, he expanded this toolchain with the PICOMUS tool which takes an unsatisfiable formula, minimises this formula by algorithmically eliminating all unnecessary clauses and returns the MUS formula. An example for a minimised formula is shown in Listing 4.1. The first four lines show statistics about the minimisation done by PICOMUS. The tool was able to minimise the formula from 6.581 clauses to just 559 clauses. When all intermediate variables[14] are ignored, a tight formula (lines 6–8) is received that can easier parsed by developers to identify variability problems than the unminimized formula, which often has several thousands of lines.

```
1 Minimized Formula from:
2 p cnf 2794 6581
3 to
4 p cnf 2794 559
5 (B00) ^ (CONFIG_SND_SOC_AU1XI2SC) ^
6 (!FILE_sound_soc_au1x_i2sc.c) ^ (!CONFIG_SND_SOC_AU1XI2SC)
7 ^ (!B00 v FILE_sound_soc_au1x_i2sc.c)
```

Listing 4.1 – An example for an additional report generated by the MUS analysis.

My analysis of every configuration entry in the model automatically calculates this MUS formula and reports only the minimised formula.

4.3 Summary

In this chapter I provided a detailed insight into the algorithms I have implemented to identify the defects I have described in the previous chapters. First I started with an explanation of the rule-based approach and supplied code examples of the algorithms written in `python`, how I implemented the impact analysis of the *unreachable* symbols and how I adjusted UNDERTAKER–CHECKPATCH to perform an *unreachable* analysis. Then I continued with the model consistency approach, where I explained every step of this analysis in detail, followed by an illustration of the slicing algorithm that was coined by Tartler [Tar13] and showed an output of the MUS analysis.

In the next chapter I will continue with a detailed evaluation of the presented approaches.

[14] Variables without a representation in models.

5 EVALUATION

In this chapter I present a detailed evaluation of the approaches described in Chapter 3. I begin with the assessment of my rule-based approach. First I show how many potential problems the tool identifies, then I discuss how the calculation time changes over a wide range of versions and finally what the impact of these findings is. Then I continue with a review of the model consistency analysis, where I start with examples of the defects this analysis has identified, discuss the influence of the models' inaccuracies, continue with a discussion about the performance and explain what the results of the adaption of the algorithm for file analyses are. Lastly I assess another problem I have investigated.

5.1 The Rule-Based Approach

The rule-based approach offers several analyses for defects or potential problems on the intermediate model. In this Section, I describe which problems or defects the tool identifies and what the implications of these findings are.

5.1.1 Unreachable Symbols

I did an *unreachable* analysis on all architectures on all major versions between Linux/`v2.6.31` and Linux/`v4.1`. A closer look at Linux/`v4.0` reveals an average of 296.1 *unreachable* symbols per architecture. However, only 15.88 percent (47.3) of these symbols are architecture-internal. This means that 84.12 percent of the total unreachable symbols per architecture are declared in the shared code within the subsystems and therefore are reachable in other architectures, except for the symbols that are *unreachable* for all architectures.

An intersection over all sets of unreachable symbols per architecture identifies symbols that are unreachable on all architectures. This step reveals 16 globally unreachable symbols in Linux/`v4.0`[15].

[15] Which are: `ASYNC_TX_DISABLE_PQ_VAL_DMA`, `ASYNC_TX_DISABLE_XOR_VAL_DMA` `BROKEN`, `FSL_FTM_TIMER`, `PINCTRL_MESON`, `REED_SOLOMON_ENC16`, `SAMSUNG_USBPHY`, `SH_LCD_MIPI_DSI`, `SND_DAVINCI_SOC_I2S`, `SND_SOC_AU1XI2SC`, `SND_SOC_SH4_SSI`, `SND_SPEAR_SOC`, `SND_SPEAR_SPDIF_IN`, `SND_SPEAR_SPDIF_OUT`, `TPS65911_COMPARATOR`, `WANT_COMPAT_NETLINK_MESSAGES`

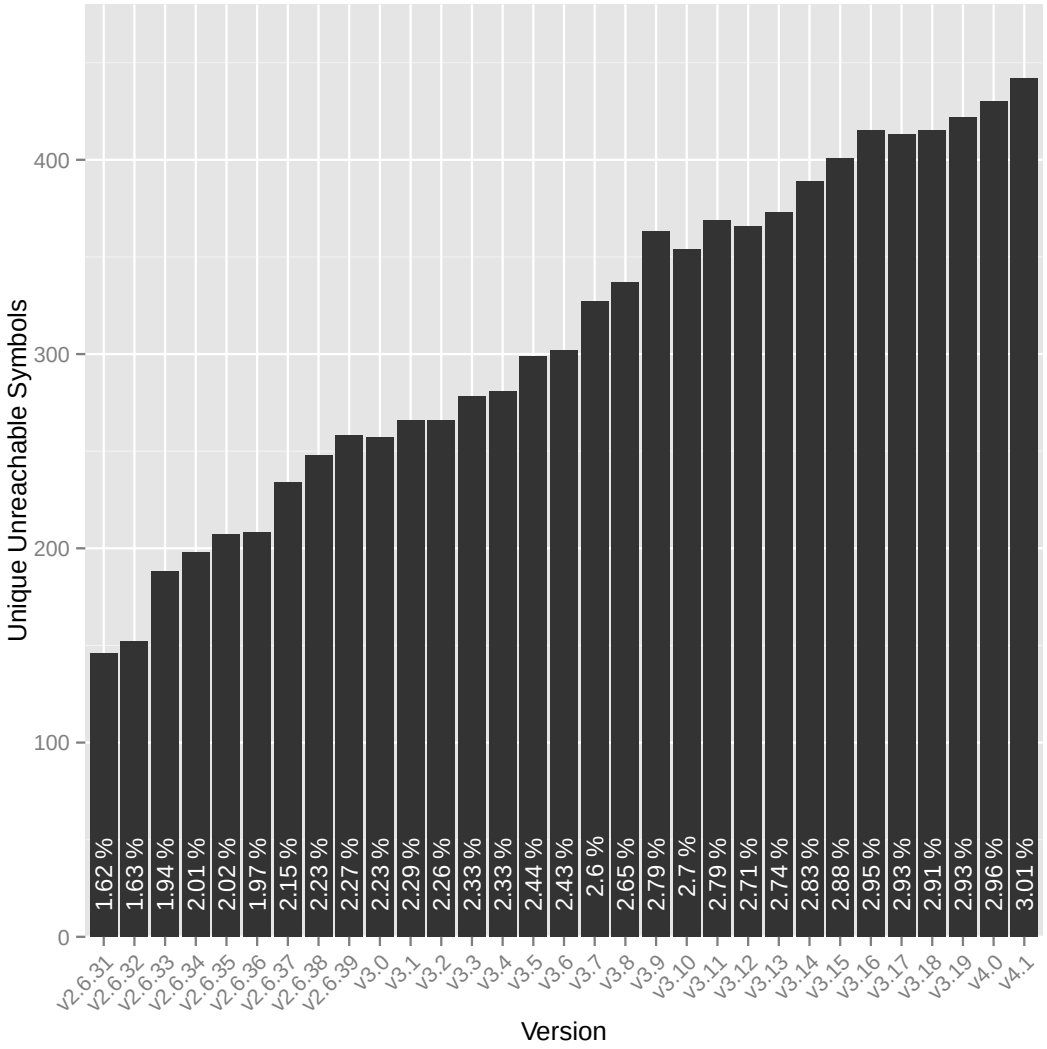

Figure 5.1 – The number of unique unreachable options from Linux/`v2.6.31` until Linux/`v4.1`

The growth of the total number of *unique unreachable* symbols[16], without the duplicates in different architectures, over all analysed versions is shown in Figure 5.1. However, as I have explained in Chapter 1 and shown in Figure 1.1, the number of features in the LINUX kernel is constantly rising. The percentages on the lower end of the bars indicate which percentage of the total amount of features are unreachable for each version. The graph shows, this number is constantly rising; starting with 1.62 percent for Linux/`v2.6.31` and 3.01 percent for Linux/`v4.1`.

Unreachable options cannot be enabled by users through graphical or commandline configuration tools and are not enabled by other language constructs. However, another way of changing the configuration is editing the files that store the configuration (as described in Section 2.1), even though it is discouraged to do that. If users or developers decide to manually add symbols that are unreachable on the destination architecture to the files that store the configuration, KBUILD will remove the *unreachable* symbols when the kernel compilation process has started by calling `scripts/kconfig/conf --silentoldconfig Kconfig`. Thus, these options are truely dead and cannot be enabled without altering KBUILD.

Another fact worth mentioning is that there are no symbols in Linux/`v4.0` that are *only* selected by one or more *unreachable* symbols and have no default and no prompt otherwise, i.e. there are not *transitive unreachable* symbols. However, there are options that depend on unreachable features but are not necessarily dead (e.g. when a symbol has the following dependency expression "`CONFIG_S390 || BROKEN`", it is possible to enable that feature for the `s390` architecture but it is dead for all other architectures, because `BROKEN` is unreachable).

5.1.1.1 Studying the Impact of Unreachable Symbols

With the knowledge that *unreachable* options are dead it is possible to add the identified options to the blacklist of the model of each architecture. These blacklist entries are always disabled in the analyses done by the UNDERTAKER toolchain and the tools are able to identify new dead files and CPP blocks. In the following subsections, I describe the impact of *unreachable* symbols on later build stages.

Dead Analysis

The dead analysis in the UNDERTAKER toolchain is able to identify dead files and CPP blocks in a LINUX source tree, as described in Section 2.6.1. A comparative analysis for all files in Linux/`v4.0`, with and without the addition of *unreachable* symbols to the blacklist, yields an identification of 347 additional defects (see Table 5.1). 49 of of these defects are dead files. This is an increase of 23.75 percent in found defects.

One of those newly recognised defects is the dead file `sound/soc/au1x/i2sc.c`, which requires the option `CONFIG_SND_SOC_AU1XI2SC` enabled to be compiled. A closer look to the definition of this symbol (see Listing 5.21) reveals this option is *unreachable* and the file is therefore dead.

[16] An *unique* unreachable symbol is a symbol that is unreachable on one or more architectures

Linux/v4.0	normal	+ Unreachable Information	
# Defects	1461	1808	+347 (+23.75%)
code	99	99	–
KCONFIG logic	257	469	+212 (+82.49%)
KBUILD logic	537	651	+114 (+21.23%)
missing	568	589	+21 (+3.70%)

Table 5.1 – The impact of the addition of *unreachable* information to the dead analysis.

```
config SND_SOC_AU1XI2SC
    tristate
```

Listing 5.21 – The definition an *unreachable* symbol in `sound/soc/au1x/Kconfig`.

Coverage Analysis

The additional identified defects described in Subsection 5.1.1.1 also influence the coverage analysis of the UNDERTAKER toolchain. The toolchain generates a set of partial configurations to cover all blocks within a single file, but blocks that are recognised as dead are excluded by the algorithm to create partial configurations. After these configurations are created, the toolchain employs KCONFIG to expand each *partial* configuration to a full configuration[17] and then checks if the file is in the list of compiled files and if the relevant KCONFIG symbols are still contained in the created `.config` file. The expansion process removes *unreachable* symbols, when they are mentioned in a partial configuration. The UNDERTAKER toolchain recognises this removal and flags the configuration as invalid. This means if a configuration covers multiple blocks, where some of these blocks are valid and one or more depend on an *unreachable* symbol, that these valid blocks were not recognised as covered.

With the usage of the *unreachable* information in the model, more blocks are recognised as dead and less configurations are flagged as invalid. This means the configuration coverage $\mathcal{CC}$ (see Equation I on Page 29) is higher because less invalid configurations are created and the normalised configuration coverage $\mathcal{CC}_N$ (see Equation II on Page 29) is higher because more blocks are recognised as dead.

To evaluate the impact of the *unreachable* information to the coverage analysis, I analysed four of the most common architectures on Linux/`v4.0` (`arm, mips, s390, x86`) with and without the consideration of *unreachable* symbols. The most notable change happens for the `mips` architecture, where the $\mathcal{CC}$ increases from 52.6 percent to 67.1 percent and the normalised $\mathcal{CC}_N$ increases from 67.7 percent to 88.0 percent. For the three other architectures, the $\mathcal{CC}$ increases only marginally. The changes in the configuration coverage leads to an increase of the *mean* configuration coverage by 7.1 percent. The full tables with the results of the coverage analysis are displayed in Appendix A.

[17] This is done by calling `KCONFIG_ALLCONFIG=path/to/config make allyesconfig`. Instead of `allyesconfig`, other expansion strategies can be used (e.g. `allmodconfig`). KCONFIG uses the configuration as a basis and then iterates over all declared options and tries to set each of the options to "y" when the strategy is `allyesconfig` and each option to "m" when the strategy is `allmodconfig`. During this process, KCONFIG checks if the option it tries to enable would cause a violation of the dependencies and leaves the option disabled in case of a violation.

One explanation of this huge increase in the configuration coverage for the embedded `mips` architecture is, that this architecture does not need to provide support for all hardware features in the kernel and thus does not enable all bridge features (starting with HAVE_ or HAS_, described in Subsection 2.5.6), which then become unreachable on this architecture. However, the mainframe architecture `s390` also does not need all hardware features, similar to `mips`, but this architecture had only an increase of 7.8 percent and started at 80.1 percent. One possible explanation for this is, that many features have `!S390` in their dependencies, dead blocks are recognised and the normalised $\mathcal{CC}_N$ is higher. `mips` on the other hand uses another approach which leaves the unused hardware features *unreachable* which were not recognised as dead by the UNDERTAKER toolchain before.

Functional Defects

Up to that point I have shown the impact on the dead analysis and the coverage analysis of the UNDERTAKER toolchain. However, the question arises if dead blocks caused by *unreachable* symbols also cause *functional* defects at run-time (e.g. null pointer dereference, uninitialized variables, etc.). Therefore I take a closer look at the *unreachable* symbols in Linux/`v4.0` and check their declarations and how they are employed in later build phases.

Out of 16 global unreachable symbols in Linux/`v4.0`, *ten* of these symbols are only used to compile a certain file[18] or enable a whole folder[19], *three* are used to enable only CPP blocks[20] that declare function dummies or add short if statements, *two* are only used in attributes of other options[21] and at least one of them is intentional `BROKEN`, which is used to tag features as defective. All of those global unreachable symbols are defined and used within the subsystems `drivers`, `crypto`, `lib`, `sound`, `net`. The option `BROKEN`, however, is used in 74 occasions in dependencies or if-statements, effectively disabling other options intentionally.

Because of the number of architecture-internal *unreachable* symbols for each architecture (on average 47.3 per architecture, see Section 5.1.1), I pick the `arm` architecture for a more detailed analysis. The pattern of these architecture-internal defects is, however, similar to the previously analysed global defects with *two* exceptions, `SBUS` and `EISA`, which are both mentioned in many documentation and source files spread across multiple subsystems and architectures, but both symbols are also defined on other architectures. *Two* symbols are only used in an `#ifndef` statement which are always true[22]. Others only enable a single file or *one* to *three* `#ifdef` blocks (e.g. `S3C64XX_DEV_SPI1`, `S3C_DEV_I2C5` and many more in `arch/arm/plat-samsung/Kconfig`).

The commit e220ba60223a9d introduced the architecture-internal *unreachable* symbol `VERIFY_PERMISSION_FAULT` in the year 2009. This commit was intended to fix incorrect permission faults for qsd8650 and also adds 19 lines of conditionally compiled assembler code. It is not clear if this option was ever enabled by other options in the time between the merge of this commit and

[18] `ASYNC_TX_DISABLE_PQ_VAL_DMA`, `FSL_FTM_TIMER`, `SAMSUNG_USBPHY`, `SND_DAVINCI_SOC_I2S`, `SND_SOC_AU1XI2SC`, `SND_SOC_SH4_SSI`, `SND_SPEAR_SOC`, `SND_SPEAR_SPDIF_IN`, `SND_SPEAR_SPDIF_OUT`, `TPS65911_COMPARATOR`

[19] `PINCTRL_MESON`

[20] `ASYNC_TX_DISABLE_PQ_VAL_DMA` and `ASYNC_TX_DISABLE_XOR_VAL_DMA`, `REED_SOLOMON_ENC16`.

[21] `SH_LCD_MIPI_DSI`, `WANT_COMPAT_NETLINK_MESSAGES`.

[22] `ARCH_HAS_ILOG2_U32` and `ARCH_HAS_ILOG2_U64`

Linux/v4.0. Nonetheless, due to the lack of detailed knowledge of this hardware platform, the confirmation if this is or was a functional defect has to be made by maintainers.

The conclusion of this analysis is, *unreachable* symbols always imply dead code but if this dead code causes functional defects awaits confirmation. Without consideration of the exceptions SBUS and EISA, the dead code is clustered and not scattered across multiple subsystems. Both exceptions are defined in multiple architectures but are *unreachable* for arm. Usages in subsystems are therefore not dead and *no* defects. These cases are excluded in the standard dead analysis within the UNDERTAKER toolchain, since the tools cross-check different architectures and are only reported as *locally* dead.

5.1.2 Unnecessary Selects on Choice Values

Unnecessary selects on choice values are select statements that are always silently ignored by KCONFIG. However, the number of these futile statements is limited. Linux/v4.0 has exactly *two* of these statements on all architectures. The Listing 5.22 shows an example output of the tool for arm.

```
Arch:  'arm'  has the following selects on ChoiceItems:
VMSPLIT_1G is selected by set('ARCH_IOP13XX')
VIRT_CPU_ACCOUNTING_GEN is selected by set('NO_HZ_FULL')
FB_LITTLE_ENDIAN is selected by set('FB_MB862XX_LIME')
```

Listing 5.22 – An example output of the tool to identify unnecessary selects on choice values.

The arm architecture is the only architecture with *three* of these statements. The first listed select on VMSPLIT_1G is exclusive to arm while the others happen in all other architectures. The contents of the set on the right side identify which symbols have such a undisciplined select statement.

5.1.3 Selects on Symbols with Dependencies

One quirk of the select statements (described in Section 2.5.4.2) in KCONFIG is that they ignore the dependencies of select*ed* features. While this does not necessarily mean that problems *have* to occur when a feature with dependencies is selected, but problems *can* arise, depending how much the code of the selected feature relies on code of its dependencies. KCONFIG seems to rely on users to enable necessary dependencies and only warns when the dependencies of a selected feature would directly contradict the chosen configuration and enables the selected feature anyway.

The architectures in Linux/v4.0 have 6741 select statements on average. Only *three* architectures stand out: arm has 9053 select statements, mips has 7860 and powerpc has 7295; the number of select statements of all other architectures fluctuate between 6514 and 6711.

Although the KCONFIG rules state, in order to avoid problems, selects should preferably be done on invisible symbols – symbols without a prompt – and on symbols without dependencies, the average number of selects on symbols with dependencies of all architectures is 5396. This means an average of 79.99 percent of all selects are on symbols with dependencies. However, only the

`arm` architecture stands out with 6550 selects. This sheer amount of potentially problematic selects makes it hard to identify selects that lead to conflicts at compile time or at run-time.

An example output of the analysis is displayed in Listing 5.23. The symbol on the left side is selected by the options on the right side.

```
Arch:  'arm'  has the following selected symbols w/ dependencies:
'MMC_DW_PLTFM' is selected by set('MMC_DW_K3', 'MMC_DW_ROCKCHIP')
'BLK_DEV_MD' is selected by set('DM_RAID')
'DVB_OR51132' is selected by set('VIDEO_CX88_DVB')
'DM_BUFIO' is selected by set('DM_VERITY', 'DM_SNAPSHOT')
[...]
```

Listing 5.23 – An example output of the tool to identify selects on symbols with dependencies.

5.1.4 Accuracy

The intermediate model is a direct mapping of the declarations in KCONFIG to the intermediate representation and the extractor is based on KCONFIG tools. This fact leads the portrayal to be exact and no accuracy issues arise, since the rule-based approach only aggregates the information presented in the intermediate models and then infers defects.

5.1.5 Performance

I evaluated the performance of the rule-based approach on a machine with a Intel i7-2600 CPU @ 3.40GHz with 8 logical cores and 8GiB RAM. Figure 5.2 shows the graph with the calculation times for the *unreachable analysis*. The blue area shows the time it takes to parse the intermediate models. Compared to the total run time of the tool, reading the models takes approximately 96 percent of the total run time and the actual analysis the remaining 4 percent. This distribution is similar for the other rule-based analyses. However, the current approach is only single threaded. I considered to parallelise the parsing process with futures, but the classes I re-used are written in PYTHON 2.7 and futures are only fully supported in PYTHON 3.4 or later. Approaches with the `multiprocessing` or `threading` libraries are subject to further work.

The values shown in Figure 5.2 are the average of 5 executions on the previous described machine. The complete analysis takes 137.17*s* for Linux/`v4.0`, while parsing the intermediate models is finished after 131.03*s*. Because of that distribution, the analyses can be used in tools that already utilise the parsed intermediate model (e.g. to perform the model transformation) without a huge performance impact. To calculate and add the *unreachable* symbols to the model had no measurable impact to the run time of the transformation tool.

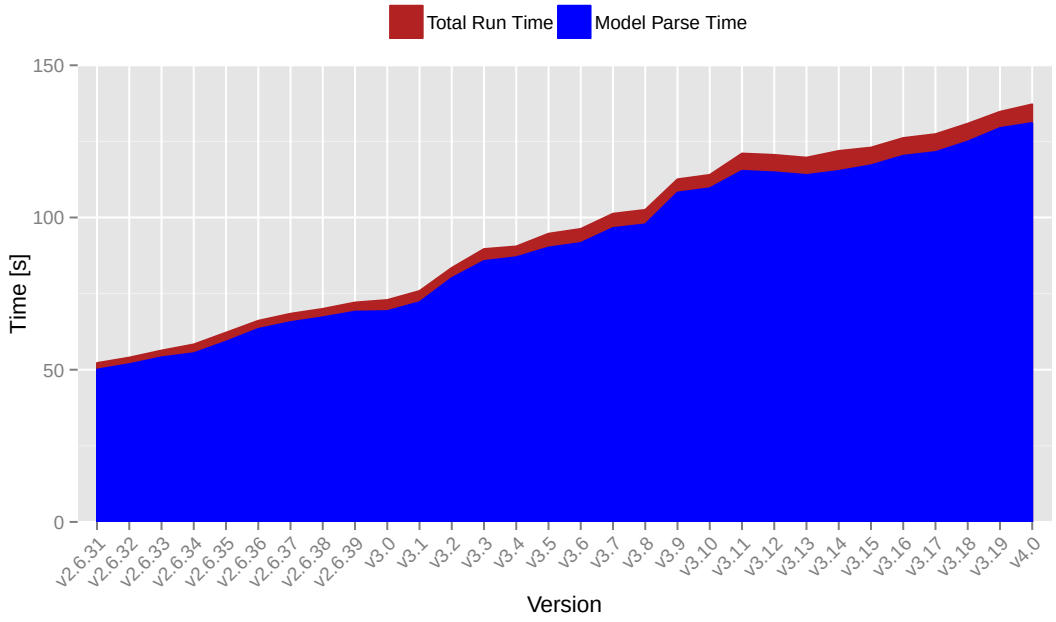

Figure 5.2 – The calculation times for the *unreachable analysis* between Linux/`v2.6.31` and Linux/`v4.0`. Reading the models is the most expensive part, while the analysis is just a small percentage ($\sim$ 4%) of the total run time.

5.1.6 Finding Defects with UNDERTAKER–CHECKPATCH

Previously, UNDERTAKER–CHECKPATCH did not parse the intermediate model of the *before* state. The addition of the *unreachable* analysis made this step necessary to identify defects. This parsing step increases the analysis time of a trivial example patch to `arch/arm/Kconfig` (see Listing 5.24) on a Intel i7-2600 CPU @ 3.40GHz with 8 logical cores and 8GiB RAM from 27*s* to 43*s* but correctly identifies the newly unreachable symbol.

Apart from the increase in the calculation time per patch, there is another downside when *unreachable* symbols are added to the models. UNDERTAKER–CHECKPATCH tries to provide hints for *logical defects* by listing options that are mentioned on the "always on" and "always off" lists but are also present in the contradictory formula. With the addition of the *unreachable* symbols to the blacklist, the list of hints is most of the time a lot longer while the mentioned options are not necessarily responsible for the contradiction. More helpful for the identification of logical defects is the MUS formula, which can be created with the `--mus` option of UNDERTAKER–CHECKPATCH.

The identification of unnecessary selects on choice values and selects on symbols with dependencies in UNDERTAKER–CHECKPATCH remains subject to further research. While the identification of futile selects is useful, it is possible that warnings about new selects on symbols with dependencies would generate too much noise for developers, since nearly 80 percent of all selects fit that type.

```
diff --git a/arch/arm/Kconfig b/arch/arm/Kconfig
index cf4c0c9..f44b3a5 100644
--- a/arch/arm/Kconfig
+++ b/arch/arm/Kconfig
@@ -719,6 +719,9 @@ config ARCH_S3C24XX
                   (and derivatives).

+config FOOOBARBAZ
+        bool
+

config ARCH_S3C64XX
        bool "Samsung S3C64XX"
        select ARCH_REQUIRE_GPIOLIB
```

Listing 5.24 – A trivial patch to analyse with UNDERTAKER–CHECKPATCH to show the functionality of the added *unreachable* analysis.

5.2 The Model Consistency Analysis

The model consistency analysis aims to identify symbols that are dead because of *logical* inconsistencies in the form of contradictions, because of violations in the referential integrity in the form of dependencies on undefined options and false optional features that appear configurable but are always part of a specific product in a sub product line in LINUX. In this section I show which defects this approach has identified for Linux/`v4.0`, continue with a discussion about accuracy and performance, and explain the findings of the adaptation to identify dead *files* at the end of the section.

5.2.1 Logical Dead Symbols

The analysis of Linux/`v4.0` identified only *one* logical dead symbol. This defect occurred on the `arm` architecture and is shown in Listing 5.25, where the option `OMAP4_ERRATA_I688` is dead because of a logical contradiction in its dependencies. The feature itself requires `ARCH_MULTIPLATFORM` to be disabled and gets an additional dependency on `ARCH_MULTI_V6` through the surrounding menu. Looking at the definition of this option reveals, however, that this option gets a dependency on `ARCH_MULTIPLATFORM` through another menu. This last dependency requires `ARCH_MULTIPLATFORM` to be enabled and creates a contradiction since this option cannot be enabled and disabled at the same time.

I reported this defect to the maintainers and submitted a patch[23] to remove the feature and the code depending on it after the developers suggested to remove the features since nobody has complained for a few years that the feature is dead.

[23] `https://lkml.org/lkml/2015/2/25/503`

```
menu "TI OMAP/AM/DM/DRA Family"
    depends on ARCH_MULTI_V6

config OMAP4_ERRATA_I688
    bool "OMAP4 errata: Async Bridge Corruption"
    depends on !ARCH_MULTIPLATFORM

endmenu

menu "Multiple platform selection"
   depends on ARCH_MULTIPLATFORM

config ARCH_MULTI_V6
    bool "ARMv6 based platforms (ARM11)"

endmenu
```

Listing 5.25 – The identified logical defect on the `arm` architecture.

5.2.2 Referential Dead Symbols

Referential dead symbols are dead because an option in their dependencies is not defined in the current sub product line. An analysis on Linux/`v4.0` reveals 27 referential dead options. One of those referential dead options is `REGULATOR_MAX77843` displayed in Listing 5.26, which has only a dependency on `MFD_MAX77843`. This feature in the dependency is, however, never defined and the dependency is therefore always false and the option and the file (`drivers/regulator/max77843.c`) which depends on that feature, is dead.

```
config REGULATOR_MAX77843
    tristate "Maxim 77843 regulator"
    depends on MFD_MAX77843
```

Listing 5.26 – An example for a missing defect in Linux/`v4.0`.

Nonetheless, a closer look at this option exposes that it was introduced in February in 2015. It is therefore very likely that this was one of the first patches to add a new driver and the others to actually enable the code remain to be merged.

5.2.3 False Optional Features

False optional features are features that appear configurable but are required through other options. Linux/`v4.0` contains 28 of these defects which come in several forms as explained in the following paragraphs.

```
config ARCH_SPARSEMEM_ENABLE
    def_bool y

config ARCH_SPARSEMEM_DEFAULT
    def_bool ARCH_SPARSEMEM_ENABLE
```

Listing 5.27 – An trivial example of a false optional defect in Linux/v4.0.

The first form is displayed in Listing 5.27 and shows that option ARCH_SPARSEMEM_DEFAULT is *transitive always on*. This happens because the default value of this feature is the value of ARCH_SPARSEMEM_ENABLE which is always on through its definition. This means the definition of ARCH_SPARSEMEM_DEFAULT could be simplified with a **def_bool y** declaration.

```
config ARM64
    def_bool y
    select ARCH_WANT_FRAME_POINTERS

config ARCH_WANT_FRAME_POINTERS
    bool

config FRAME_POINTER
    bool "Compile the kernel with frame pointers"
    depends on DEBUG_KERNEL && (CRIS || ..) || ARCH_WANT_FRAME_POINTERS
    default y if (DEBUG_INFO && UML) || ARCH_WANT_FRAME_POINTERS
```

Listing 5.28 – An more complex example of a false optional defect in Linux/v4.0.

A more complicated case is shown Listing 5.28, where the architecture feature ARM64, which is always enabled, selects ARCH_WANT_FRAME_POINTERS. The FRAME_POINTER option is therefore always enabled and appears configurable but is always part of every product in the arm64 sub product line. This means the definition of FRAME_POINTER could be simplified for arm64.

```
config ARM64
    def_bool y
    select ARM_GIC

config ARCH_MEDIATEK
    bool "Mediatek MB..."
    select ARM_GIC
```

Listing 5.29 – An example of an unnecessary select in Linux/v4.0.

The last example in Listing 5.29 does not show a false optional feature, but an unnecessary select for the feature ARCH_MEDIATEK. ARM_GIC is always enabled because it is selected by the architecture's feature ARM64 and the select in ARCH_MEDIATEK is redundant and can be dropped.

5.2.4 Accuracy

The quality of the results of the model consistency analysis highly depends on the accuracy of the used representation of the KCONFIG feature model. I have explained in Section 3.5 how I improved the models' accuracy, but symbol visibility, default values and selects persist to be hard to map to propositional formulas and the mapping of these attributes remains approximative. The inaccuracies were therefore only reduced but not eliminated. Even though these challenges endure, I was able to identify several consistency issues after eliminating a few false positives caused by selects on symbols with dependencies by hand.

5.2.5 Performance

For the model consistency analysis I used an AMD server machine with a Processor 6180 SE CPU @ 2.5GHz with 48 physical cores and 64 GiB RAM to analyse all 30 architectures simultaneously with *one* process per architecture. Although the slicing algorithm described in Subsection 4.2.2 identifies the set of relevant features for each presence condition of a symbol, the analysis proves to be costly in computing time. Figure 5.3 shows the computing times in seconds, starting with Linux/`v2.6.31` to Linux/`v4.1`. Over the course of the *eleven* versions, *three* architectures stand out in terms of

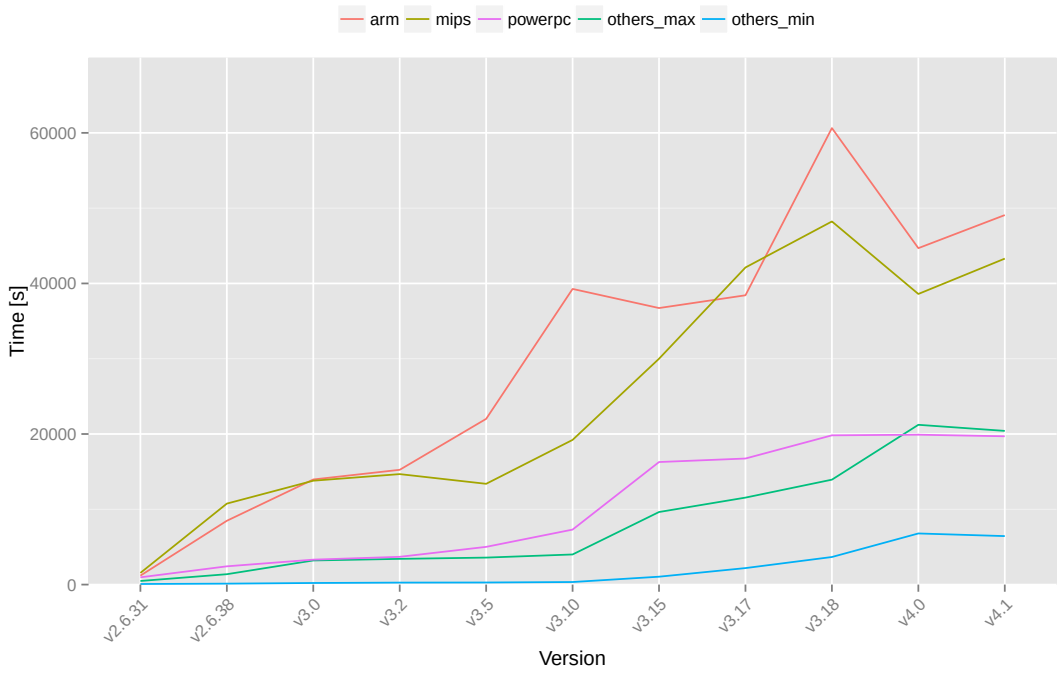

Figure 5.3 – The calculation times in seconds of the *consistency analysis* between Linux/`v2.6.31` and Linux/`v4.1`.

computing time: `arm`, `mips` and `powerpc`, which I display separately. All computation times of other architectures vary between the `others_min` and `others_max` lines shown in the graph. While the analysis of earlier versions is finished in under an hour, the calculation time is constantly rising similar to the number of features. However, the evaluation time of the three mentioned architectures seems to grow exponentially, in contrast to all other architectures for which the analysis time seems to grow linearly. The `arm` architecture reaches a peak in Linux/`v3.18` with nearly 17 hours of computing time, followed by the `mips` architecture taking 13.39 hours. After this peak, both versions drop to 12.38 hours for `arm` and 10.72 hours for `mips`. I was not able to identify the reasons for this peak.

Possible Optimisations

Having these calculation times in mind, the question arises if the process could be optimised:

- Unfortunately, it is not possible to re-use intermediate results during this analysis. The satisfiability of one symbol does not mean it could not cause a defect in its usage in combinations with other symbols and their conditions.
- Further parallelisations might be possible, but the analysis of each architecture is started in a distinct process and the limitation of started threads would require the communication of those processes. The strategy of starting one process per analysed file, or model in that case, instead of lightweight threads proved to be a improvement in the robustness of the tool since potential errors do not cease all other processes. However, this optimisation remains a topic of further research.

5.2.6 Finding Dead or Undead Files by Analysing File Presence Conditions

The algorithm described in Subsection 3.3 is able to analyse the presence conditions of files with just a few adaptations. This analysis is about as fast as the normal model consistency analysis and identifies 44 dead files which are dead because there is a contradictions in their presence condition or because there are features in the presence condition and their dependencies missing. This step also identifies *two* previously mentioned defective files: `drivers/gpio/gpio-mb86s7x.c`, which is classified as missing defect, and `sound/soc/au1x/i2sc.c`, which is tagged as logical defect. Defective files, which are architecture-internal, are found in 14 architectures. The `arm` architecture has, for example, 4 dead files because of a logical contradiction and 1 dead file because of undefined features. The full list of all identified defective files is displayed in Appendix B.

5.3 Another Investigated Problem

I also have investigated the occurrence of *unused* symbols in Linux/`v3.18` and Linux/`v4.0`. These symbols are declared in KCONFIG but never used in KBUILD or source files. My analysis did not find any *unused* symbols in Linux/`v3.18` and only *one* in Linux/`v4.0`. The identified option

ARCH_SPEAR_AUTO was introduced in the year 2012 with commit $5b65fc56$ and later became *unreachable* with commit $3b0b8ec9$ where artefacts of a feature were removed that never made it into the kernel.

5.4 Summary

In this chapter, I evaluated several approaches to identify defects and potential problems in KCONFIG. First, I have shown that there are many *unreachable* symbols in LINUX. A closer look at Linux/v4.0 yielded 16 global *unreachable* symbols and on average 47.3 architecture-internal *unreachable* symbols per architecture. An evaluation of the impact of these *unreachable* symbols revealed that the UNDERTAKER finds 347 additional defects, which is an increase of 23.75 percent. A coverage analysis uncovered an increase of the $\mathcal{CC}_N$ for the mips architecture from 67.7 percent to 88.0 percent, while the impact on other architectures remained small but noticeable.

The Linux version v4.0 also has *two* unnecessary selects on choice values and on average 5396 selects on symbols with dependencies per architecture.

Furthermore, my analyses unveiled one *logically* dead option, 27 features that are dead because of *missing* symbols in their dependencies and 28 features that are false optional. I used an adaption of the algorithm to identify logically, referential and false optional features to examine file preconditions and identified 44 globally dead files and several architecture-internal dead files. Finally, I have identified one option that is unused in the source tree of Linux/v4.0.

6 DISCUSSION

The evaluation yields that my approaches are able to answer the questions I raised in Section 2.8. I was able to identify dead options in KCONFIG three different flavours: *logically* dead, *missing* dead and *unreachable* dead. False optional features were identified through the *undead* analysis and other analyses yield gratuitous statements in KCONFIG or find potentially dangerous selects on symbols with dependencies that can influence later build stages. In this chapter, I want to discuss the implications of those findings.

6.1 Use Cases

There are several possibilities to use the tools I have implemented. The rule-based approaches are implemented in `python` and are therefore easily integrated into UNDERTAKER–CHECKPATCH. While the *unreachable* analysis and the checks to identify unnecessary selects on choice values provide a valuable contribution for developers, the checks for selects on symbols with dependencies, however, might create too much noise to be useful since 80 percent of all selects are selects on symbols with dependencies. Another possibility to use these checks is the script I have implemented or they could be incorporated into the tool that aggregates the conditions stored in the intermediate model to the variability model, which is also implemented in `python`. This tool already has the necessary information during the aggregation process, but I think a integration into UNDERTAKER–CHECKPATCH contributes more to a useful identification.

However, the calculation time of the model consistency analysis is too high to integrate it into UNDERTAKER–CHECKPATCH, but an analysis through the UNDERTAKER could be integrated into the daily analysis of linux-next done by Ruprecht [Rup15].

6.2 About Unreachable Symbols

The analyses in Section 5.1.1 lead ultimately to the question: Why are *unreachable* symbols in the kernel? I believe there are three reasons for the presence of these symbols in the kernel [24]:

[24] However, I do not rule out I might have missed some.

❶ **Unintentionally:**
Removed selects or defaults might render a feature *unreachable*, without being noticed.

❷ **Intentionally:**
Unreachables might be features that are not fully implemented and are therefore *unreachable* by intention.

❸ **Process related:**
Unreachable symbols might be used in downstream kernels[25].

The first ❶ reason for *unreachable* features in the large code base of LINUX is that code and features become obsolete and developers remove them. When they remove features with selects that are only enabled by this select, these features become *unreachable* and the corresponding code becomes dead and might stay in the kernel unnoticed.

Secondly ❷, I assume that symbols are intentionally *unreachable*. A well known intentionally *unreachable* symbol is `BROKEN`. This symbol is often used in dependencies to mark a specific feature as defective, sometimes in general and sometimes only for specific architectures where conflicts occur. However, it is difficult to prove that other *unreachable* symbols are intentionally *unreachable*. One indicator is the date of the commit. When the commit is fairly recent, the argument that this feature is not yet completely implemented is at hand and one or more features to select the *unreachable* one might be in the queue of the maintainer. The development in LINUX often consists of many small patches, sometimes patch-series, where some patches might take longer to be merged and others might be integrated faster. It makes sense to leave code disabled to avoid conflicts or erroneous behaviour until the extension is fully implemented. Those *unreachable* symbols might therefore be remainders of the product line evolution within LINUX.

The third ❸ reason for *unreachable* features could be, that LINUX distributions often adapt software to their needs, add drivers or other patches that are not (yet) in the kernel because they consider the review process to exhausting or just do not want their code merged in the kernel. Other architectures (e.g. `blackfin`) often have their own repositories and often do not work upstream (i.e. submit their patches to the LINUX kernel). This process regularly leaves the state of some drivers or features incomplete when developers do not have the time or companies do not have the money to pursue their patches, adapt them to the comments of reviewers or maintainers to get them merged. Other developers implement or extend features to simplify downstream development.

6.3 Unnecessary Selects on Choice Values

It is hard to tell if these defects lead to a functional impairment. While constructing the FM, developers inserted the select statement to enable another option. When they assume that this choice value is defined, according to the select statement they introduced, and write their code appropriately, they might introduce code that causes a functional defects or is simply dead. These defects probably originate in the lack of insight into KCONFIG details of developers.

[25] Downstream kernels are derived products of the LINUX kernel with local modifications (e.g. modifications by distributions which are not (yet) merged in the usual LINUX kernel). An example for this behaviour is the `Ubuntu` distribution.

6.4 Logical and Referential Dead Symbols

The number of *logical* and *referential* defects in LINUX is low. I assume that this means for *logical* defects that they are either caught during the review process most of the time or people recognise that the feature they rely on is dead and they investigate and solve the problem. However, *referential* defects might have similar origins as I have described for *unreachable* symbols (❶ ❷ and ❸). They might have been introduced unintentionally while removing old features, they might be used as a way to implement product line evolution or the features depend on other features that are only introduced in downstream kernels.

6.5 False Optional Features

False optional features in KCONFIG manifest in different ways (as described in Section 5.2.3). While the resolution of those defects might improve the comprehensibility of the declared variability, it is nearby that developers introduced these connections to imply a form of dependence, even if both symbols are always enabled together.

6.6 Other Investigated and Identified Problems

KCONFIG was developed with the needs of the LINUX kernel in mind and often chose a fast pragmatic approach over usability and sometimes consistency (e.g. selects on symbols with dependencies). This lead to several unobvious usability and consistency issues, which I want to discuss in this section, along with other problems I have investigated while researching consistency issues in KCONFIG.

One peculiarity of KCONFIG is that the graphical interface to configure the kernel only shows options whose dependencies are currently satisfied. While this is perfectly reasonable to ensure consistency through the visibility of symbols, this might confuse users: “Which set of options do I have to enable option X?”. The answer to this question is often far from trivial: when users search for their desired option, they might get a set of options which they have to enable to uncover their option. But the features in the listed set might also have dependencies that have to be satisfied to reveal exactly that feature. With 14541 features in Linux/`v4.0` this can be a tedious task and most users do not attempt to create a small configuration from scratch[26]. However, the mentioned search functionality has its own peculiarities. When a symbol has multiple definitions, the contents of these definition are often mixed up and there is no clean distinction between each definition. Even if there was a clean distinction between these definitions, users also might not be aware of

[26]To tackle that problem, Tartler et al. [Tar+12a] implemented a toolchain to trace and tailor a kernel to the specific needs of users. This is achieved by compiling an instrumented kernel, trace which functions are used and then try to create a configuration that covers all used functions. Heinloth [Hei14] and Ruprecht, Heinloth, and Lohmann [RHL14] later refined this toolchain to be more precise and to require less manual adaption by users. However, these are research tools that have not yet reached maturity or scalability on stock hardware. Another attempt to yield a small working configuration as configuration starting point is possible with Linux/`v3.17-rc1` or later is the integration of the `tinyconfig` target in LINUX's Makefiles. These modifications were done by the kernel tinification project (`https://tiny.wiki.kernel.org/`).

the semantics of multiple definitions (described in Subsection 2.5.1). This search function also displays the conditions in the `dependency` or `selected by` sections in propositional logic, which might be unknown to users. These conditions are always displayed in one line in at least one of the configuration tools (`mconf`). This property leads to the concealment of parts of the formula when an option has too many conditions in the `dependency` or `selected by` fields, because the tool does not wrap over long lines.

Most of the listed problems could be avoided, or at least alleviated, by providing better tool support, where the tools aggregate the conditions properly and then display the necessary information.

Less easy to solve are impairments in usability which are a direct consequence of unexpected behaviour. Users might not be aware of the select semantics, fond of propositional logic or understand the peculiarities of the `tristate` logic (briefly explained as additional information in Subsection 2.5.4.1). Multiple definitions of symbols with different dependencies and prompt texts might also be confusing to users.

7 CONCLUSION

Variability in LINUX is not only spread over different levels of the build process but also causes various types of defects that can manifest in every phase of the build process or even at run time. Identifying these defects per hand proved to be difficult, if not impossible. Therefore comprehensive tool support is needed to help developers to manage variability, which is a ever-growing and inevitable part of the kernel.

Previous works focused on the identification of defects in source files with the help of a holistic view of the variability but did not analyse the configuration space alone and thus did not do a holistic analysis of defects in the configurability in LINUX. In this thesis I presented approaches to identify additional defects in this kernel and close this gap. These approaches are able to diagnose contradictions in symbols' dependencies, determine violations of referential integrity or spot gratuitous attributes and interactions of symbols in KCONFIG. Furthermore, I have identified many violations of KCONFIG's own rules and investigated the impact of the newly identified *unreachable* symbols on analyses coined by Tartler and Sincero. This identification lead to a optimisation of the creation process of partial configurations for the coverage analysis which can be used to optimise the test coverage of static analysis tools. The results of my thesis are partially integrated into the new UNDERTAKER–CHECKPATCH tool and thus help to identify and eliminate more defects in early stages of the development process. Additionally, I have discussed usability issues of KCONFIG and explained several quirks in this configuration toolchain which might surprise developers or lead to unexpected behaviour. Moreover, I used an adapted form of the algorithm to identify defective options in KCONFIG to diagnose file presence conditions and thus search for dead and undead files in LINUX.

Future Work

The quality of the results of the analyses to identify dead or false optional features in KCONFIG depends on the accuracy of the model which represents the modelled variability. Even though the model helps to identify many variability defects, it currently has several identified flaws which should be addressed in future work to not only improve the accuracy of the newly introduced analyses but

also the analyses introduced by Sincero [Sin13] and Tartler [Tar13]. Furthermore, the run time of the `mips`, `arm` and `powerpc` architectures stood out negatively for the newly introduced model consistency analysis. For future work, the origins of this difference in run time could be investigated, as well as optimisations in the presented algorithms to reduce this calculation time beyond further parallelisation.

Moreover, there could be other defects that could be identified through extensions of my approaches that KCONFIG and its tools do not recognise yet. Another point that could be researched is the occurrence of *unreachable* or *missing* defects. They could be tracked over the years of LINUX development and evaluated if they are mostly utilised for product line evolution, with the exceptions of a few mistakes the developers made, and disappear in later version.

Another additional research topic would be the adaption of the approaches I presented to other projects like BUSYBOX, COREBOOT or FIASCO that use KCONFIG as their language to model their variability. The UNDERTAKER toolchain already supports these projects and the variability models are similar to the models of LINUX.

Appendices

A COVERAGE DATA

Architecture	# comp. files (total)	Total kLOC	Total configs	in `#ifdef` blocks	# variation points (dead/undead rate)		allyes CC	allyes CC_N	VAMPYR CC	VAMPYR CC_N
mean	12,873.5 (16,165)	10,055.8		4.4%	37,733.3	(24%)	53.7%	68.9%	62%	79.6%
hardware	9,689.5 (12,896)	7,697.1		3.6%	28,011.3	(29%)	47%	64.9%	55.1%	76.1%
software	3,184 (3,269)	2,358.9		7.1%	9,722	(8%)	72.9%	77.8%	81.7%	87.4%
arm	14,870 (16,895)	11,122.4	17399	4.4%	39,701	(18%)	59.9%	71.1%	72.5%	85.9%
hardware	11,646 (13,626)	8,725	13577	3.8%	29,979	(20%)	55.4%	67.6%	69.6%	84.9%
software	3,224 (3,269)	2,397.6	3822	6.8%	9,722	(10%)	73.9%	80.4%	81.4%	88.7%
mips	12,618 (16,147)	9,968.4	14852	4.4%	37,434	(24%)	40.7%	52.3%	52.6%	67.7%
hardware	9,387 (12,878)	7,554.9	11012	3.5%	27,712	(30%)	30.2%	42.4%	41.9%	58.7%
software	3,231 (3,269)	2,413.5	3840	7.2%	9,722	(6%)	70.3%	73.2%	83.1%	86.7%
s390	10,335 (15,551)	8,283.5	12095	4.2%	35,927	(37%)	49%	75.6%	51.9%	80.3%
hardware	7,306 (12,282)	6,083.9	8499	3.2%	26,205	(46%)	40.5%	72.7%	41.9%	75.4%
software	3,029 (3,269)	2,199.7	3596	7%	9,722	(13%)	71.9%	80.6%	78.8%	88.4%
x86	13,671 (16,067)	10,849.2	16039	4.5%	37,871	(18%)	64.5%	77%	69.7%	83.4%
hardware	10,419 (12,798)	8,424.6	12178	3.7%	28,149	(23%)	60.7%	76.9%	65%	82.4%
software	3,252 (3,269)	2,424.7	3861	7.2%	9,722	(5%)	75.3%	77.3%	83.4%	85.7%

Table A.1 – Comprehensive coverage data for Linux v4.0, commit id `f68c461e`: rate (defect corrected) of covered variation points for `allyesconfig` and for the VAMPYR approach. This experiment was run with model version 1.0 and undertaker revision refs/changes/59/4659/40 and coverage algorithm min. The values in this table represent the results *without* the consideration of *unreachable* symbols.

Architecture	# comp. files (total)	Total kLOC	Total configs	in `#ifdef` blocks	# variation points (dead/undead rate)		allyes CC	allyes CC_N	VAMPYR CC	VAMPYR CC_N
mean	12,593 (16,165)	9,881.9		4.3%	37,733.3	(26%)	53.7%	70.8%	65.6%	86.7%
hardware	9,434 (12,896)	7,536.9		3.5%	28,011.3	(32%)	47%	67%	59.8%	85.4%
software	3,159 (3,269)	2,344.9		6.9%	9,722	(11%)	72.9%	79.2%	82.1%	89.5%
arm	14,792 (16,895)	11,075.1	17283	4.4%	39,701	(19%)	59.9%	71.8%	72.5%	86.8%
hardware	11,585 (13,626)	8,686.5	13498	3.8%	29,979	(21%)	55.4%	68.2%	69.6%	85.7%
software	3,207 (3,269)	2,388.7	3785	6.6%	9,722	(12%)	73.9%	81.5%	81.4%	90%
mips	12,475 (16,147)	9,874.5	14639	4.3%	37,434	(25%)	40.7%	53.2%	67.1%	88%
hardware	9,275 (12,878)	7,482.7	10867	3.4%	27,712	(31%)	30.2%	43.1%	60.9%	86.6%
software	3,200 (3,269)	2,391.9	3772	6.9%	9,722	(9%)	70.3%	75.2%	84.9%	91%
s390	9,557 (15,551)	7,811.5	11063	3.9%	35,927	(43%)	49%	82.9%	51.9%	88.1%
hardware	6,575 (12,282)	5,633.7	7554	2.8%	26,205	(53%)	40.5%	82.9%	41.9%	86%
software	2,982 (3,269)	2,177.8	3509	6.8%	9,722	(16%)	71.9%	83%	78.8%	91.2%
x86	13,548 (16,067)	10,766.1	15872	4.4%	37,871	(19%)	64.5%	77.8%	69.8%	84.4%
hardware	10,301 (12,798)	8,344.8	12028	3.6%	28,149	(24%)	60.7%	77.9%	65.1%	83.6%
software	3,247 (3,269)	2,421.4	3844	7.2%	9,722	(6%)	75.3%	77.6%	83.4%	86.2%

Table A.2 – Comprehensive coverage data for Linux v4.0, commit id `f68c461e`: rate (defect corrected) of covered variation points for `allyesconfig` and for the VAMPYR approach. This experiment was run with model version 1.0 and undertaker revision refs/changes/87/5087/7 and coverage algorithm min. The values in this table represent the results *with* the consideration of *unreachable* symbols.

B DEAD FILES

The representation of the file presence conditions in the models is similar to the representation of options, except that the *normalized* filename is used as a key on the left side of the implication instead of the symbol. The normalization process replaces all characters matching the following RegExp [-+:,/] with a "_" character in order to avoid conflicts in the parser of propositional formulas and adds a FILE_-prefix.

B.1 Global Dead Files (44)

```
FILE_drivers_clocksource_fsl_ftm_timer.c   logical: all
FILE_drivers_edac_i82443bxgx_edac.c        logical: all
FILE_drivers_gpio_gpio_mb86s7x.c           missing: all
FILE_drivers_infiniband_hw_ipath_ipath_wc_ppc64.c   logical: all
FILE_drivers_iommu_msm_iommu.c             logical: all
FILE_drivers_iommu_msm_iommu_dev.c         logical: all
FILE_drivers_macintosh_via_maciisi.c       logical: all
FILE_drivers_media_platform_marvell_ccic_mmp_driver.c   logical: all
FILE_drivers_media_rc_ir_rx51.c            logical: (um, score, arm)...
FILE_drivers_mfd_tps65911_comparator.c     logical: all
FILE_drivers_misc_spear13xx_pcie_gadget.c  logical: all
FILE_drivers_net_wireless_b43_phy_ac.c     logical: all
FILE_drivers_net_wireless_b43_phy_lcn.c    logical: all
FILE_drivers_net_wireless_b43_tables_phy_lcn.c          logical: all
FILE_drivers_pcmcia_rsrc_iodyn.c     logical: (s390)  | missing: rest
FILE_drivers_pinctrl_meson_pinctrl_meson.c      logical: all
FILE_drivers_pinctrl_meson_pinctrl_meson8.c     logical: all
FILE_drivers_pps_generators_pps_gen_parport.c   logical: all
FILE_drivers_regulator_max77843.c          missing: all
FILE_drivers_regulator_mt6397_regulator.c  missing: all
FILE_drivers_regulator_sky81452_regulator.c     missing: all
FILE_drivers_scsi_NCR_D700.c   logical: (powerpc) | missing: rest
```

```
FILE_drivers_scsi_NCR_Q720.c   logical: (powerpc) | missing: rest
FILE_drivers_scsi_eata_pio.c   logical: all
FILE_drivers_scsi_sni_53c710.c missing: all
FILE_drivers_staging_board_board.c         logical: all
FILE_drivers_staging_board_kzm9d.c         logical: all
FILE_drivers_tty_serial_etraxfs_uart.c     logical: rest  ...
FILE_drivers_tty_serial_sprd_serial.c      logical: (um, score)...
FILE_drivers_usb_musb_cppi_dma.c           logical: all
FILE_drivers_usb_musb_da8xx.c              logical: all
FILE_drivers_usb_musb_davinci.c            logical: all
FILE_drivers_video_fbdev_omap2_dss_rfbi.c  logical: all
FILE_drivers_xen_xen_acpi_cpuhotplug.c     logical: all
FILE_drivers_xen_xen_acpi_memhotplug.c     logical: all
FILE_drivers_xen_xen_stub.c                logical: all
FILE_kernel_locking_rtmutex_tester.c       logical: all
FILE_sound_soc_au1x_i2sc.c                 logical: all
FILE_sound_soc_davinci_davinci_i2s.c       logical: all
FILE_sound_soc_samsung_smdk_wm8580pcm.c    logical: (um, score, m68k)...
FILE_sound_soc_sh_ssi.c                    logical: all
FILE_sound_soc_spear_spdif_in.c            logical: all
FILE_sound_soc_spear_spdif_out.c           logical: all
FILE_sound_soc_spear_spear_pcm.c           logical: all
```

B.2 Architecture-Internal Defective Files

```
Arch: arm 5
FILE_arch_arm_mach_imx_devices_platform_imx21_hcd.c  logical
FILE_arch_arm_mach_imx_devices_platform_pata_imx.c   logical
FILE_arch_arm_mach_s3c24xx_setup_spi.c  logical
FILE_arch_arm_mm_proc_arm1020e.S        missing
FILE_arch_arm_plat_omap_debug_leds.c    logical

Arch: unicore32 9
FILE_arch_unicore32_kernel_clock.c      undead
FILE_arch_unicore32_kernel_irq.c        undead
FILE_arch_unicore32_kernel_puv3_core.c  undead
FILE_arch_unicore32_kernel_time.c       undead
FILE_arch_unicore32_mm_alignment.c      undead
FILE_arch_unicore32_mm_cache_ucv2.S     undead
FILE_arch_unicore32_mm_dma_swiotlb.c    undead
FILE_arch_unicore32_mm_proc_ucv2.S      undead
FILE_arch_unicore32_mm_tlb_ucv2.S       undead
```

```
Arch: tile 7
FILE_arch_tile_kernel_smp.c             undead
FILE_arch_tile_kernel_smpboot.c         undead
FILE_arch_tile_kernel_tlb.c             undead
FILE_arch_tile_lib_spinlock_32.c        undead
FILE_arch_tile_lib_spinlock_64.c        undead
FILE_arch_tile_lib_usercopy_32.S        undead
FILE_arch_tile_lib_usercopy_64.S        undead

Arch: blackfin 1
FILE_arch_blackfin_kernel_ipipe.c       missing

Arch: x86 4
FILE_arch_x86_kernel_cpu_perf_event.c   undead
FILE_arch_x86_kernel_irq_work.c         undead
FILE_arch_x86_kernel_perf_regs.c        undead
FILE_arch_x86_lib_rwsem.S               undead

Arch: avr32 1
FILE_arch_avr32_boot_u_boot_empty.S             undead

Arch: powerpc 4
FILE_arch_powerpc_kernel_of_platform.c          undead
FILE_arch_powerpc_kernel_prom_parse.c           undead
FILE_arch_powerpc_sysdev_fsl_85xx_cache_sram.c  logical
FILE_arch_powerpc_sysdev_fsl_85xx_l2ctlr.c      logical

Arch: ia64 4
FILE_arch_ia64_kernel_ftrace.c          missing
FILE_arch_ia64_kernel_paravirt.c        logical
FILE_arch_ia64_kernel_paravirt_patch.c  logical
FILE_arch_ia64_kernel_paravirtentry.S   logical

Arch: mips 5
FILE_arch_mips_kernel_8250_platform.c   logical
FILE_arch_mips_kernel_irq_rm7000.c      logical
FILE_arch_mips_kernel_r6000_fpu.S       logical
FILE_arch_mips_mm_tlb_r8k.c             logical
FILE_arch_mips_pci_pci_rt2880.c         missing

Arch: mn10300 8
FILE_arch_mn10300_kernel_gdb_io_serial.c        missing
```

```
FILE_arch_mn10300_kernel_gdb_io_serial_low.S    missing
FILE_arch_mn10300_kernel_gdb_io_ttysm.c         missing
FILE_arch_mn10300_kernel_gdb_io_ttysm_low.S     missing
FILE_arch_mn10300_kernel_gdb_low.S      missing
FILE_arch_mn10300_kernel_gdb_stub.c     missing
FILE_arch_mn10300_kernel_profile.c      missing
FILE_arch_mn10300_kernel_profile_low.S  missing

Arch: sh 1
FILE_arch_sh_kernel_iomap.c             undead

Arch: openrisc 1
FILE_arch_openrisc_kernel_prom.c        undead

Arch: hexagon 1
FILE_arch_hexagon_kernel_stacktrace.c   undead

Arch: m32r 5
FILE_arch_m32r_mm_cache.c               undead
FILE_arch_m32r_mm_extable.c             undead
FILE_arch_m32r_mm_init.c                undead
FILE_arch_m32r_mm_mmu.S                 undead
FILE_arch_m32r_mm_page.S                undead
```

LIST OF ACRONYMS

API	application programming interface
CPP	C preprocessor
LKM	loadable kernel module
RegExp	regular expression
SAT	(boolean) satisfiability problem
SPL	Software Product Line
CC	Configuration Coverage
FM	Feature Model
DSL	domain specific language
MUS	minimal unsatisfiable subset

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

REFERENCES

[ABW14] Iago Abal, Claus Brabrand, and Andrzej Wasowski. “42 Variability Bugs in the Linux Kernel: A Qualitative Analysis.” In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE ’14. Vasteras, Sweden: ACM, 2014, pp. 421–432. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2642990. URL: http://doi.acm.org/10.1145/2642937.2642990.

[Bat05] Don Batory. “Feature Models, Grammars, and Propositional Formulas.” English. In: *Software Product Lines*. Ed. by Henk Obbink and Klaus Pohl. Vol. 3714. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 7–20. ISBN: 978-3-540-28936-4. DOI: 10.1007/11554844_3. URL: http://dx.doi.org/10.1007/11554844_3.

[Ber+10] Thorsten Berger et al. “Feature-to-code mapping in two large product lines.” In: *Proceedings of the 14th Software Product Line Conference (SPLC ’10)*. (Jeju Island, South Korea). Ed. by Kyo Kang. Vol. 6287. Lecture Notes in Computer Science. Poster session. Heidelberg, Germany: Springer-Verlag, Sept. 2010, pp. 498–499. ISBN: 978-3-642-15578-9.

[Ber+12] Thorsten Berger et al. “Variability modeling in the systems software domain.” In: (2012).

[Bie08] Armin Biere. “PicoSAT essentials.” In: *Journal on Satisfiability, Boolean Modeling and Computation* 4 (2008), pp. 75–97.

[Bru03] Renato Bruni. “Approximating minimal unsatisfiable subformulae by means of adaptive core search.” In: *Discrete Applied Mathematics* 130.2 (2003), pp. 85–100.

[BS05] James Bailey and PeterJ. Stuckey. “Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization.” English. In: *Practical Aspects of Declarative Languages*. Ed. by ManuelV. Hermenegildo and Daniel Cabeza. Vol. 3350. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 174–186. ISBN: 978-3-540-24362-5. DOI: 10.1007/978-3-540-30557-6_14. URL: http://dx.doi.org/10.1007/978-3-540-30557-6_14.

[BSW03] Maria Garcia de la Banda, Peter J. Stuckey, and Jeremy Wazny. "Finding All Minimal Unsatisfiable Subsets." In: *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*. PPDP '03. Uppsala, Sweden: ACM, 2003, pp. 32–43. ISBN: 1-58113-705-2. DOI: 10.1145/888251.888256. URL: http://doi.acm.org/10.1145/888251.888256.

[Die+12] Christian Dietrich et al. "A Robust Approach for Variability Extraction from the Linux Build System." In: *Proceedings of the 16th Software Product Line Conference (SPLC '12)*. (Salvador, Brazil, Sept. 2–7, 2012). Ed. by Eduardo Santana de Almeida, Christa Schwanninger, and David Benavides. New York, NY, USA: ACM Press, 2012, pp. 21–30. ISBN: 978-1-4503-1094-9. DOI: 10.1145/2362536.2362544.

[Doc] *Kconfig*. the Linux kernel documentation. URL: https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt (visited on 04/28/2015).

[Hac13] Ralf Hackner. "Konfigurationsanalyse mit SAT Lösern in Linux/Kconfig." Diploma Thesis. University of Erlangen, Dept. of Computer Science, 2013.

[Hei14] Bernhard Heinloth. "Automatic Tailoring of the Multi-Purpose Linux Operating System on Embedded Devices." Master's Thesis. University of Erlangen, Dept. of Computer Science, 2014. URL: https://www4.cs.fau.de/Ausarbeitung/MA-I4-2014-09-Heinloth.pdf.

[Kan+90] Kyo C Kang et al. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. DTIC Document, 1990.

[LB13] Jean-Marie Lagniez and Armin Biere. "Factoring out assumptions to speed up MUS extraction." In: *Theory and Applications of Satisfiability Testing–SAT 2013*. Springer, 2013, pp. 276–292.

[LS08] MarkH. Liffiton and KaremA. Sakallah. "Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints." English. In: *Journal of Automated Reasoning* 40.1 (2008), pp. 1–33. ISSN: 0168-7433. DOI: 10.1007/s10817-007-9084-z. URL: http://dx.doi.org/10.1007/s10817-007-9084-z.

[NC01] Linda Northrop and Paul Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. ISBN: 978-0-201-70332-0.

[Par72] David Lorge Parnas. "On the Criteria to be used in Decomposing Systems into Modules." In: *Communications of the ACM* (Dec. 1972), pp. 1053–1058.

[RHL14] Andreas Ruprecht, Bernhard Heinloth, and Daniel Lohmann. "Automatic Feature Selection in Large-Scale System-Software Product Line." In: *13th International Conference on Generative Programming and Component Engineering (GPCE '14)*. New York, NY, USA: ACM Press, 2014, pp. 39–48. DOI: 10.1145/2658761.2658767. URL: https://www4.cs.fau.de/Publications/2014/ruprecht_14_gpce.pdf.

[Rin+14] L.F. Rincón et al. “An Ontological Rule-Based Approach for Analyzing Dead and False Optional Features in Feature Models.” In: *Electronic Notes in Theoretical Computer Science* 302.0 (2014). Proceedings of the {XXXIX} Latin American Computing Conference (CLEI 2013), pp. 111 –132. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2014.01.023. URL: http://www.sciencedirect.com/science/article/pii/S1571066114000243.

[Rot14] Valentin Rothberg. “Years of Variability Bugs in Linux – How to Avoid them.” Master’s Thesis. University of Erlangen, Dept. of Computer Science, 2014. URL: https://www4.cs.fau.de/Ausarbeitung/MA-I4-2014-07-Rothberg.pdf.

[Rup15] Andreas Ruprecht. “Lightweight Extraction of Variability Information from Linux Makefiles.” Master’s Thesis. University of Erlangen, Dept. of Computer Science, 2015. URL: https://www4.cs.fau.de/Ausarbeitung/MA-I4-2015-01-Ruprecht.pdf.

[Sin+07] Julio Sincero et al. “Is The Linux Kernel a Software Product Line?” In: *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*. (Kyoto, Japan). Ed. by Frank van der Linden and Björn Lundell. 2007.

[Sin+10] Julio Sincero et al. “Efficient Extraction and Analysis of Preprocessor-Based Variability.” In: *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE ’10)*. (Eindhoven, The Netherlands). Ed. by Eelco Visser and Jaakko Järvi. New York, NY, USA: ACM Press, 2010, pp. 33–42. ISBN: 978-1-4503-0154-1. DOI: 10.1145/1868294.1868300.

[Sin13] Julio Sincero. “Variability Bugs in System Software.” PhD thesis. Friedrich-Alexander University Erlangen-Nuremberg, 2013.

[Tar+11a] Reinhard Tartler et al. “Configuration Coverage in the Analysis of Large-Scale System Software.” In: *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS ’11)*. (Cascais, Portugal). Ed. by Eric Eide et al. New York, NY, USA: ACM Press, 2011, 2:1–2:5. ISBN: 978-1-4503-0979-0. DOI: 10.1145/2039239.2039242.

[Tar+11b] Reinhard Tartler et al. “Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem.” In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys ’11)*. (Salzburg, Austria). Ed. by Christoph M. Kirsch and Gernot Heiser. New York, NY, USA: ACM Press, Apr. 2011, pp. 47–60. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966451.

[Tar+12a] Reinhard Tartler et al. “Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability.” In: *Proceedings of the 8th International Workshop on Hot Topics in System Dependability (HotDep ’12)*. (Los Angeles, CA, USA). Berkeley, CA, USA: USENIX Association, 2012, pp. 1–6.

[Tar+12b] Reinhard Tartler et al. “Configuration Coverage in the Analysis of Large-Scale System Software.” In: *ACM SIGOPS Operating Systems Review* 45.3 (Jan. 2012), pp. 10–14. ISSN: 0163-5980. DOI: 10.1145/2094091.2094095.

[Tar+14] Reinhard Tartler et al. “Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue.” In: *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX 2014)*. Ed. by USENIX Association. Philadelphia, PA, USA, 2014, pp. 421–432. ISBN: 978-1-931971-10-2. URL: http://www4.cs.fau.de/Publications/2014/tartler_14_usenix.pdf.

[Tar13] Reinhard Tartler. “Mastering Variability Challenges in Linux and Related Highly-Configurable System Software.” PhD thesis. Erlangen: Friedrich-Alexander-Universität Erlangen-Nürnberg, 2013.

[Web] *VAMOS: Variability Management in Operating Systems*. Research Group Homepage. URL: https://www4.cs.fau.de/Research/VAMOS/ (visited on 07/04/2015).

[WP01] A.W. Williams and R.L. Probert. “A measure for component interaction test coverage.” In: *Computer Systems and Applications, ACS/IEEE International Conference on. 2001*. 2001, pp. 304–311. DOI: 10.1109/AICCSA.2001.934001.