FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

**Andreas Ruprecht**

# Lightweight Extraction of Variability Information from Linux Makefiles

Masterarbeit im Fach Informatik

April 27, 2015

Please cite as:
Andreas Ruprecht, “Lightweight Extraction of Variability Information from Linux Makefiles”, Master’s Thesis, University of Erlangen, Dept. of Computer Science, 2015.

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

# Abstract

In Linux `v3.19`, the configuration system KCONFIG offers more than fourteen thousand configurable options that allows users to build a kernel specific to their respective needs. As these configurable options span across all layers involved in the kernel build process (KCONFIG, the build system KBUILD, and the code), manual checking for inconsistencies caused by these options is close to impossible.

Recently, Valentin Rothberg presented UNDERTAKER-CHECKPATCH, a tool based on the UNDERTAKER toolchain which allows developers to quickly check their proposed patches for errors introduced by the use of configurable options. Unfortunately, the tool currently can not incorporate information from KBUILD, as the extractor needs several hours to compute the variability data.

In this thesis, I therefore present a parsing-based approach to extract variability information from Makefiles. Contrary to earlier assumptions, I show that my solution is able to extract highly accurate conditions from KBUILD, and that it is able to work robustly across a wide range of Linux versions while slashing the runtime to just over one second on Linux version `v3.19`. This, for the first time, allows us to run an accurate, daily analysis of the commits sent into the `linux-next` development tree, and to have very fast response times to the authors of the buggy commits.

Through its modular design, my tool can easily be enhanced to support other software projects, which I demonstrate by providing plug-in modules for BUSYBOX and COREBOOT.

# Kurzfassung

KCONFIG, das Konfigurationssystem des Linux-Betriebssystemkerns, bietet in Linux `v3.19` mehr als vierzehntausend konfigurierbare Optionen an, die es Nutzern erlauben, den Kern genau auf ihre jeweiligen Bedürfnisse anzupassen. Da die konfigurierbaren Optionen in allen Ebenen des Build-Prozesses (in KCONFIG selbst, im Build-System KBUILD und im Quelltext) verwendet werden, ist eine manuelle Überprüfung auf Inkonsistenzen, die durch diese Optionen entstehen können, nahezu unmöglich.

Valentin Rothberg stellte vor kurzem UNDERTAKER-CHECKPATCH vor, ein auf der UNDERTAKER-Toolkette basierendes Software-Programm, das es Entwicklern ermöglicht, ihre Änderungen schnell auf Fehler hin zu testen, die durch die Verwendung konfigurierbarer Optionen hervorgerufen werden. Momentan kann das Programm jedoch keine Informationen aus KBUILD in die Analyse integrieren, da der Extraktor einige Stunden benötigt, um die Variabilitätsdaten zu berechnen.

In dieser Arbeit präsentiere ich daher einen textbasierten Ansatz zur Extraktion von Variabilitätsinformationen aus Makefiles. Entgegen früherer Annahmen zeige ich, dass durch einen solchen Ansatz sehr präzise Bedingungen aus KBUILD extrahiert werden können, und dass der Ansatz auch über eine Vielzahl von Linux-Versionen hinweg zuverlässig funktioniert. Die Laufzeit des neuen Extraktors beträgt für Linux `v3.19` jedoch nur knapp mehr als eine Sekunde. Dies erlaubt es uns zum ersten Mal, täglich eine detaillierte Analyse der Änderungen, die in den Entwicklungszweig des Linux-Kernes eingebracht werden, durchzuführen und den Entwicklern schnell Rückmeldung geben zu können.

Durch den modularen Aufbau kann mein Extraktor leicht um die Unterstützung anderer Software-Projekte erweitert werden, was ich beispielhaft an Erweiterungs-Modulen für BUSYBOX und COREBOOT demonstriere.

## 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 vorgele gen 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 15. April 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 April 15th, 2015.

(Andreas Ruprecht)
Erlangen, 27. April 2015

# Contents

# 1 Introduction

Linux, as well as other system software, offers a great deal of static configurability to tailor it with respect to a specific application or hardware platform. Linux 3.19, for instance, contains more than fourteen thousand configurable features, defined by its KCONFIG variability model and associated tools. Technically, the implementation of all these features is spread over multiple levels of the software generation process, including the configuration system itself, the build system KBUILD, C preprocessor, the GCC compiler, the linker, and more. This enormous variability has become unmanageable in practice; in the case of Linux it already has led to thousands of *variability defects* [21], that is, bugs and other quality issues related to the implementation of variable features. These defects emerge as dead or undead `#ifdef`-blocks in the code – seemingly configuration-conditional code that can, however, never be selected or deselected.

As part of the VAMOS [23]/CADOS [5] project, Sincero [18] and Tartler [20] have developed the UNDERTAKER toolchain, which can extract the variability information from KCONFIG and search for dead and undead blocks in the code of Linux. Extending their work to also incorporate the variability information expressed in the build system, Dietrich et al. [9] presented an approach for the extraction of the configurational constraints for individual source files. In the KBUILD Makefiles, developers can specify that files should only be built if the corresponding configurable features have been selected in the configuration step. As the underlying MAKE language allows arbitrarily complex logic to be used in order to express these dependencies, their extractor, GOLEM, does not try to gather the data directly from the Makefiles, but rather *probes* the build system in a systematic way. By using KBUILD as a black box, GOLEM is particularly robust with respect to changes in the Makefiles across different architectures and versions.

The probing strategy – switching relevant configuration options on and off, and observing the changes in regards of files which will be built – however, has one

big disadvantage: its high runtime. For the most recent Linux version `v3.19`, the extraction process takes over three hours for the `x86` architecture alone; as Linux now supports 30 different architectures, it would take days to extract all required constraints for an analysis of all defects in Linux. While this high runtime might still be acceptable for research through an off-line analysis of single stable versions, Rothberg [17] recently developed UNDERTAKER-CHECKPATCH to foster the use of the UNDERTAKER toolchain in production; with his tool, Linux developers can easily check if their patch introduces any variability defects before sending it out to the maintainers. In this scenario, waiting a few days for the result of the analysis of a patch, howsoever small, is not acceptable. On the other hand, using the data from KBUILD greatly increases the number of defects correctly detected by UNDERTAKER.

In this thesis, I therefore present MINIGOLEM, a parsing-based extractor for variability information from Linux Makefiles which allows the collection of data from KBUILD in just over one second. Contrary to the concerns expressed by Dietrich et al. [9] over the feasibility of a text-based evaluation of Makefiles, the tool is able to robustly extract highly accurate data for all Linux versions released during the last five years. Through its modular design, it can further be easily adapted to support other system software projects, which I demonstrate by providing implementations for the build system of BUSYBOX and COREBOOT.

The remainder of the thesis is structured as follows: In Chapter 2, I will outline the concepts and tools which are used to implement configurability in the Linux kernel, and how BUSYBOX and COREBOOT adapted them for their own purpose. Furthermore, I will describe how the UNDERTAKER toolsuite uses the extracted variability data to detect and classify defects, and how UNDERTAKER-CHECKPATCH can be used in the workflow of a Linux developer, before presenting related work from other researchers. Next, in Chapter 3, I will elaborate on the challenges which a thorough solution has to face, and subsequently show how my approach is able to tackle them. Chapter 4 then provides an insight into the implementation of my tool, and describes a newly developed experiment which we use to analyze the `linux-next` integration tree on a daily basis. In Chapter 5, I will provide a detailed evaluation regarding the runtime, the robustness, and the accuracy of my tool, covering Linux, BUSYBOX, and COREBOOT.

# 2 Fundamentals

In this chapter, I will present the basic mechanisms and tools which are used to configure and build Linux, and how other system software projects adapted these tools for their own use. Furthermore, I will explain how the current version of the UNDERTAKER tool extracts and uses this information to detect bugs and anomalies, and how UNDERTAKER-CHECKPATCH employs UNDERTAKER to analyze patches. Lastly, I will give an overview over related approaches by other researchers.

## 2.1 Configuring and Building Linux

The way variability is used in the Linux kernel spreads over different, but interlocked layers: Features defined in the configuration system will have effects on other configurable options, on the build system (Makefiles and the compiler) as well as on the source code through the C preprocessor.

To get an insight on how this interaction takes place, I will now describe how configuration options are defined in the configuration system and how they are used throughout the build process.

### 2.1.1 Configuration of the Kernel: KCONFIG

The basic process of configuring a Linux kernel to one's needs begins with a user-defined selection of configuration options (*features*) from a command-line or graphical user interface.

Underlying this process is the KCONFIG language, in which a developer will define the features and possible constraints they might have. An example for such a feature definition can be seen in Listing 2.1. Here, a feature called "`USB_HID`" is defined. It is given a type, *tristate*, which means the corresponding functionality can be statically compiled into the kernel or built as a loadable kernel module (LKM). Other possible

types are *boolean* (the feature will either be statically compiled or not compiled at all), *integer, hex* (to define numeric values like addresses or the number of supported CPU cores) and *string* (for configurable features like the default kernel command line). Additionally, a feature is given a short description which will be visible to the user who is about to select this feature. Line 7 tells us that `USB_HID` depends on two other features, namely `USB` and `INPUT`. Hence, the user will only be able to select this feature if both `USB` and `INPUT` – and their recursive dependencies – are already enabled; if the dependencies can not be fulfilled, the feature is not visible during the configuration process and can thus not be enabled. Additionally, a feature can *select* other features (c.f. line 8), meaning that when the user selects `USB_HID`, the option `HID` will automatically be enabled. In general, the dependencies and selections can be arbitrarily complex boolean expressions and can have conditional guards, thus only triggering a selection if other constraints are met.

```
4 config USB_HID
5     tristate "USB HID transport layer"
6     default y
7     depends on USB && INPUT
8     select HID
9     ---help---
10      Say Y here if you want to connect USB keyboards, mice,
11      joysticks, graphic tablets, or any other HID based devices
12      to your computer via USB, as well as Uninterruptible
13      Power Supply (UPS) and monitor control devices.
14      [...]
15      If unsure, say Y.
16
17      To compile this driver as a module, choose M here: the
18      module will be called usbhid.
```

**Listing 2.1** – An example for a KCONFIG feature definition, taken from `drivers/hid/usbhid/Kconfig` (Linux v3.19).

Once the user has made her selection of desired features, the collected information is stored as a list of key-value pairs in a file called `.config` in the root directory of the kernel source code (see Listing 2.2 for an example).

### 2.1.2 Which Files Will Be Compiled: KBUILD

To determine which source code artifacts have to be compiled, KCONFIG uses the information from `.config`, transforms it into MAKE syntax and saves the result into an auto-generated Makefile at `include/config/auto.conf` which can then be used by KBUILD, the kernel's build system. Additionally, it prefixes the configuration options with the string "`CONFIG_`", so our feature from Listing 2.1 will be called

```
1608 CONFIG_INPUT=y
1609 CONFIG_USB=y
1610 CONFIG_USB_HID=m
1611 CONFIG_HID=y
```

**Listing 2.2** – An excerpt from the `.config` file which is a result of the kernel configuration step and contains all selected options.

`CONFIG_USB_HID` from now on. The generated Makefile contains all selected features as a list of variable definitions which are assigned their respective value from the `.config` file.

Through a pattern called "*Dancing Makefiles*" proposed in 1997[1], the following build process is split up into different Makefiles: While the central build rules are defined in a range of Makefiles in the `scripts/` subdirectory, the selection which files will be compiled is left to simpler Makefiles in the subdirectories which are traversed recursively. A simple example for such a *selecting* Makefile can be seen in Listing 2.3. Note that the generated `auto.conf` is automatically included into every processed Makefile, allowing a developer to use configurable options to guide the inclusion of source files and whole subdirectories.

```
125 obj-$(CONFIG_HID_WALTOP)        += hid-waltop.o
126 obj-$(CONFIG_HID_WIIMOTE)       += hid-wiimote.o
127 obj-$(CONFIG_HID_SENSOR_HUB)    += hid-sensor-hub.o
128
129 obj-$(CONFIG_USB_HID)           += usbhid/
130 obj-$(CONFIG_USB_MOUSE)         += usbhid/
```

**Listing 2.3** – An example for a KBUILD Makefile, taken from `drivers/hid/Makefile` (Linux v3.19).

The selection of files or subdirectories is accomplished by adding them to internal lists (`obj-{y,n,m}`); all object files in `obj-y` will be statically compiled into the kernel image, everything in `obj-m` will be compiled as a LKM, and all object files in `obj-n` will simply not be compiled. If the right hand side of the assignment is a directory, MAKE will descend into that subdirectory if the corresponding configuration option is either `y` or `m`. In our example, MAKE will compile `hid-sensor-hub.c` into `hid-sensor-hub.o`, if the configuration option `CONFIG_HID_SENSOR_HUB` was selected. The information about the variables is accessed by including the information from `include/config/auto.conf`, and the variable `$(CONFIG_HID_SENSOR_HUB)`

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

will be substituted with its selected value, `y` or `m`. Similarly, MAKE will descend into the subdirectory `usbhid/`, if `CONFIG_USB_HID` was enabled. In this subdirectory, MAKE will again look for a Makefile and process it accordingly. As the variability information can only trigger the selection on the level of entire compilation units, we denote KBUILD as the implementation of so-called *coarse-grained variability*.

### 2.1.3 Which Code Will Be Compiled: C Preprocessor

Within the files, a developer can even be more precise and specify single lines of code to be configurable on a very *fine-grained* level.

The *fine-grained variability* is achieved by using conditional preprocessor blocks – `#ifdef` macros – with configuration options. Besides `auto.conf`, the KBUILD system also generates a C header file (`autoconf.h`) which is then included into the compilation process for every source file. Note that special care is taken for *tristate* symbols: as some code might only be desired if the surrounding file is compiled as a LKM (or only if it is not), the header contains an additional symbol with the suffix "`_MODULE`" for every *tristate* symbol (e.g., `CONFIG_USB_HID_MODULE`). This symbol is set when the user selected `'m'` in the configuration step.

```
39 /* for apple IDs */
40 #ifdef CONFIG_USB_HID_MODULE
41 #include "../hid-ids.h"
42 #endif
```

**Listing 2.4** – An example for an `#ifdef` block inside a source file, taken from `drivers/hid/usbhid/usbmouse.c` (Linux v3.19).

An example of such preprocessor use can be seen in Listing 2.4. Here, the `#include` statement in line 41 will only be reached if `CONFIG_USB_HID` was selected as `'m'`, thus only if it is compiled as an LKM.

#### Summary

As described above, the configuration and build process for Linux can be divided into three separate, but inter-dependent steps [17]:

1. In KCONFIG, features and their constraints are defined. Through a user interface, kernel configuration options can be selected and deselected, creating a *configuration variant* in the `.config` file.

2. Based on the selection from step 1, KBUILD (the build system) implements *coarse-grained variability* by selecting translation units for further processing and compilation.

3. Inside the selected source files, the C preprocessor implements *fine-grained variability*. This step uses `#ifdef` macros to include or exclude conditional blocks of code inside the file, leading to only selected parts being passed to the compiler.

## 2.2 Use of KCONFIG and KBUILD in Other Projects

While KCONFIG and KBUILD were "invented" and are mostly developed in the context of the Linux kernel, it is not the only system software project employing them to manage and use configurability. An overview of their usage has been presented by Berger et al. [1] – in this thesis, I focus on two specific instances which have also been analyzed in previous work from our group.

### 2.2.1 The BUSYBOX Tool Suite

BUSYBOX [4] is a collection of the most common UNIX utilities combined inside one executable which is optimized for size, thus aiming at small and embedded systems.

Like Linux, it also uses KCONFIG with the previously described user interfaces as the mechanism behind its configurability. A user can choose compiler flags and other build options as well as select which tools should be included in the resulting image. As the result should be *one* binary, BUSYBOX does not support `tristate` features and LKMs. Additionally, some information about configurable options is generated from the source code before the configuration step. Here, files called `Config.in` form the template. These files contain a line `INSERT` which marks the spot where the generated information should be placed. Inside the source code, a developer can specify additional configuration options by placing textual comments starting with `//config:` in the file. All these lines will be collected by a script called `gen_build_files.sh` which is run before the configuration starts, and are included into a generated `Config` file which will then be evaluated by KCONFIG.

The build process is then instrumented with KBUILD, albeit with some further small modifications compared to Linux:

- All KBUILD files do not use the `obj-{y,n}` lists described earlier, but rely on the prefixes `libs-` to select subdirectories for further processing and `lib-` for files which should be built.

- Like in the configuration step, some information in the KBUILD files is generated from the source code files before the build step. For the build system, the template for the final `Kbuild` file is called `Kbuild.src` while further constraints for the build process can be specified using a comment starting with `//kbuild:` in the source code. When `gen_build_files.sh` is run during the configuration step, it will also search for these lines and replace them with the actual build information from the code. An example for this is shown in Listing 2.5.

```
7  libs-y                  += libcoreutils/
8
9  lib-y:=
10
11 INSERT
12 lib-$(CONFIG_CAL)       += cal.o
```

**(a)** The template `Kbuild.src` file, taken from the `coreutils/` subdirectory.

```
13 [...]
14 //kbuild:lib-$(CONFIG_CAT)      += cat.o
15 //kbuild:lib-$(CONFIG_MORE)     += cat.o # more uses it if ↘
       stdout isn't a tty
16 [...]
```

**(b)** Excerpt from `coreutils/cat.c`.

```
7  libs-y                  += libcoreutils/
8
9  lib-y:=
10
11 lib-$(CONFIG_BASENAME) += basename.o
12 lib-$(CONFIG_CAT)      += cat.o
13 lib-$(CONFIG_MORE)     += cat.o # more uses it if stdout isn't ↘
       a tty
```

**(c)** Generated `Kbuild` file in the `coreutils/` subdirectory

**Listing 2.5** – Example for the generation of KBUILD code in the BUSYBOX tool suite, v.1.24.0.

### 2.2.2 The COREBOOT Project

COREBOOT [6] is an open source project trying to eliminate the need for a proprietary BIOS firmware by providing the same (or similar) hardware initialization logic in open source software. In contrast to the vendor-provided firmware, users can freely configure the firmware and only choose features they need in their specific scenario.

Like Linux and BUSYBOX, COREBOOT uses KCONFIG to manage the configuration options and provide an interface to the user. As the product of the configuration and building step can only be one single firmware image, `tristate` options and LKM support are not used.

As opposed to Linux and BUSYBOX, the build system does not make use of the "Dancing Makefiles" pattern. Instead, subdirectories given in the `subdirs-y` list are simply traversed recursively, while the files which need to be compiled are collected in lists named after the class they belong to, namely `ramstage-y`, `romstage-y`, `bootblock-y`, `smm-y`, `smmstub-y`, `cpu_microcode-y` and `verstage-y` in the most recent version. Again, configurable options from KCONFIG can be used to conditionally include files (see Listing 2.6 for an example). Only after all directories have been traversed – as opposed to Linux and BUSYBOX, where compilation is done on a per-subdirectory basis – the compiler and linker are run.

```
5  ramstage-y += die.c
6
7  smm-y += printk.c
8  smm-y += vtxprintf.c
9  smm-$(CONFIG_SMM_TSEG) += die.c
10
11 romstage-y += vtxprintf.c
12 romstage-$(CONFIG_CACHE_AS_RAM) += console.c
```

**Listing 2.6** – Excerpt of a Makefile used by the COREBOOT build system, taken from `src/console/Makefile.inc`.

Both projects also use the C preprocessor combined with a generated header file from the selected configuration to implement *fine-grained* variability on the `#ifdef` block level.

## 2.3 Defect Analysis With the UNDERTAKER Tool

As the KCONFIG options are used throughout every level of configurability, it becomes hard to manually keep track of all constraints that might hold for a given `#ifdef` block when editing a single source code file. In order to provide tool support to manage configurability and to detect errors caused by configurability in system

software, the VAMOS [23]/CADOS [5] project developed the UNDERTAKER tool (first presented in [19], with a more detailed explanation in [21]). In the following, I will describe how UNDERTAKER uses the variability information from the different granularity levels, and which errors can be detected with this approach.

First, the variability information from KCONFIG is transformed into a model. This model describes all variability defined by KCONFIG – including dependencies and other constraints – in terms of boolean formulas. The conditions from KBUILD can be extracted with the GOLEM tool, and are subsequently added to the model. Lastly, the (possibly nested) structure of `#ifdef` blocks in the source file which we analyze is also transformed into a boolean formula, describing the presence condition for every block, i.e. which configuration options have to be enabled or disabled to include it.

Using these formulas, we can now analyze the structure of variability and detect **symbolic violations** (i.e., references to KCONFIG features which do not exist) or **logic defects** (i.e., contradictions or tautologies in the presence conditions for the code). To identify the cause of a defect, a boolean satisfiability problem (SAT) solver is employed to test if the corresponding boolean formulas can be solved.

Logic defects can further be grouped into different subclasses which are categorized by their origin.

### 2.3.1 Code Defects

To get an understanding for the first defect variant, consider the following example: Let us assume a file has the structure of `#ifdef` blocks as shown in Listing 2.7.

```
1 #ifdef CONFIG_USB_HID
2     // lots of code     (B0)
3 #ifdef CONFIG_USB_HID
4     // some code        (B1)
5 #else
6     // some more code   (B2)
7 #endif
8 #endif
```

**Listing 2.7** – Example structure of `#ifdef` blocks inside a source file.

We can see that the file contains nested `#ifdef` blocks, which both reference the same symbol. When we generate the formulas for the blocks in the file, we find that for the code at `(B0)` to be compiled, we need to fulfill the formula `CONFIG_USB_HID`, for the code at `(B1)` we need `CONFIG_USB_HID && CONFIG_USB_HID` and lastly, for `(B2)` the formula would be `CONFIG_USB_HID && !CONFIG_USB_HID`.

Feeding all these formulas to the SAT solver, we get the result that `(B1)` is always enabled under the precondition that its surrounding (*parent*) block is enabled – it

is thus considered to be **undead**. Correspondingly, (B2) is considered **dead**; the formula for the block contains a contradiction, meaning it can never be enabled. As the tautology and contradiction can already be derived from the code alone – without considering dependencies from KCONFIG – we classify these as **code defects**.

These errors might seem obvious from the minimal example above, but in Linux, we often have to deal with files several thousand lines long which contain dozens of nested `#ifdef` blocks where it is hard to keep track of the full conditions for the line of code a developer is currently changing.[2]

### 2.3.2 KCONFIG and KBUILD Defects

While the errors above were already detectable from looking at the source code alone, there are more difficult, KCONFIG- and KBUILD-related bugs which I would also like to present using an example shown in Listing 2.8.

```
1 #ifdef CONFIG_USB_HID
2     // some code        (B0)
3 #ifdef CONFIG_INPUT
4     // some more code   (B1)
5 #endif
6 #else
7     // some more code   (B2)
8 #endif
```

**(a)** Contents of `hid-example.c`.

```
1 obj-$(CONFIG_USB_HID)       += hid-example.o
```

**(b)** Makefile with conditional compilation of `hid-example.c`.

**Listing 2.8** – Example for defects caused by including KCONFIG and KBUILD constraints.

When we look at the code alone, it looks fairly simple. `B0` and `B2` depend on `CONFIG_USB_HID` to be enabled or disabled, respectively, while `B1` depends on the condition of `B0` and `CONFIG_INPUT`.

However, looking back at the KCONFIG feature definition in Listing 2.1, we can see that `CONFIG_USB_HID` depends on `CONFIG_INPUT` and `CONFIG_HID`. Thus, it

[2] In fact, inconsistencies like this are not uncommon and keep being introduced. For example, in `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=f6a55884d76c5f493538866793fddd47b4ecf646`, I fixed such a defect in the most recent version of Linux which was introduced in late November 2014.

can only be enabled when its dependencies have already been met. Vice versa, this means that when `CONFIG_USB_HID` is enabled, the KCONFIG definition ensures that `CONFIG_INPUT` and `CONFIG_HID` will definitely be set.

In our example code at Listing 2.8a, block `B1` can thus be marked as **undead**; we can only reach block `B1` if we compile its surrounding (*parent*) block `B0`. Hence, the presence condition of block `B0` (`CONFIG_USB_HID`) must always be *true* for block `B1`, and KCONFIG will have ensured – already during the configuration step – that `CONFIG_INPUT` is also enabled.

As we need to incorporate information from the KCONFIG model to find this class of errors, they are aptly labelled **kconfig defects**.

Lastly, the information from KBUILD can be used to check for so-called **kbuild defects**. The Makefile in Listing 2.8b states that `hid-example.c` will only be compiled if `CONFIG_USB_HID` is enabled. The conditions introduced from the build system are modelled by creating a "virtual" block which surrounds the whole file and depends on the conditions formulated in the KBUILD file.

For the code in Listing 2.8a, this means that everything inside the file depends on `CONFIG_USB_HID` to be set; consequently, code inside block `B2` is **dead**, as the combined preconditions for the block and the file form a contradiction.

Note that the current implementation of the UNDERTAKER tool does not distinguish between the latter two and labels them both as **kconfig** defects. For this thesis, I have modified the code to represent both classes independently – further details on this modification are shown in Section 4.2.

As symbolic violations are relatively easy to detect (by textually searching for the KCONFIG definition of a variable using `grep` or similar tools), there already is a tool called `checkkconfigsymbols.py` inside the Linux source tree. As this tool presently can not process individual patches but only check the whole Linux tree[3], it seems that developers are currently reluctant to integrate it into their routine before submitting patches. As a result, every week several patches are submitted which leave dangling references in the tree, and developers have to be manually notified of their oversights.

Logic defects, on the contrary, are even harder to understand and detect, and thus, often go into the kernel unnoticed. The lack of tool support for automated checks inspired Valentin Rothberg to develop UNDERTAKER-CHECKPATCH [17] as part of his Master's thesis.

[3] Valentin Rothberg sent a patch to allow this checking (see `https://lkml.org/lkml/2015/3/16/190`), but this change has not yet been merged into the Linux mainline tree.

## 2.4 UNDERTAKER-CHECKPATCH

As the development of Linux takes place by submitting patches to subsystem maintainers in order to get code into the kernel, UNDERTAKER-CHECKPATCH analyzes PATCH files in order to allow a developer to quickly check her work before sending it out to the persons maintaining the affected part of Linux.

The tool tries to find changes in terms of defects, both symbolic and logic, by essentially comparing the state of the Linux tree **before** and the state **after** a patch is applied. After finding out which files are changed by the given PATCH file, and generating variability models for the **before** state, UNDERTAKER-CHECKPATCH runs the defect analysis of UNDERTAKER on those files. Next, it parses the PATCH file and updates the range information, i.e., which line numbers the `#ifdef` block covers, for all blocks in the affected files. After applying the patch – and possibly generating new variability models, if a KCONFIG file was changed by the PATCH file – a second defect analysis with UNDERTAKER is run, now on the **after** state.

By comparing the results of the two dead analyses, UNDERTAKER-CHECKPATCH can then determine if new defects were introduced or if the patch fixed or removed them. To make a developer's life easier, the tool subsequently also tries to further analyze the cause of the defect. For example, if a PATCH removes a configurable option from KCONFIG, UNDERTAKER-CHECKPATCH will search for any remaining references to this option which the developer might have forgotten.

## 2.5 Related Work

The importance of taking build system variability information into account when analyzing a large software system like Linux was first presented in a poster session by Berger et al. [3] and explored in more detail in a technical report [2]. The authors implemented KBUILDMINER [11], a *fuzzy parser* to recognize documented as well as undocumented variability specification patterns in the KBUILD Makefiles. However, their approach needs manual modifications in 28 of the analyzed Makefiles. As these modifications are not publically documented, I assume they are non-trivial and are very likely to vary from one version to another; additionally, the authors only provide their extracted conditions for one architecture in one single Linux version (`x86` in Linux `v2.6.33.3`). Thus, I consider this approach not robust enough for an ongoing, highly automated analysis on the most recent versions of Linux, which is one of the main targets of my research.

A more detailed evaluation of the inconsistencies that might be caused by the KBUILD variability in Linux was presented by Nadi and Holt [15]. They found that in the Linux releases `v2.6.26` through `v2.6.39` around 60–100 anomalies could be

detected when looking at KBUILD files and the associated files alone. Most of them are due to the presence of `.c` files in the file system which are never mentioned in (and thus not known to) KBUILD. Based on these results, they implemented a new, regular expression-based parser for the KBUILD Makefiles [16]. In this work, they also extended the UNDERTAKER tool to evaluate the impact of the additional constraints in the formulas. Their findings clearly indicate the importance of taking KBUILD into account: Between `v2.6.30` and `v3.0`, they found 400 to 1900 **more** defects per version than without using the build system data. For an extended version of the paper, they improved their parser, called it MAKEX [14] and published the source code online [13]. Extending their analysis up to Linux version `v3.6`, they were again able to find around 20 percent more defects than without the KBUILD data.

Tackling the robustness issues that arise with textual processing of Makefiles, Dietrich et al. [9] presented a different, probing based approach. Their GOLEM tool first collects information about the KCONFIG features which are mentioned in (and thus influence) KBUILD files. Starting from an empty set of enabled configuration options, they add single features one by one and test which files would be additionally compiled – or which subdirectories will be visited under which conditions – compared to the initial state. For every file found in the probing step, the corresponding KCONFIG feature selection is saved. As this approach treats the build system as a black box and uses it to generate the relevant information, it is more robust to changes or complex semantics in MAKE. However, due to the increasing number of probing steps required, the runtime is very high: Extracting the information for the `x86` architecture alone takes more than three hours on the most recent Linux release `v3.19`.[4] While this might still be acceptable for an off-line study of variability related defects, it renders it practically unusable for a developer who might want to check if their patch introduces any defect.

As the results of two theses in the VAMOS project, GOLEM as well as the UNDERTAKER toolchain have also been ported to support the analysis of BUSYBOX [24] and COREBOOT [10].

[4] In Chapter 5, I present a detailed analysis of the runtime over the last 5 years.

# 3 Design

In this chapter, I first describe which challenges arise for a thorough approach to extract variability information from the build system. After this, I present the conceptual design of my extractor, and how its design is able to cope with the identified challenges.

## 3.1 Challenges

As the goal for this thesis is the development of a text-based variability extractor which should be (a lot) faster than the currently used GOLEM tool while generating as similar data as possible, I first identify the four main requirements which need to be met by my extractor in order to make its practical employment valuable.

**(1) Robustness** First of all, the extraction process must be robust. One of the major reasons for the development of a probing-based approach by Dietrich et al. [9] was the complexity of the (turing-complete) MAKE language. The Linux coding guidelines do not specify any restrictions on the expression of variability, so the kernel developers naturally come up with "non-standard" ways of composing their portion of a KBUILD file. As an example, variables can be modified through string operations like `$(addprefix prefix, string,...)` (which adds the prefix `'prefix'` to all following arguments), they can be evaluated dynamically using `$(eval)` or even result from the execution of shell commands with the `$(shell)` function.

A proper solution must face this complexity in a structured manner: If the expression used in KBUILD can not be evaluated by the approach, this should not break the extraction - instead, we must be able to gracefully ignore the *incomplete* information depending on the expression.

Furthermore, MAKE also allows the use of `if{n}def` or `if{n}eq` statements, creating (possibly variability-related) blocks inside KBUILD.[5] If the conditions for these blocks can not be properly determined, we need to ignore everything inside the blocks as well, as we can not have enough knowledge about the conditions implied by the missing information.

Even more importantly, a good solution must be able to continue working even when the build system is changing. For Linux, this is particularly true: In the development cycle between releases `v3.18` and `v3.19` alone, about 630 commits (out of 13,652 in total) modified KBUILD files. This is not surprising, as newly written files need to be introduced into the build system to integrate them into Linux. Our approach should hence be able to handle these modifications easily and preferably without the requirement for manual adjustments in either KBUILD (c.f. KBUILDMINER [11]) or our tool.

If, however, we detect a major change in the build system that requires an adaption of the extraction process, the approach should make it easy to plug in possible extensions to the current state.

**(2) Accuracy** Equally important and going hand-in-hand with robustness is the requirement for the highest possible accuracy. For the later analysis steps, it is extremely important to find as many files as possible, as well as to properly describe the variability constraints for the individual files - wrong information might lead to false positives, while missing information will not improve the analysis.

In addition to the fairly simple `obj-{y,m}` lists described in Section 2.1.2, KBUILD allows the more complex specification of *composite objects*. These are used when the code is split up into different source files which should then be linked together as a LKM or into the kernel. An example for the use of this technique is shown in Listing 3.1.

```
123 wacom-objs                          := wacom_wac.o wacom_sys.o
124 obj-$(CONFIG_HID_WACOM)             += wacom.o
```

**Listing 3.1** – An example for a composite object in KBUILD, taken from `drivers/hid/Makefile` (Linux v3.19).

[5] For example, to check if a configuration option `CONFIG_INPUT` has been set, a developer might use
```
ifdef CONFIG_INPUT
# Do something depending on CONFIG_INPUT
endif
```
or alternatively:
```
ifeq ($(CONFIG_INPUT),y)
# Do something depending on CONFIG_INPUT
endif.
```

Here, no file called `wacom.c` exists (as line 124 would suggest). Instead, the KBUILD makefile contains a specification for `wacom-objs`, which forms the list of object files that will be linked together to form `wacom.o`. In addition to the format `<driver>-objs`, the lists `<driver>-{y,m}` (with the "usual" `<driver>-$(CONFIG_XY)` idiom) can be used if a developer needs to specify additional constraints for parts of the final `<driver>.o` file. Similarly, a KBUILD file can contain developer-defined variables which can make up parts of the build information.

As already mentioned above, MAKE furthermore contains `if{n}eq` or `if{n}def` statements. These can also use variability information which then influences building of the files in their corresponding block (for example, everything from the statement `ifeq ($(CONFIG_XY),y)` to the corresponding `endif` will only be evaluated if `$(CONFIG_XY)` was set to `y`).

Our extractor hence needs to be able to correctly understand the structure of such `if{n}{def,eq}` statements if any variability information is handled by them. Furthermore, we need to be able to (recursively) process information about variable and macro definitions and properly evaluate the structure of composite object files from their corresponding source files.

**(3) Speed** With the development of UNDERTAKER-CHECKPATCH, Rothberg [17] enhanced the UNDERTAKER [22] tool suite with a tool that can check PATCH files and GIT commits for newly introduced or repaired variability related defects.

However, up to now the variability information from KBUILD could not be taken into account even when we only want to analyze one single commit. This is due to the very high runtime of the current extractor, GOLEM. When executed on a single machine with 8 CPU cores and 16 GB of RAM, GOLEM takes more than three hours per architecture - with a total of 30 architectures in `v3.19`, this would mean extraction times of nearly **four days** to generate the variability information for just one single state of the tree. Even when the extraction is split across machines, as can be done in our lab, a minimum waiting time of six hours (for commits which change KCONFIG files, models have to be generated before and after applying the patch) renders GOLEM entirely unusable for automated analysis as well as for a developer who might want to check her 10-line commit for variability-related errors.

On the other hand, the information from KBUILD is very important for the analysis: The total number of defects found in Linux increases by more than 40 percent when the conditions which GOLEM generates for the files are employed, compared to an analysis without this data.

In order to integrate the build system variability into the analysis (and possibly into the workflow of a kernel developer), our approach therefore needs to be fast – preferably, the extraction should not take more than a few seconds per architecture.

**(4) Adaptability** While Linux might be the biggest openly available configurable system-software project – and thus, the most interesting target for the research of variability-related bugs – it certainly is not the only one. As I have already explained in Section 2.2, the KBUILD system is also used by the infrastructure of the COREBOOT and BUSYBOX projects, and the tools developed in the VAMOS [23]/CADOS [5] research projects have been ported to support their analysis.

As my approach should provide a "drop-in" alternative to the already-present GOLEM extractor, it should hence support not only Linux, but (at least) the three named projects. Optimally, it should provide an easy interface for an extension beyond these to any other text-processable build system.

## 3.2 The Approach

To conquer the challenges above, I developed a line-based parser for the build system files. Even though text-based evaluation of a turing-complete language like MAKE can not achieve a 100 percent accuracy by design, I accept the risk of potentially generating imprecise data for a small subset of the evaluated files in favor of a much lower processing time compared to the probing approach taken by GOLEM.

The parser itself is split up into two parts, separating the generic tasks common to every project from the actual information extraction logic which will be different for every project.

The core part of the tool implements – besides the evaluation of command line parameters – the generic processing of files: It opens a given file and reads it line by line, but does not try to interpret the contents in any way.

Instead, it hands the data over to project-specific "plug-in" modules at predefined points during the processing, which will then extract the required information from the current data.

I identified a total of five points in the parsing process where project-specific actions might be necessary. These points are also shown schematically in Figure 3.1:

❶ **Initialize – before any files have been processed**: At this point, *global variables* that are needed across the evaluation of different files can be created and initialized. Additionally, we need to find the "entry points" for the top-down processing of the build files at this stage.

❷ **Preprocess – after a file has been read into memory, but before any lines from this file have been evaluated**: We might also need some *file-local variables* to store information while we are processing individual files – these can be created here.

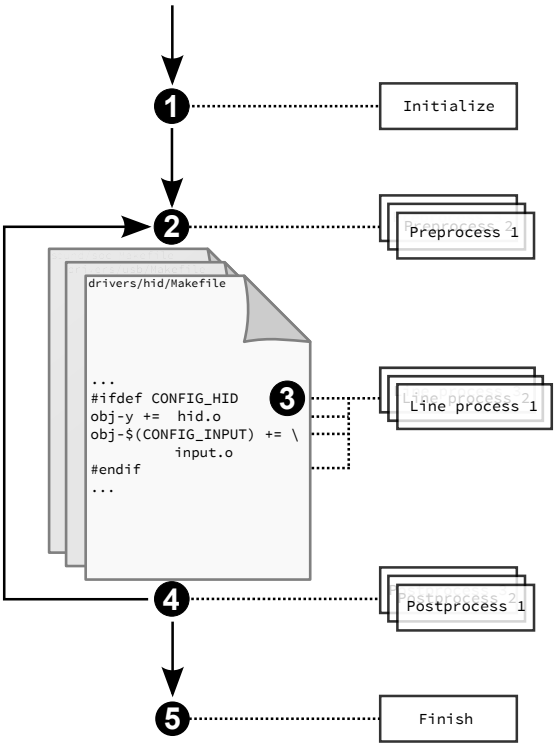



**Figure 3.1** – Schematic depiction of the five points where project-specific "plug-in" modules can be attached into the parsing process.

❸ **Line process – after a line has been read from the file**: This is where most of the actual data collection happens. We can look at the current line, try to extract any variability-related information from it, and store the gathered information.

❹ **Postprocess – after a file has been processed**: During the evaluation of the file, we might have encountered some data which we can only process after the whole file has been read. For example, a subdirectory might be visited depending on different conditions at different locations in the current file. Descending into this subdirectory for further extraction can hence only be done after we have collected every possible condition.

❺ **Finish – after all files have been processed**: In some cases, information might also be spread across different files, thus, we need an opportunity to do some work just before the program finishes.

This design greatly boosts both portability and the robustness of my approach: In order to support a new project, only the plug-in modules have to be designed or adapted – it is not necessary to have duplicates of the core part of the parser. This

makes it easy to port the infrastructure. Furthermore, if new features emerge in KBUILD that need special attention, all we need is another small module which can handle this particular case.

In the following, I will elaborate on the plug-in modules which I developed for Linux in greater detail, and subsequently describe the differences to the modules for BUSYBOX and COREBOOT.

### 3.2.1 The Plug-In Modules for Linux

Having identified the five intervention points during the parsing process, I am now able to assign specific tasks to the modules which are executed at the respective places.

❶ **Initialize** In Linux, KBUILD enters a total of 14 directories[6] from the top-level Makefile without any further configurational constraints. These form the starting point for our recursive top-down evaluation of the source tree. In addition, we need to evaluate the architecture-specific Makefile, which resides in `arch/$(ARCH)/` for the chosen target architecture. As some architectures like ARM or MIPS use special logic to allow the selection of sub-architectures or boards, we need to take special care for them in order to correctly identify the directories that would be visited by the build system.

❷ **Preprocess** As Linux Makefiles specify the conditional constraints only for files inside the current directory, we can collect all the dependencies on a per-directory/per-Makefile basis. Hence, we only need *file-local variables* to store the information about variability. In detail, we need to create data structures to store (a) the values of any variable definitions in the current Makefile, (b) the structure of possibly variability-dependent `if{n}{def,eq}` blocks, (c) the conditions for subdirectories encountered during the pass, (d) the conditions and names of identified *composite objects* and – most importantly – (e) the conditions for files in the current directory.

❸ **Line process** Here, two cases are possible, where each one is handled by a separate module.

Firstly, the line could describe the structure of additional constraints through `if{n}{def,eq}`, `else` or `endif` statements. The individual cases are covered by trying to match a regular expression on the line and extracting any

[6] Namely: `init/`, `drivers/`, `sound/`, `firmware/`, `net/`, `lib/`, `usr/`, `kernel/`, `mm/`, `fs/`, `ipc/`, `security/`, `crypto/` and `block/`.

configurable elements from a match.[7] If we encounter an `if{n}{def,eq}` but can not evaluate its condition, a global nesting level counter is incremented. If this counter is bigger than 0, we can not reliably determine the configurational preconditions and skip every line up to the corresponding `endif`.

The second case is the core part of the extraction: Here, we try to find any additions to the `obj-{y,m}` lists. After textually replacing the values for any variables defined in the Makefile, a complex regular expression is used to check for any conditional constraints and to extract the right-hand side of the assignment.[8] For every item on the right hand side, we check for a corresponding file or directory in the source tree. If there is a file, we store the current configurational constraints as one possible selection required to compile the file. If it is a directory, we also store its conditions in a separate data structure. If neither exists, we assume we are dealing with a *composite object*, again storing the current conditions for later use.

❹ **Postprocess** After collecting all the required data from the lines in the previous step, we can now process them, according to their type.

- For all composite objects, we scan the contents of the file again for the corresponding definitions (see Section 3.1, Robustness). This is done recursively to support nested composite object and macro definitions. Whenever we find source files, their full condition is stored.
- For all directories, we build the logical disjunction over the configurational constraints we discovered, and recursively parse the Makefile in the directory using the disjunction as a precondition which is passed to the parsing step.
- For all files we found, we again build the logical disjunction over their respective constraints and join them with the conditions of the current directory passed down. Finally, the (normalized) name of the file and the logical formula of its precondition are written to the output.

❺ **Finish** For Linux, no action is required after the whole parsing process has run – every information has already been processed in step 4.

[7] For example, I use the expression `\s*(ifdef,ifndef)\s+(.*)` to check for `ifdef/ifndef`. If it matches, the actual expression is captured in a group, as well as the rest of the line. I then try to match the latter against `CONFIG_([A-Za-z0-9_-]+)` to extract the relevant configuration option if there is one.

[8] The full regular expression used is `\s*(obj|lib)-(y|m|\$[\(\]CONFIG_([A-Za-z0-9_--]+[\)\])\s*(:=|\+=|=)\s*(([A-Za-z0-9.,_\$\(\)/-]+\s*)+)`. Note the `lib-` prefix, which is mainly used in the architecture-specific Makefiles as well as the `lib/` subdirectory.

### 3.2.2 The Plug-In Modules for BUSYBOX and COREBOOT

As KBUILD also forms the basis for the BUSYBOX and COREBOOT build systems, the processing modules for these projects are quite similar to those for Linux. I will only describe the differences to the modules used for the "Linux version" of KBUILD, thereby again emphasizing the good portability properties of my approach.

#### BUSYBOX

As the directory structure of BUSYBOX differs from the one in Linux, the list of directories visited is initialized with the respective values for BUSYBOX.[9] Additionally, we exploit the functionality of the `gen_build_files.sh` script: We can simply run it to transform the template KBUILD files into their final form (c.f. Section 2.2.1).

The remainder of the logic is almost identical to Linux. We only need to slightly modify the "central" regular expression to check for the `lib-` prefix (as opposed to `obj-` in Linux). As an optimization, checking for and treating `if{n}{def,eq}` expressions can be neglected – BUSYBOX does not use them in any KBUILD file.

#### COREBOOT

For COREBOOT, more modifications are required. Firstly, the base directories are also specified in the main Makefile as part of the `subdirs-y` list. Similarly, the names of the classes which build the internal lists are specified as an assignment to `classes-y`. Both lists are evaluated in the initialization module of the parser.

In COREBOOT, compilation of the files is only done after the whole build system information has been processed (see Section 2.2.2), and some files are included from many different directories throughout the tree. This means that we also need a *global* variable to collect the build system preconditions for individual files.

While parsing of `if{n}{def,eq}` conditions is equivalent to Linux, the extraction of the files again has to be adapted to support the different names of the lists – instead of one regular expression, we build all possible variants defined by the names in `classes-y`. If one of them matches, we continue as described for Linux above, saving the conditions for the found files into the global variable.

As a consequence of collecting the information for the files in a global variable, the conditions can only be written to the output after all files have been processed.

[9] Namely, these are `applets/`, `archival/`, `archival/libarchive`, `console-tools/`, `coreutils/`, `coreutils/libcoreutils/`, `debianutils/`, `e2fsprogs/`, `editors/`, `findutils/`, `init/`, `libbb/`, `libpwdgrp/`, `loginutils/`, `mailutils`, `miscutils/`, `modutils/`, `networking/`, `networking/libiproute/`, `networking/udhcp`, `printutils/`, `procps/`, `runit/`, `selinux/`, `shell/`, `sysklogd/`, `util-linux/` and `util-linux/volume_id/`.

# 4 Implementation

In this chapter, I present how I implemented the parser based on the concept presented in Chapter 3. Furthermore, I explain its integration into the UNDERTAKER toolchain of the VAMOS/CADOS project and the modifications to the current state of the tools.

## 4.1 The Parser: MINIGOLEM

First, I would like to take a closer look at the implementation details of the parser, and how they contribute to meeting the challenges described earlier.

### Choice of Language

As more than 75 percent of the UNDERTAKER toolchain are written in C++, and due to the high speed of programs written in compiled languages in general, I implemented the first prototype in C++. The compilation, however, is a big drawback when we look at the modularity criterion and the hence proposed plug-in–based, modular design: While it is possible to load additional, user-provided code into a C++ program through dynamic library loading, it would still require the user to build shared libraries for all modules they want to use.

Considering the fact that the UNDERTAKER toolchain is not only available as source code but can also be installed from binary packages in a number of Linux distributions (Debian, Ubuntu, ...), we definitely do not want the user to be forced to (re-)build the tools whenever she wants to analyze a different project. Similarly, while it would be possible to pre-build and ship the different variants we have already implemented on our side, we would not be able to let the user easily experiment with modifications to the parsing logic, for example to add support for a completely different build system to the tool.

After some experimentation with the language features, I decided to drop the C++ implementation and ported the then-current state to PYTHON. While the runtime increased by 160 percent (from around 0.5*s* to 1.3*s*), I still deem it fast enough – especially when compared to GOLEM – while offering more possibilities for runtime extensions to the parser.

## Modular Design

As I have described earlier, the parser consists of two parts. The core of the parser only implements a generic abstraction for opening and reading files, but does not extract any variability information. Extraction is implemented in separate, project-specific modules that can be attached to five specific points during the process. In the implementation, the five different types of actions (**Initialize, Preprocess, Line process, Postprocess** and **Finish**, see Section 3.2) are expressed by using five different abstract base classes from which the concrete extraction modules have to be derived. These classes are called `InitClass`, `BeforePass`, `DuringPass`, `AfterPass` and `BeforeExit`, respectively, and define the function interfaces which then need to be implemented.

```
268 for pyfile in os.listdir(additional_path):
269     if not pyfile.endswith(".py") or pyfile.startswith("__"):
270         continue
271 
272     module = importlib.import_module(pyfile[:-3])
273     for (name, cls) in sorted(module.__dict__.items()):
274         if issubclass(cls, BaseClasses.InitClass):
275             parser.init_class = cls(read_model, args.arch)
276         elif issubclass(cls, BaseClasses.BeforePass):
277             parser.before_pass.append(cls(read_model, args.arch))
278         elif issubclass(cls, BaseClasses.DuringPass):
279             parser.during_pass.append(cls(read_model, args.arch))
280         elif issubclass(cls, BaseClasses.AfterPass):
281             parser.after_pass.append(cls(read_model, args.arch))
282         elif issubclass(cls, BaseClasses.BeforeExit):
283             parser.before_exit.append(cls(read_model, args.arch))
```

**Listing 4.1** – Excerpt from the core module of the parser which facilitates the dynamic loading of the project-specific modules. The `additional_path` parameter describes the directory which should be searched for the modules. Every file inside this directory is then imported using `importlib.import_module`, and any classes derived from the predefined `BaseClasses` are loaded and added to their respective lists.

When the parser starts, it first tries to determine the project it is currently running on. If it is one of Linux, BUSYBOX or COREBOOT, it can automatically load the classes which I have implemented for this thesis. To override or extend the set of loaded classes, the user can provide a directory on the command line. Inside this directory, the user should have placed one or more PYTHON files in which she implemented the functionality in classes derived from one of the five base classes described above. These classes will then be instantiated and loaded into respective lists provided by the core module. A user can provide more than one derived class per baseclass: This allows a separation of tasks into specialized modules (for example, processing of `if{n}{def,eq}` expressions and of the `obj-` definitions in Linux are implemented in separate classes, both derived from `DuringPass`). When multiple classes are loaded, their execution order is established by sorting the modules alphabetically by their name. The code for the loading functionality is shown in Listing 4.1.

Once all processing modules have been loaded, the `process` method of the `InitClass` module is called. Here, the list of directories to be visited in the first step has to be initialized. After this information has been established, the core module calls the main processing function, which is listed in Listing 4.2, for every directory.

```
123 def process_kbuild_or_makefile(self, path, conditions):
124     self.enter_new_symbolic_level()
125 
126     for processor in self.before_pass:
127         processor.process(self, os.path.dirname(path))
128 
129     self.read_whole_file(path)
130 
131     for line in self.file_content[path]:
132         for processor in self.during_pass:
133             if processor.process(self, line, ↘
                    os.path.dirname(path)):
134                 break
135 
136     for processor in self.after_pass:
137         processor.process(self, path, conditions)
138 
139     self.leave_symbolic_level()
```

**Listing 4.2** – (Simplified) main processing function of the core module. The parser maintains a file-local dictionary for variables, which is initialized and reset by the `{enter,leave}_symbolic_level()` methods. Note the call into the `BeforePass`, `DuringPass` and `AfterPass` subclasses at the respective points during the processing of a file.

When this function is called on a directory, it will first create a new dictionary (PYTHON's own key-value store data type) which will later contain file-local variables, and stash the current dictionary (`enter_new_symbolic_level`, Line 124). Then, all modules derived from `BeforePass` will be called. After reading in the whole file (and resolving any `include` statements on the way), the parser iterates over the lines and hands them to the `DuringPass` modules. Once the whole file has been processed, the code in the `AfterPass` modules is run, before the file-local variables are reset to their old state.

The individual modules then implement the functionality which I already described in Section 3.2.1 and Section 3.2.2 – for example, the Linux part contains two subclasses of `DuringPass` which implement (a) the detection of `if{n}{def,eq}` statements and (b) the detection of additions to the `obj-{y,m}` lists.

For Linux, the specific modules are implemented in a total of 508 lines of code (see also Table 4.1). To avoid unnecessary code duplication, I further exploit the concept of class inheritance: When identical functionality is required in another project – for example, treating `if{n}{def,eq}` is the same in Linux and in COREBOOT – the corresponding modules can inherit from one another; speaking in code, the `Coreboot.CorebootIf` module does not inherit from `BaseClasses.DuringPass`, but rather from `Linux.LinuxIf`, thereby fully acquiring its functionality without having to write more code. Thus, only differences to the process for Linux have to be modelled. This also reduces the lines of code required for other, similar projects: The logic for COREBOOT is handled in 297, the logic for BUSYBOX in only 187 additional lines of code.

| Part | # LoC |
|---:|---:|
| Core parser | 192 |
| Linux modules | 508 |
| COREBOOT modules | 297 |
| BUSYBOX modules | 187 |
| Σ | 1,184 |

**Table 4.1** – Lines of code for the core parser and the respective modules for the individual projects, not counting comments or blank lines.

## 4.2 Modifications to the UNDERTAKER Toolchain

In order to seamlessly integrate the parser into the current UNDERTAKER toolchain, I modified some of its components.

Currently, the models describing the features and their dependencies in KCONFIG are generated with `undertaker-kconfigdump` which wraps around the extraction

and the transformation into boolean formulas. To extract the KBUILD data (called *inferences*) using GOLEM, `undertaker-kconfigdump` can be called with the `-i` flag which then sets up the call to GOLEM and the integration of the extracted data into the generated model.

Similar to the GOLEM case, I added a command line flag `-p` to `undertaker-kconfigdump` which will call the parser with the required parameters and integrate the preconditions into the generated models.

In UNDERTAKER itself, I modified the classification of defects. So far, the analysis first checked if any code defects could be found without taking the KCONFIG model into account (see Section 2.3.1). If this was not the case, the whole model (including KBUILD data) was loaded and the required data was added into the formula. A defect that stems from the KBUILD preconditions thus would result in a report of a **kconfig** `{un}dead` block due to the lack of a designated **kbuild** defect class. While this is sufficient to detect if there was a problem, it does not give a very clear indication to the user where she might want to investigate to find the cause for the defect.

To improve the distinction between **kconfig** and **kbuild defects**, I changed the analysis step to work as follows. Let $\mathscr{C}_{CPP}$ be the conditions for the block which are expressed by the C preprocessor inside the file, and $\mathscr{C}_{\text{KBUILD}}$ the extracted KBUILD condition for the source file.

- First, we analyze if $\mathscr{C}_{CPP}$ is solvable. If it is not – or if it is a tautology – then the block is classified as a **code defect**. No changes are required for this step.

- Next, $\mathscr{C}_{CPP}$ is extended with information from the transitive KCONFIG definitions of all its components by using the slicing algorithm presented by Sincero [18]. Thus, we get $\mathscr{C}_{\text{KCONFIG}} = slice(\mathscr{C}_{CPP}, model_{\text{KCONFIG}})$.

  So far, the (transitive) KBUILD conditions for the file were also added in this step, resulting in $\mathscr{C}_{\text{KCONFIG}-Defect} = \mathscr{C}_{CPP} \wedge \mathscr{C}_{\text{KCONFIG}} \wedge slice(\mathscr{C}_{\text{KBUILD}}, model_{\text{KCONFIG}})$. In the implementation, this was facilitated by introducing a bi-implication with the top-level block `B00` and the variable which contains the additional constraints from KBUILD.[10]

  With my modification, we do not add the KBUILD information yet, and only feed $\mathscr{C}_{\text{KCONFIG}-Defect} = \mathscr{C}_{CPP} \wedge \mathscr{C}_{\text{KCONFIG}}$ to the SAT solver. If the solver finds contradictions or a tautology, respectively, we classify the block as a **kconfig defect**.

- Instead, to allow the distinction of defects resulting from variability in the build system, the constraints from KBUILD are added in the next, separate step. As before, we need to apply the slicing algorithm to $\mathscr{C}_{\text{KBUILD}}$ to resolve

[10] For example, for the source file `kernel/smp.c` the additions to the formula would be `(B00 <-> FILE_kernel_smp.c) && (FILE_kernel_smp.c -> CONFIG_SMP)`.

any additional transitive dependencies. Then, we extend the formula from above and get $\mathscr{C}_{\text{KBUILD}-Defect} = \mathscr{C}_{\text{KCONFIG}-Defect} \wedge slice(\mathscr{C}_{\text{KBUILD}}, model_{\text{KCONFIG}})$. This formula is retested in an independent pass of the SAT solver. If the formula for the block can not be solved (or is a tautology), we classify it as a **kbuild defect**.

- Lastly, the formula is extended to detect if any symbols which are not defined by KCONFIG cause the problem. Therefore, all undefined symbols from $\mathscr{C}_{\text{KBUILD}-Defect}$ are explicitely forced to evaluate to `False`.[11] Let $\mathscr{M}$ be the set of missing KCONFIG symbols which are not defined in the model. We then build $\mathscr{C}_{Missing-Defect} = \mathscr{C}_{\text{KBUILD}-Defect} \wedge \left(\bigwedge_{m \in \mathscr{M}} \neg m\right)$ and test the resulting formula with the SAT solver. If this is a contradiction or a tautology, the defect is classified as a **missing defect**. This step also did not need any modifications.

## 4.3 Using the Data in UNDERTAKER-CHECKPATCH

One motivation for the development of the parser was the possibility to include the variability information from KBUILD into the automated defect analysis of PATCH files submitted into the Linux kernel which was not feasible with GOLEM due to its high runtime.

To facilitate this analysis, I extended UNDERTAKER-CHECKPATCH to make it recognize the newly introduced **kbuild** defect class described above. In addition to the block's precondition, UNDERTAKER-CHECKPATCH also prints the file preconditions from KBUILD for the analyzed architectures, guiding the developer towards the conditions from the build system for further, manual analysis.

Furthermore, together with Valentin Rothberg I have developed an experiment in the `versuchung` framework [8] which analyzes `linux-next` (the current development tree of Linux [12]) for variability-related errors.

In this experiment, we get the most current version of `linux-next` and extract all commits newly added since the last update the day before. We then walk through the GIT commits, calling UNDERTAKER-CHECKPATCH on every commit.

Note that walking through the GIT history requires some special precautions to properly analyze the individual commits. The current `linux-next` tree is built by merging 215 different trees on top of the most recent mainline source tree[13], and

[11] Without explicitly setting them to `False`, a SAT solver considers them as *free* variables and is allowed to set them to `True` when generating a valid assignment.

[12] Most patches for Linux are written for this tree and are first merged there before they make it into Linus' mainline tree. The tree is usually updated Mondays to Fridays, and the updated tree is tagged with its date. The web interface for `linux-next` can be found at `http://git.kernel.org/cgit/linux/kernel/git/next/linux-next.git`.

[13] See `https://lkml.org/lkml/2015/4/13/287`.

all these trees might be based on different prior version of the kernel. By using the output of `git log` to extract the commits, we only get a "flattened" view of all these trees; through the `-topo-order` parameter, we force `git log` to first show all commits coming from one individual development tree before it proceeds with another tree.

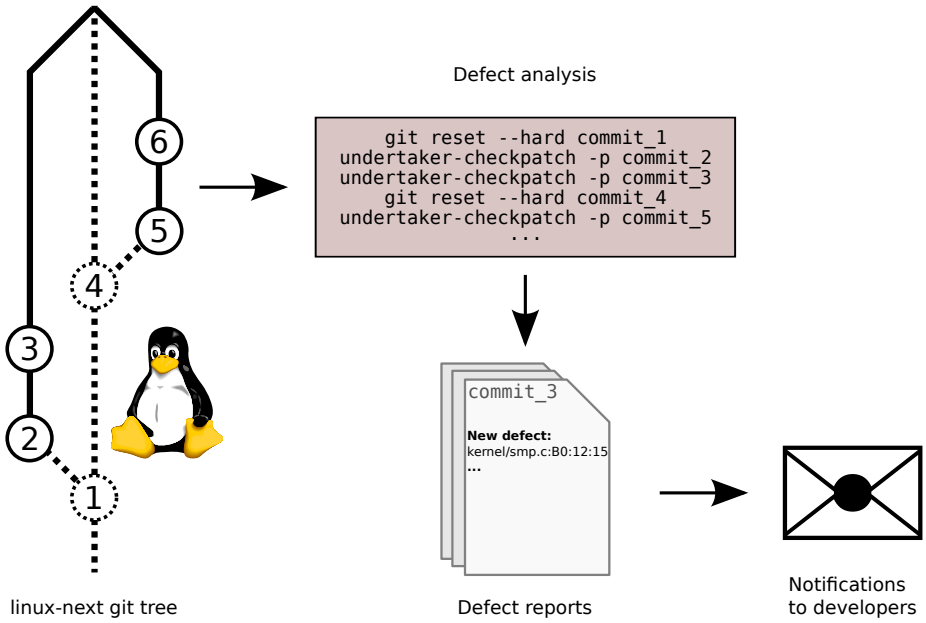



**Figure 4.1** – Overview of the `linux-next` experiment. In the Linux tree, commits are numbered in the order in which they will be processed. The actual commits we want to analyze are 2, 3, 5 and 6; to have the correct basis for commit 2, however, we have to reset the GIT repository to its parent, commit 1. After we have checked commits 2 and 3 on this tree, we then switch to a different merged tree which originated from a different point in the mainline GIT history. Thus, it is necessary to first reset to commit 4 before analyzing its descendants, commits 5 and 6.

Analyzing the commits in this particular order (processing the merged trees en bloc, and processing the individual commits from one tree in the original order they also have in that tree, from oldest to newest) has the advantage that we can greatly reduce the overhead of model generation: After generating the KCONFIG/KBUILD models for the earliest commit in the branch currently being worked on, we can pass them to UNDERTAKER-CHECKPATCH which then only needs to generate new models for the state **after** the patch from the first commit has been applied; for every commit from the same tree, these will then be used as the *input* models for the second patch and so on.

Special care only needs to be taken when we have finished processing all commits from one tree with UNDERTAKER-CHECKPATCH: When the next commit $C_{(n+1)}$ is from

a different tree, we can not simply use the models generated for the analysis of the last commit $C_{(n)}$ we processed, as these two commits are most likely not related (i.e., $C_{(n)}$ is not the parent commit of $C_{(n+1)}$). Instead, we need to find the *real parent* of $C_{(n+1)}$, reset the analysis Linux tree to this parent, and generate new KCONFIG/KBUILD models. After doing this, we can process $C_{(n+1)}$ and all following commits from the same tree as described above (see Figure 4.1 for a graphic example).

The experiment is run daily on our build servers and reports all newly introduced, repaired and unchanged defects in an automatically generated email. Currently, Valentin Rothberg and I manually check all defect reports for false positives. This is necessary, because changes to Linux are often split up into a range of individual patches which are submitted as one *patch series* – our tool, however, can not see the connection between the corresponding commits in GIT. Furthermore, some defects might be intentionally introduced and have an explanatory comment next to them; again, these are situations our approach can not automatically detect. If we have established the validity of an introduced defect, we send out notifications to the respective authors, providing them with an explanation of the defect present in their patch.

# 5 Evaluation

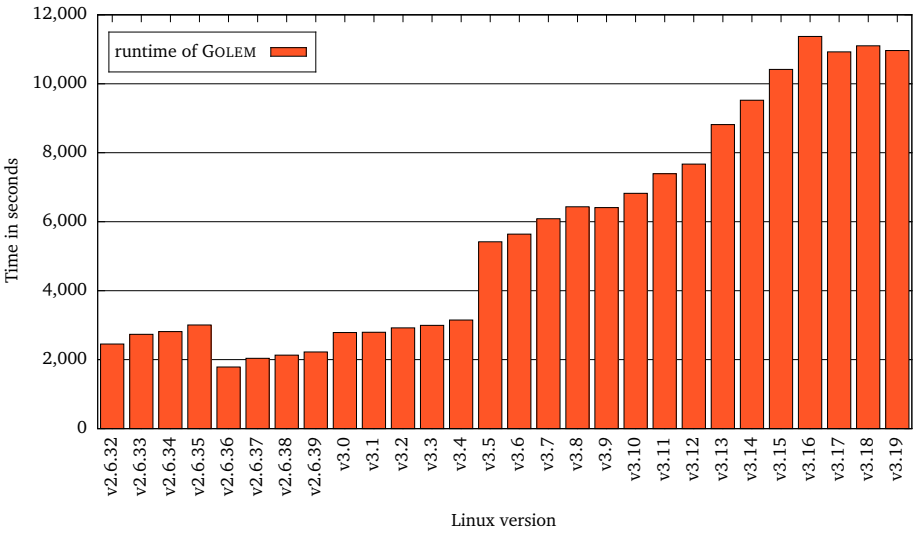
In this chapter, I present a detailed evaluation of the approach. Therefore, I assess how the parser tackles the challenges which I formulated in Section 3.1; after showing that the approach is fast and able to robustly extract data for all Linux versions released in the past five years, I will demonstrate its accuracy by comparing the extracted formulas for the most recent release, `v3.19`, and demonstrate the improvements in the `dead/undead` analysis of the UNDERTAKER tool when the parser data is used. Furthermore, I present concrete examples of additional variability inconsistencies which I found – and fixed – using the automated checking experiment for `linux-next`. Lastly, I show the results for the parser on COREBOOT and BUSYBOX.

## 5.1 Speed

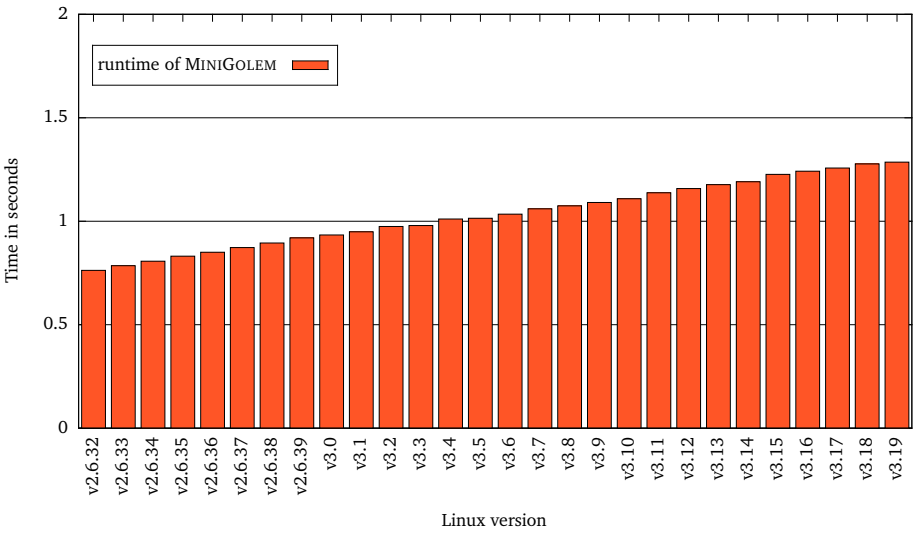
The main reason prohibiting the employment of GOLEM in UNDERTAKER-CHECKPATCH – and thus, the motivation for the work presented in this thesis – is its high runtime. On the most recent release of Linux, `v3.19`, GOLEM takes more than three hours to extract the variability data from KBUILD on a machine equipped with a quad-core Core i5-4590 CPU with 3.3 GHz and 16 GiB RAM – and this is only for one architecture, `x86`[14]!

However, the runtime was not always this high: GOLEM was developed around the time when Linux `v3.2` was current. On this version, the extraction process for `x86` only takes less than 50 minutes on an identical machine. As we can see from Figure 5.1a, the runtime has since increased massively with almost every release, sometimes by over 1,000 seconds from one release to the next.

[14] I use `x86` as the reference architecture in this section because it is the architecture which has received most attention from researchers (including ourselves), and because it is still considered the "core" architecture of the Linux kernel.




**(a)** Runtime of GOLEM on the `x86` architecture in seconds for all Linux versions since v2.6.32. Values displayed are averaged from three independent runs on identical machines (Core i5-4590 processor with 3.3 GHz, 16 GiB RAM).




**(b)** Runtime of MINIGOLEM on the `x86` architecture in seconds for all Linux versions since v2.6.32. Values displayed are averaged from five independent runs on the same machines as for the GOLEM runs.

**Figure 5.1** – Comparison of runtimes for GOLEM and the new, parsing-based approach from Linux version `v2.6.32` to the most recent version, `v3.19` on the `x86` architecture. Due to GOLEM's high runtime, I could only execute three full runs across all versions. Note the different scale of the y axis.

There are a few noteworthy peculiarities in the figure: From `v2.6.35` to `v2.6.36`, the runtime abruptly decreases by over 40 percent (from 3,005 to 1,787 seconds). Up to `v3.4`, it continues to increase steadily, only to jump up from 3,149 to 5,417 seconds – over 70 percent! – for `v3.5`. Subsequently, I tracked these big jumps down to individual commits: while they both modify the main Linux Makefile, it is not obvious why the changes have such a high impact on the runtime.[15] I assume the modifications have some influence on a *top-level* point of variability, thereby nearly halving (in the case of the decrease between `v2.6.35` and `v2.6.36`) or doubling (between `v3.4` and `v3.5`) the number of required probing steps.

Additionally, since peaking in version `v3.16` at more than 11,300 seconds, the runtime of GOLEM stagnates around the 11,000 seconds mark, even though the number of KCONFIG features as well as the number of conditions extracted from the build system continue to grow.

The development of the runtime of MINIGOLEM on the other hand looks a lot smoother: As shown in Figure 5.1b, parsing the Makefiles only takes 0.76 seconds on `v2.6.32`. Through the releases, a near-perfect linear increase can be observed by about 0.02 seconds per release, leading to an extraction time of only 1.29 seconds on the most recent version `v3.19` – this is more than **eight thousand** times faster than GOLEM! But being fast is worth nothing if the extracted data is bad, so we need to look at the formulas and measure their quality.

## 5.2 Robustness and Quality

As already mentioned, an important point for the integration of MINIGOLEM into an automated analysis of Linux is the robustness: the process must be able to perform reliably in the presence of changes to the build system throughout the development process. When I started writing MINIGOLEM, the most recently released Linux version was `v3.16`, and I used this version as the testing base throughout the whole implementation process; only after I was satisfied with the extracted data did I run the tool on older Linux releases as well.

To measure the quality of the extracted variability information, I first need to determine all files for which GOLEM and MINIGOLEM have both extracted information from KBUILD (more formally: Let $F_g$ be the set of files for which GOLEM extracted variability information, and $F_m$ be the set of files reached by MINIGOLEM. Then $F_{intersect} = F_g \cap F_m$ contains all files which are covered by both approaches). Next, to determine if the extracted information for a file is logically equivalent, I build the *bi-implication* between the condition from GOLEM and the condition from MINIGOLEM,

[15] Going from `v2.6.35` to `v2.6.36`, the offending commit is `6588169 - "kbuild: allow assignment to {A,C,LD}FLAGS_MODULE on the command line"`. Between `v3.4` to `v3.5`, the increase occurs at commit `1f2bfbd - "kbuild: link of vmlinux moved to a script"`.

and use a checking tool, LIMBOOLE [12], to determine if the resulting formula is always valid (again, more formally: $\forall f \in F_{intersect}$: Let $C_{(f,g)}$ be the condition for file $f$ extracted by GOLEM, and let $C_{(f,m)}$ be the condition for the same file $f$ extracted by MINIGOLEM. If the equation $(C_{(f,g)} \iff C_{(f,m)})$ holds, then both approaches extracted the same conditions for $f$ from KBUILD).

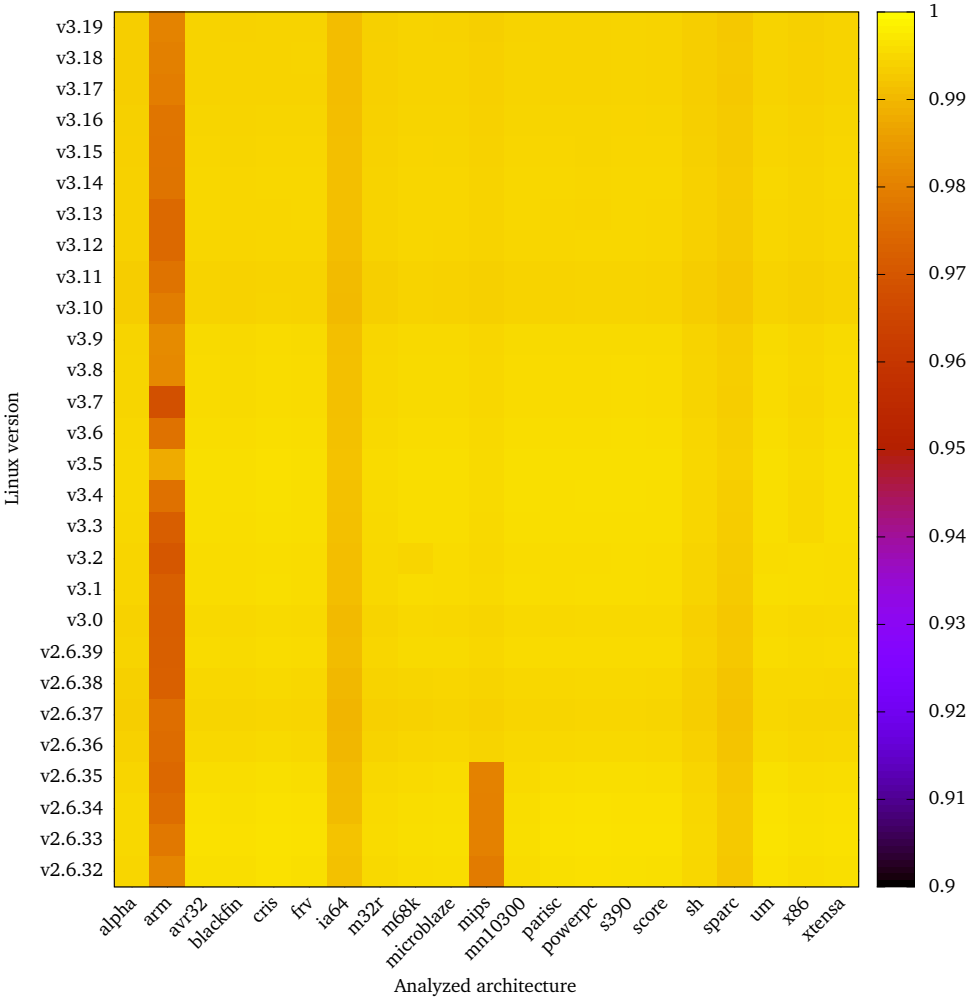



**Figure 5.2** – Percentage of logically equivalent extracted conditions for files which are covered by both approaches. The x axis denotes the 21 architectures available on every measured version, which are shown along the y axis.

As every architecture is handled independently by our toolchain, the individual models are tested separately to additionally determine how well the extraction process works for the respective target architecture. Over the course of five years, some architectures were added to Linux, while others were removed; in order to

show a consistent image, I picked the 21 architectures available on every version from `v2.6.32` to `v3.19` for this analysis.

Figure 5.2 shows the results for all versions and all architectures as a heat map. To my own surprise, the formulas extracted by MINIGOLEM for most version/architecture combinations are almost equivalent to the data from GOLEM; the percentage of logically equal formulas is mostly at around 99.3 percent.

There are, however, some clearly identifiable outliers: `ARM` and `MIPS` (from `v2.6.32` to `v2.6.35`) which reach around 96 to 98 percent accuracy, respectively, as well as `IA64` and `SPARC`, both reaching 99.1 percent accuracy. While this is still quite high, I manually looked into the formulas to determine the cause for this discrepancy.

For `ARM`, I found that GOLEM sometimes adds features to the formula for a file which do not appear anywhere in the build system. This is likely to be an undesired effect of its probing strategy: During the evaluation of a concrete selection of KCONFIG features, KCONFIG might trigger the selection of *other* features through their dependencies. For GOLEM, it then looks like the file can only be built with the selected feature enabled, and it writes this feature into the dependencies for the file. As an example, in version `v3.7`, MINIGOLEM extracts the condition `CONFIG_ARCH_INTEGRATOR` for the file `arch/arm/mach-integrator/leds.c`, while GOLEM generates `CONFIG_ARCH_INTEGRATOR && CONFIG_COMMON_CLK_VERSATILE`. The latter symbol appears nowhere inside the path through KBUILD to build the file, but `CONFIG_ARCH_INTEGRATOR` **selects** it in KCONFIG.

Similarly, in the problematic versions of `MIPS`, GOLEM adds `CONFIG_PCI` to the condition for all files inside the `arch/mips/` directory where the corresponding architecture **select**s `CONFIG_PCI` in its KCONFIG definition.[16]

The explanation for the slightly lower values for `IA64` and `SPARC` is analogous – in `IA64`, around 50 files carry an additional dependency on `CONFIG_SGI_XP` in the conditions extracted by GOLEM; in `SPARC`, around 30 files show a reference to `CONFIG_SPARC32`.

All these additional conditions, however, are information coming from the KCONFIG dependencies rather than the KBUILD files and should hence not appear in the KBUILD data. Consequently, I consider the data from MINIGOLEM to be a better representation for the actual build system conditions and conclude that the "lower" accuracy is in fact not a problem with MINIGOLEM, but rather the implication of an improvement over the current state of GOLEM.

[16] Going from `v2.6.35` to `v2.6.36`, the build process for sub-architectures of `MIPS` was reorganized – this also made the wrong behaviour disappear from GOLEM.

## 5.3 A Closer Look at Accuracy

I now want to dive deeper into the formulas that MINIGOLEM generates, and illustrate in more detail where the differences to the GOLEM data are and where they stem from. To do this, I use the newest version of Linux, `v3.19`, and extract the build system conditions for the `x86` architecture with both GOLEM and MINIGOLEM. A brief statistical overview over the data is given in Table 5.1.

| | | |
|---|---|---|
| Number of files found by GOLEM ($F_g$) | 15,072 | |
| Number of files found by MINIGOLEM ($F_m$) | 15,303 | |
| Files covered by both approaches ($F_g \cap F_m$) | 14,944 | |
| $\Rightarrow$ Files only covered by GOLEM | 128 | |
| $\Rightarrow$ Files only covered by MINIGOLEM | 359 | |
| Number of equivalent formulas for files in $F_g \cap F_m$ | 14,831 | (99.24%) |
| $\Rightarrow$ Number of differing formulas | 113 | |

**Table 5.1** – Statistics for the extracted KBUILD conditions by GOLEM and MINIGOLEM in Linux version `v3.19` on the `x86` architecture.

We can see that GOLEM found build system conditions for 15,072 files, while MINIGOLEM found 15,303 files. This corresponds to 96.1 and 97.6 percent, respectively, of all source files in the Linux tree for the `x86` architecture. The number of files which have conditions in both data sets is 14,944, leading to 128 files only present in the GOLEM data and 359 files only in MINIGOLEM.

For all files covered by both extractors, I again use LIMBOOLE [12] to determine if the generated conditions are logically equivalent (c.f. Section 5.2). Out of the 14,944 conditions in ($F_g \cap F_m$), 14,831 are indeed equivalent, which gives us a 99.24 percent accuracy. Much more interesting than the equivalent formulas, however, are the differing ones and the reasons for the difference, which I will now present in more detail.

First, let us take a look at the 128 files which GOLEM lists, but for which MINIGOLEM could not extract conditions from KBUILD. In the GOLEM data, 37 conditions are for `object` files rather than source files. These stem from developer-defined Makefile targets which do not fit into the `obj-` list pattern used for the regular build process – these are used to specify dependencies or build rules which differ from the regular pattern. MINIGOLEM does not show them because it relies on the information from the `obj-{y,m}` lists, but as the UNDERTAKER only analyzes source files, this data is not needed anyway.

Most of the remaining files have conditions which MINIGOLEM currently can not evaluate sufficiently. In one case, the path to the source files is generated dynamically

inside the Makefile, in another case a dynamic `$filter()` call is used to build the condition for an `ifneq` block inside the Makefile. These are all situations in which a (static) parser has too little insight into the data actually being evaluated by KBUILD; luckily, less than 90 files are treated like this – I consider this a low-enough number to ignore for now in favor of the runtime improvement.

Lastly, GOLEM shows some bogus entries for files like `cat` or `make`; these are probably caused by incorrect parsing of error messages during the probing steps which then end up in the output with some conditions attached.

Next, I examine the 359 files which MINIGOLEM found in addition to the common set. Here, I am able to identify the common cause for their absence in the data extracted by GOLEM.

By design, GOLEM picks up all KCONFIG items mentioned in the Makefiles and systematically probes KBUILD by turning individual KCONFIG features off and on, and looking at which other files are built if one extra feature is enabled additionally. In some Makefiles, however, two (or more) KCONFIG features control the inclusion of a particular file (see Listing 5.1 for an example).

```
17 ifeq ($(CONFIG_MEDIA_CONTROLLER),y)
18   obj-$(CONFIG_MEDIA_SUPPORT) += media.o
19 endif
```

**Listing 5.1** – Example for a Linux Makefile where compilation of media.c depends on more than one KCONFIG feature, taken from `drivers/media/Makefile` (Linux `v3.19`).

In this case, `media.o` will only be built if both `CONFIG_MEDIA_CONTROLLER` and `CONFIG_MEDIA_SUPPORT` are enabled. With a probing approach agnostic of this dependency, the two KCONFIG items will be tested separately; due to the fact that GOLEM always starts with the empty selection of options (i.e., all configuration options are disabled), enabling only one of the respective KCONFIG items will never trigger compilation of `media.o`, and golem will not report the configurational constraints for the underlying source files (see Dietrich et al. [9], Section 6.1 for a further discussion of these cases).

Lastly, we need to understand the origins of the remaining 113 files for which both MINIGOLEM and GOLEM have extracted conditions ($C_{(f,m)}$ and $C_{(f,g)}$, respectively), but for which the conditions are not equivalent, i.e. do not represent a valid *bi-implication* ($C_{(f,g)} \Leftrightarrow C_{(f,m)}$) when tested with LIMBOOLE.

When we have established that $(C_{(f,g)} \Leftrightarrow C_{(f,m)})$ is not valid, we can further distinguish three different sub-cases:

1. $C_{(f,g)} \Rightarrow C_{(f,m)}$, i.e., the condition from GOLEM implies the condition from MINIGOLEM: If this formula is valid (again, this can be tested using LIMBOOLE), it means that the formula produced by GOLEM describes stronger constraints than the formula from MINIGOLEM.

   A total of 22 files correspond to this case; the stronger constraints are caused by either an architecture variable like `CONFIG_X86_32` which is not explicitly used in the build system but has an influence on dependencies or by missing `_MODULE` counterparts for `tristate` KCONFIG features in the GOLEM formula.

2. $C_{(f,m)} \Rightarrow C_{(f,g)}$, i.e., the condition from MINIGOLEM implies the condition from GOLEM: This case is the opposite of the first, meaning that the formula from MINIGOLEM is more restrictive than the formula from GOLEM.

   With 91 conditions fulfilling the implication, this covers all remaining files. The overwhelming majority of MINIGOLEM conditions in this group contains a negated KCONFIG item (i.e., for `drivers/char/nvram.c`, the condition is `!CONFIG_GENERIC_NVRAM && CONFIG_NVRAM`) while the GOLEM formula does not contain the negated item (in our example, GOLEM only produces the conditions `CONFIG_NVRAM`). This is again based on an assumption of Dietrich et al. [9] that enabling a KCONFIG item will always select **additional** source files for compilation but never **remove** any files. If the assumption is violated (e.g., files are added to `obj-y` inside an `ifndef CONFIG_GENERIC_NVRAM` block), GOLEM fails to detect the correct constraints. MINIGOLEM, however, will detect the negation and pick up the corresponding condition for the source file.

3. $C_{(f,g)} \neq C_{(f,m)}$, i.e., the conditions are entirely different. In the analyzed data, this case did never appear, further assuring me that the quality of the data generated by parsing is sufficiently high for practical use.

## Comparison to MAKEX [13]

In addition to GOLEM, I also tested the extracted conditions for `x86` on Linux version `v3.19` against the data produced by the MAKEX parser. The extraction process takes around two seconds per architecture – about 50 percent slower than MINIGOLEM, but still orders of magnitude faster than GOLEM.

Table 5.2 shows a summary of the data. We can see that MAKEX only finds a total of 13,296 conditions for source files, and that the number of files covered by both

| | | |
|---|---|---|
| Number of files found by MAKEX ($F_x$) | 13,296 | |
| Number of files found by MINIGOLEM ($F_m$) | 15,303 | |
| Files covered by both approaches ($F_x \cap F_m$) | 12,036 | |
| ⇒ Files only covered by MAKEX | 1,260 | |
| ⇒ Files only covered by MINIGOLEM | 3,267 | |
| Number of equivalent formulas for files in $F_x \cap F_m$ | 11,885 | (98.75 %) |
| ⇒ Number of differing formulas | 151 | |

**Table 5.2** – Statistics for the extracted KBUILD conditions by MAKEX and MINIGOLEM in Linux version `v3.19` on the `x86` architecture.

approaches is at only 12,036, leaving more than 3,000 files which appear only in MINIGOLEM, while 1,260 conditions are present only in the MAKEX data.

A manual inspection of the generated conditions revealed that the files missing from MAKEX are almost entirely parts of composite objects, while the surplus files in MAKEX are not present in the Linux source tree but have names closely related to the composite objects in question.

When looking at the source code of MAKEX, I found that it does not check with the underlying file system if a file really exists before printing its condition to the output. In combination with parsing errors for some types of composite objects, this leads to (a) the output of the name of the *composite* object (which has no corresponding source file!) and (b) missing conditions for the individual parts which constitute the composite object.

While the accuracy for the files covered by both approaches is at a very good 98.75 percent, missing **over** 20 **percent** of source files handling composite objects is not acceptable for a thorough defect analysis of Linux.

Overall, MINIGOLEM extracts the configurational constraints from KBUILD faster than MAKEX and achieves a much higher coverage of the source files. The high accuracy in files described by both extractors once again underlines the quality of the constraints extracted by MINIGOLEM.

## Summary

The extraction of constraints from KBUILD through a parsing-based approach works better than we first thought: The data generated by MINIGOLEM achieves a very high accuracy – more files are found, more than 99 percent of the extracted constraints are logically equivalent, and the remaining constraints are not equivalent only because of a better, tighter description of the variability constraints by the parser.

Generally, we can see that both approaches have small, yet distinct limitations regarding the extraction of data.

By design, GOLEM can not sufficiently generate conditions for files which depend on more than one configurable option in the same Makefile, and for files which are only built when the corresponding option is disabled. This essentially is a consequence of the fact that GOLEM tries to achieve *statement coverage* in the Makefile but can only test every statement independently, agnostic of the *path* leading to this statement which might induce further constraints (e.g., from a surrounding `if{n}{def,eq}` block). Additionally, in some cases GOLEM infers extra constraints from KCONFIG that technically are not part of KBUILD.

On the other hand, it handles "non-standard" cases better than MINIGOLEM; whenever the constraints require the dynamic evaluation of variables or functions inside MAKE, the parsing-based, static approach is doomed to fail while the probing mechanism of GOLEM makes MAKE itself evaluate everything it needs. Furthermore, if files do not fit into the regular `obj-{y,m}` pattern but are rather generated through the use of custom MAKE rules, MINIGOLEM will not be able to detect them.

Luckily, both limitations are only rarely hit inside Linux, and the high accuracy over the course of five years and across all architectures indicates that the mentioned cases will remain exceptions.

## 5.4 (Additional) Defects Found With MINIGOLEM

In order to measure how the generated conditions compare, I also evaluate how the detection of `dead` and `undead` blocks is affected by the differences between the GOLEM and MINIGOLEM data.

To do this, I instruct UNDERTAKER to analyze every file in the Linux source code of the most recent release, `v3.19`, and to generate reports for them if any defects were found. The reports are then grouped by their defect class (see Section 4.2). This process is run with the models from GOLEM and then again with the models from MINIGOLEM.

An overview of the results can be seen in Table 5.3. The analysis shows that the number of `dead` (i.e., never compiled) blocks caused by conditions from KBUILD increases by 7.6 percent, while the number of blocks identified as `undead` (i.e., always compiled) grows by more than 13 percent. Additionally, 2 defects are no longer reported as `dead` when checking with missing symbols forced to `False`.

Nearly all of the defects which UNDERTAKER found additionally stem from the architecture-dependent code inside the different `arch/` directories, and target files for which more than one condition is specified inside the same Makefile (see the discussion in Section 5.3); for this case, the parsing approach provides better data for the analysis steps than GOLEM.

| Defect class (type) | with GOLEM | with MINIGOLEM | Change in percent |
|---|---|---|---|
| code (dead): | 46 | 46 | – |
| code (undead): | 47 | 47 | – |
| kconfig (dead): | 149 | 149 | – |
| kconfig (undead): | 156 | 156 | – |
| kbuild (dead): | 209 | 225 | **+7.6** |
| kbuild (undead): | 213 | 241 | **+13.1** |
| missing (dead): | 494 | 492 | −0.5 |
| missing (undead): | 99 | 99 | – |
| Total: | Σ 1,413 | Σ 1,455 | Σ +3.0 |

**Table 5.3** – Comparison of the defects found with UNDERTAKER in Linux version `v3.19` when using the file conditions from GOLEM or MINIGOLEM, respectively, grouped by defect class.

One example for a defect only found with MINIGOLEM is presented in Listing 5.2: Here, the source file `arch/x86/kernel/apic/x2apic_uv_x.c` contains a preprocessor block which should only be compiled when `CONFIG_SMP` is enabled (c.f. Listing 5.2a). When we look at the corresponding KBUILD Makefile (Listing 5.2b), we see that the file is only compiled when two other KCONFIG options have been enabled: Line 14 tells us the file will only be added to `obj-y`, the list of compiled object files, when `CONFIG_X86_UV` has been enabled (i.e., set to "y") in the configuration. Additionally, surrounding this statement is an `ifeq` block with its condition only becoming true if `CONFIG_X86_64` has also been set.

The corresponding, correct condition generated by MINIGOLEM for the source file is hence `CONFIG_X86_64 && CONFIG_X86_UV`.[17]

When UNDERTAKER now walks through the analysis steps described in Section 4.2, it finds no problem with the code itself (there are no nested `#ifdef` blocks posing any problem), and also adding the transitive KCONFIG information from `CONFIG_SMP` does not lead to a tautology or contradiction in the presence condition for the block.

However, when the conditions from the build system (and their transitive dependencies) are added, UNDERTAKER will report that the `#ifdef` can never become false, and thus, the corresponding block is `undead`. This is easily explained: Looking at the KCONFIG features in Listing 5.2c, we can see that `CONFIG_X86_UV` has (among others) a dependency on `CONFIG_NUMA` which itself has a dependency on `CONFIG_SMP`. This, through the transitive `CONFIG_NUMA` option, effectively means that `CONFIG_X86_UV` can only be enabled if `CONFIG_SMP` has been enabled earlier, and thus, the source file also can only be compiled when `CONFIG_SMP` is enabled.

[17] Note that GOLEM could not extract any preconditions for the file as it depends on more than one condition in the Makefile – see also Section 5.3.

```
205 static int uv_wakeup_secondary(int phys_apicid, unsigned long ↘
        start_rip)
206 {
207 #ifdef CONFIG_SMP
    [...]
226 #endif
227     return 0;
228 }
```

**(a)** Excerpt from `arch/x86/kernel/apic/x2apic_uv_x.c` with a preprocessor block depending on `CONFIG_SMP`.

```
11 ifeq ($(CONFIG_X86_64),y)
12 # APIC probe will depend on the listing order here
13 obj-$(CONFIG_X86_NUMACHIP)      += apic_numachip.o
14 obj-$(CONFIG_X86_UV)            += x2apic_uv_x.o
   [...]
18 endif
```

**(b)** Excerpt from the Makefile at `arch/x86/kernel/apic/Makefile`. The compilation of `x2apic_uv_x.c` depends on two KCONFIG features, `X86_64` and `X86_UV`.

```
409 config X86_UV
410         bool "SGI Ultraviolet"
411         depends on X86_64
412         depends on X86_EXTENDED_PLATFORM
413         depends on NUMA
    [...]
1204 config NUMA
1205         bool "Numa Memory Allocation and Scheduler Support"
1206         depends on SMP
```

**(c)** Excerpt from the KCONFIG file at `arch/x86/Kconfig`, describing the `NUMA` and `X86_UV` KCONFIG features.

**Listing 5.2** – Example for a `kbuild undead` defect from Linux `v3.16` which could only be found with the constraints from MINIGOLEM.

I fixed this defect by removing the `#ifdef` and its corresponding `#endif` and submitted a patch to the Linux maintainers who accepted it into the mainline Linux kernel.[18]

[18] The corresponding commit can be found at `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=8091c1f8ea2374695c105591179b1269fb5f2fbb`.

## 5.5 Adaptability – BUSYBOX and COREBOOT

In the last section of this chapter, I want to evaluate how well the plug-in modules for BUSYBOX and COREBOOT work, both in terms of extraction speed as well as in terms of quality of the extracted conditions. All measurements were run on the same machines where I also tested Linux, comprising a Core i5-4590 CPU with 3.3 GHz and 16 GiB of RAM.

### BUSYBOX

For the evaluation of BUSYBOX, I use the latest stable release, version `1.23.2` which is available from the GIT repository.[19]

The extraction of the KBUILD conditions takes 44.1 seconds with GOLEM while MINIGOLEM finishes in 0.69 seconds – over **60 times** as fast.

The comparison of the conditions generated by both tools is nothing short of impressive (see also Table 5.4): GOLEM emits configurational constraints for a total of 549 files, MINIGOLEM only has one entry less (548). When we look at this "file", however, we can see that GOLEM generated a bogus entry for a file called "`rm`" with many conditions associated to it – this is probably another parsing error of the MAKE output in the extraction logic of GOLEM.

| | | |
|---|---|---|
| Number of files found by GOLEM ($F_g$) | 549 | |
| Number of files found by MINIGOLEM ($F_m$) | 548 | |
| Files covered by both approaches ($F_g \cap F_m$) | 548 | |
| ⇒ Files only covered by GOLEM | 1 | |
| ⇒ Files only covered by MINIGOLEM | 0 | |
| Number of equivalent formulas for files in $F_g \cap F_m$ | 547 | (**100 %**) |
| ⇒ Number of differing formulas | 0 | |

**Table 5.4** – Statistics for the extracted KBUILD conditions by GOLEM and MINIGOLEM in BUSYBOX version `1.23.2`.

### COREBOOT

In COREBOOT, GOLEM fails to extract data from newer versions, as the developers changed the build system to require their own version of the compiler and other tools. MINIGOLEM, on the other hand, continues to work on these versions without any problems. For a comparison of the quality of the conditions, however, I need a version of COREBOOT where GOLEM can generate data; therefore, I use the same

[19] The repository is located at `git://git.busybox.net/busybox`.

version which Hengelein [10] used in his evaluation. This is at commit id `bef3d347e` from late 2012.

For this version, it takes GOLEM 358 seconds to extract its conditions, MINIGOLEM is done after around 0.76 seconds, which represents a speedup by a factor of **470**.

A summary of the extracted data by GOLEM and MINIGOLEM is given in Table 5.5.

| | | |
|---|---|---|
| Number of files found by GOLEM ($F_g$) | 3,974 | |
| Number of files found by MINIGOLEM ($F_m$) | 3,150 | |
| Files covered by both approaches ($F_g \cap F_m$) | 3,012 | |
| ⇒ Files only covered by GOLEM | 962 | |
| ⇒ Files only covered by MINIGOLEM | 138 | |
| Number of equivalent formulas for files in $F_g \cap F_m$ | 2,992 | (99.34 %) |
| ⇒ Number of differing formulas | 20 | |

**Table 5.5** – Statistics for the extracted KBUILD conditions by GOLEM and MINIGOLEM in COREBOOT version 4.0 (commit `bef3d347e`).

Here, we can see that GOLEM extracts about 800 more conditions than MINIGOLEM (3,974 vs. 3,150), and that slightly more than 3,000 files appear in both data sets. As COREBOOT Makefiles quite often include common files (e.g., CPU specific code for certain processor revisions), GOLEM also (wrongly) produces multiple entries for the same file in its output as it fails to properly match them – MINIGOLEM only emits one entry per file.

A closer inspection of the additional files with conditions in GOLEM reveals that all (without any exception) of these files are *object files* which are located inside the `build/` subdirectory; they are intermediate products generated during the build process and subsequently are not in scope of a parser. On the other hand, UNDERTAKER does not need any data for *object files*; thus, these files can safely be disregarded.

The conditions for the 20 files for which GOLEM and MINIGOLEM achieve different results all correspond to cases where either the probing strategy fails due to multiple constraints for the source file in the same Makefile or a file is built only when a KCONFIG option is disabled – thus, MINIGOLEM correctly provides tighter constraints on these files.

Lastly, the 138 files only found by MINIGOLEM all correspond to mainboard-specific files inside the `src/mainboard/` directory. I assume that GOLEM does not properly detect all possible directories and thus fails to descend into some of them, leading to missing data for the files inside these directories.

# 6 Conclusion

Variability in system software projects is often spread across different layers – in Linux, configurable options can be specified in the KCONFIG language, and are subsequently used in KCONFIG to describe dependencies between options, in the build system KBUILD to include or exclude files from the compilation process as well as in the source files through the C preprocessor to include or exclude code on a *fine-grained* level.

Due to the dispersion of information across the different layers and involved files, and due to the complexity of the dependencies involved, developers are prone to make errors. In previous work, Sincero [18], Tartler [20] and Dietrich [7] have developed the UNDERTAKER toolchain which can extract information from all layers and detect defects caused by erroneous use of configurable options. Using their work as a basis, Rothberg [17] presented UNDERTAKER-CHECKPATCH, an easy-to-use tool which allows a developer to check a PATCH file for configurability defects before submitting it into the Linux kernel.

UNDERTAKER-CHECKPATCH, however, could previously not use information from KBUILD, as the runtime of the GOLEM extractor exceeds **three hours** per architecture on the most recent Linux release, rendering it useless for integration into a developers workflow.

To overcome this obstacle, I have developed MINIGOLEM, a parsing-based extractor for variability information from KBUILD which can process one architecture in just over **one second**. Contrary to our initial expectations, the extracted data is very accurate: The tool produces constraints for more than 97 percent of all files in Linux – even more files than GOLEM can reach. A quantitative comparison of the conditions for files which are covered by both extractors revealed that more than 99 percent of these conditions are logically equivalent. For the remaining files, the conditions from MINIGOLEM are mostly more precise than the conditions from GOLEM, as the Linux KBUILD files sometimes violate the assumptions upon which GOLEM's probing

strategy is based. Furthermore, my approach is highly robust: The quality of the extracted data is equally high for all Linux versions released in the last five years, ranging from `v2.6.32` to `v3.19`.

The higher accuracy also results in an increase of KBUILD-related defects found by UNDERTAKER: The number of `dead` blocks increases by 7.6 percent, while more than 13 percent more `undead` blocks can be detected.

With the integration of MINIGOLEM into the UNDERTAKER toolchain and into UNDERTAKER-CHECKPATCH, I also designed an experiment which evaluates the changes to the `linux-next` development tree every day. This now allows us to catch variability-related errors immediately after they first appear – we have already received highly positive responses regarding our work from developers and maintainers, and continuously work on improving the reports.

Moreover, through its modular design, MINIGOLEM can easily be ported to other software projects: I demonstrated this by providing plug-in modules not only for Linux, but also for BUSYBOX and COREBOOT. Again, I was able to show that the extracted constraints are equally accurate or even more precise than the constraints from GOLEM.

## Future Work

Currently, when the parsing process encounters an expression it can not evaluate, it simply skips the current line; if the unparseable expression is inside the condition of an `if{n}{def,eq}` statement, it skips all code which depends on this statement as we can not reason about the impact of the unparseable statement on the variability. Even though this currently only affects around 50 lines in Linux `v3.19`, it might be interesting to see if it is possible to (a) reliably detect the reason why a particular line is failing and (b) switch to an alternative evaluation method which is better suited for these types of statements.

Another point which could be improved is the reporting of defects by the experiment running on `linux-next`: In its current state, the output of UNDERTAKER-CHECKPATCH might not be entirely comprehensible for a developer – this is mostly caused by the inherent complexity of some defect's causes as they can stem from the interaction of many different KCONFIG features, leading to huge formulas with hundreds of KCONFIG features involved. While UNDERTAKER-CHECKPATCH already supports the generation of a *minimally unsatisfiable subformula* to identify the main culprits for `dead` defects, the output is often cryptic and hard to properly understand for developers who might not be familiar with the format of the propositional formulas involved.

# List of Acronyms

| | |
|---|---|
| **LKM** | loadable kernel module |
| **SAT** | boolean satisfiability problem |

# List of Listings

# List of Figures

# List of Tables

# References


[1] Thorsten Berger et al. “A Study of Variability Models and Languages in the Systems Software Domain.” In: *IEEE Transactions on Software Engineering* 39.12 (Dec. 2013), pp. 1611–1640. ISSN: 0098-5589. DOI: 10.1109/TSE.2013.34.

[2] Thorsten Berger et al. *Feature-to-Code Mapping in Two Large Product Lines*. Tech. rep. University of Leipzig (Germany), University of Waterloo (Canada), IT University of Copenhagen (Denmark), 2010.

[3] 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.

[4] *BusyBox Project Homepage*. URL: http://www.busybox.net/ (visited on 02/10/2015).

[5] *CADOS - Configurability Aware Development of Operating Systems*. FAU Erlangen-Nuremberg, 2015. URL: https://www4.cs.fau.de/Research/CADOS/.

[6] *Coreboot Project Homepage*. URL: http://www.coreboot.org/ (visited on 02/10/2015).

[7] Christian Dietrich. “A Robust and Portable Approach for extracting Build-System Variability.” Bachelor Thesis. University of Erlangen, July 2012.

[8] Christian Dietrich and Daniel Lohmann. “The Dataref Versuchung: Saving Time Through Better Internal Repeatability.” In: *SIGOPS Operating Systems Review* 49.1 (2015), pp. 51–60.

[9] 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.

[10] Stefan Hengelein. “Variabilitätsanalysen in Coreboot.” Bachelor Thesis. University of Erlangen, Jan. 2013.

[11] Thorsten Berger and Steven She. *Google Code Project: various variability extraction and analysis tools*. URL: http://code.google.com/p/variability/ (visited on 02/13/2015).

[12] *Limboole*. URL: http://fmv.jku.at/limboole/ (visited on 02/13/2015).

[13] *snadi / Makex – Bitbucket*. URL: https://bitbucket.org/snadi/makex (visited on 02/13/2015).

[14] Sarah Nadi and Ric Holt. “The Linux kernel: a case study of build system variability.” In: *Journal of Software: Evolution and Process* 26.8 (2014), pp. 730–746.

[15] Sarah Nadi and Richard C. Holt. “Make it or Break it: Mining Anomalies from Linux Kbuild.” In: *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE ’11)*. 2011, pp. 315–324. DOI: 10.1109/WCRE.2011.46.

[16] Sarah Nadi and Richard C. Holt. “Mining Kbuild to Detect Variability Anomalies in Linux.” In: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR ’12)*. (Szeged, Hungary, Mar. 27–30, 2012). Ed. by Tom Mens, Yiannis Kanellopoulos, and Andreas Winter. Washington, DC, USA: IEEE Computer Society Press, 2012. ISBN: 978-1-4673-0984-4. DOI: 10.1109/CSMR.2012.21.

[17] Valentin Rothberg. “Years of Variability Bugs in Linux – How to Avoid them.” Master’s Thesis. University of Erlangen, Sept. 2014.

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

[19] 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.

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

[21] 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.

[22] *CADOS Undertaker*. URL: https://undertaker.cs.fau.de/ (visited on 03/11/2015).

[23] *VAMOS - Variability Management in Operating Systems*. FAU Erlangen-Nürnberg, 2015. URL: https://www4.cs.fau.de/Research/VAMOS/.

[24] Manuel Zerpies. “Variabilitätsanalyse in Busybox.” Bachelor Thesis. University of Erlangen, Jan. 2013.