

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Patrick Plagwitz

Web-based Visualization of Configuration Defects in Linux Source Code

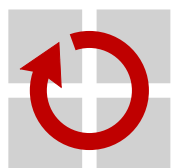
Bachelorarbeit im Fach Informatik

31. Dezember 2015

Please cite as:

Patrick Plagwitz, "Web-based Visualization of Configuration Defects in Linux Source Code" Bachelor's Thesis, University of Erlangen, Dept. of Computer Science, December 2015.

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



Web-based Visualization of Configuration Defects in Linux Source Code

Bachelorarbeit im Fach Informatik

vorgelegt von

Patrick Plagwitz

angefertigt am

**Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme**

**Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg**

Betreuer: **Andreas Ziegler, M.Sc.
Valentin Rothberg, M.Sc.**

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

Beginn der Arbeit: **August 2015**
Abgabe der Arbeit: **31. Dezember 2015**

Erklärung

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

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.

(Patrick Plagwitz)

Obermichelbach, 31. Dezember 2015

ABSTRACT

With over 15,000 selectable features, Linux is a prime example of highly configurable system software. However, this configurability is implemented with a mixture of MAKE scripts, C preprocessor (CPP) `#ifdef` annotations, and a separate feature model written in the specially conceived KCONFIG language. The extremely loose coupling between these three stages of the Linux build system can possibly lead to contradictory or redundant `#ifdef` conditions in the program code. UNDERTAKER-CHECKPATCH, based on the UNDERTAKER toolchain, can find and analyze these inconsistencies, called defects, resulting in automatically generated defect reports. However, textual explanations must still be written manually for these to be useful to Linux developers – a repetitive effort that warrants improvement.

In this thesis, I develop the program WUNDERTAKER that explains defects in an automated way via graphical means. This allows for an enhanced form of UNDERTAKER-CHECKPATCH's output that is both easier to understand and reduces the amount of text that needs to be written manually. Designed as a web application, WUNDERTAKER employs appropriate visualization techniques, some of them tested in other GUIs or examined in studies. I then show that WUNDERTAKER is fast enough and robust enough to support an ongoing experiment where UNDERTAKER-CHECKPATCH's defect reports are sent via e-mail to Linux developers, two of them pointing out its fitness for the task. The final result is a usable program that could be extended into a multitude of directions in further work.

KURZFASSUNG

Mit über 15.000 auswählbaren Features ist Linux ein vorrangiges Beispiel von stark konfigurierbarer Systemsoftware. Diese Konfigurierbarkeit ist jedoch implementiert durch eine Mischung aus MAKE-Skripten, C-Präprozessor (CPP)-`#ifdef`-Annotationen und einem getrennten Feature-Modell, das in der speziell konzipierten KCONFIG-Sprache geschrieben ist. Die extrem lose Kopplung zwischen diesen drei Abschnitten des Linux-Buildsystems kann unter Umständen zu widersprüchlichen oder redundanten `#ifdef`-Bedingungen im Programmcode führen. UNDERTAKER-CHECKPATCH, basierend auf der UNDERTAKER-Toolchain, kann diese Widersprüchlichkeiten, die Defekte genannt werden, finden und analysieren. Das führt zu automatisch generierten Defekt-Reports. Für diese müssen trotzdem noch textbasierte Erklärungen manuell geschrieben werden, damit sie für Linux-Entwickler einen Nutzen haben – ein sich wiederholender Prozess, der es rechtfertigt, verbessert zu werden.

In dieser Arbeit entwickle ich das Programm WUNDERTAKER, das Defekte durch grafische Mittel automatisiert verdeutlicht. Das ermöglicht eine verbesserte Form der Ausgabe von UNDERTAKER-CHECKPATCH, die sowohl einfacher zu verstehen ist als auch die Menge an Text reduziert, die manuell geschrieben werden muss. WUNDERTAKER ist entworfen als Webanwendung und setzt geeignete Visualisierungstechniken ein, manche davon erprobt in anderen GUIs oder untersucht in Studien. Dann zeige ich, dass WUNDERTAKER schnell genug und robust genug ist, um ein momentan laufendes Experiment zu unterstützen, wo die Defekt-Reports von UNDERTAKER-CHECKPATCH über E-Mail an Linux-Entwickler gesendet werden. Zwei dieser Entwickler weisen darauf hin, dass es für diese Aufgabe geeignet ist. Das Endergebnis ist ein benutzbares Programm, das in folgenden Arbeiten in eine Vielzahl von Richtungen weiter erweitert werden könnte.

CONTENTS

| | |
|--|------------|
| Abstract | v |
| Kurzfassung | vii |
| 1 Introduction | 1 |
| 2 Fundamentals | 3 |
| 2.1 The Linux build process | 3 |
| 2.1.1 KCONFIG | 3 |
| 2.1.1.1 Mode of operation | 3 |
| 2.1.1.2 Translation into a propositional formula | 5 |
| 2.1.2 KBUILD | 6 |
| 2.1.2.1 Mode of operation | 6 |
| 2.1.2.2 Translation into a propositional formula | 7 |
| 2.1.3 CPP | 7 |
| 2.1.3.1 Mode of operation | 7 |
| 2.1.3.2 Translation into a propositional formula | 8 |
| 2.1.3.3 Problems with the CPP | 9 |
| 2.2 UNDERTAKER toolchain | 10 |
| 2.2.1 UNDERTAKER-CHECKPATCH | 11 |
| 2.2.2 Defect classes | 11 |
| 2.2.2.1 <i>code</i> defects | 11 |
| 2.2.2.2 <i>kconfig</i> defects | 12 |
| 2.2.2.3 <i>kbuild</i> defects | 13 |
| 2.2.2.4 <i>missing</i> defects | 13 |
| 2.2.2.5 <i>no_kconfig</i> defects | 13 |
| 2.3 Related work | 13 |
| 3 WUNDERTAKER | 15 |
| 3.1 Environment | 15 |

| | | |
|--------------|--|-----------|
| 3.1.1 | WUNDERTAKER as an UNDERTAKER GUI | 15 |
| 3.1.2 | WUNDERTAKER as a GIT repository view | 16 |
| 3.2 | The didactics of defect reports | 16 |
| 3.2.1 | <i>missing</i> | 16 |
| 3.2.2 | <i>code</i> | 17 |
| 3.2.3 | <i>kconfig</i> and <i>kbuild</i> | 18 |
| 3.3 | Requirements | 19 |
| 3.4 | Implementation | 19 |
| 3.4.1 | Visualization techniques | 19 |
| 3.4.1.1 | The code view | 19 |
| 3.4.1.2 | The folder view | 22 |
| 3.4.2 | Colors | 23 |
| 3.4.3 | Software that WUNDERTAKER uses | 25 |
| 3.4.3.1 | Vertical Design | 26 |
| 3.4.3.2 | Horizontal Design | 26 |
| 4 | Evaluation | 29 |
| 4.1 | Targets and Procedure | 29 |
| 4.2 | Results | 31 |
| 5 | Conclusion | 35 |
| Lists | | 37 |
| | List of Acronyms | 37 |
| | List of Figures | 38 |
| | List of Tables | 40 |
| | List of Listings | 42 |
| | Bibliography | 44 |

INTRODUCTION

With over 15,000 selectable features, Linux exemplifies highly and compile-time configurable system software. Its KCONFIG language is used to model Linux features and constraints on them. However, the actual implementation of features is accomplished by annotations made around program code with the C preprocessor (CPP), as well as by a coarser-grained kind of conditional compilation defined in the build scripts of Linux.

Two variability models are thus induced whose effective separation from one another is so profound that programs in the build process working with one model can be entirely agnostic of programs working with the other. Their sole connection is a Linux configuration variant – the result of a user selecting or deselecting features from the KCONFIG model, and the associated KCONFIG tool checking the selection for consistency. After such a variant has been generated, it is passed on to the CPP and the KBUILD system. At this point, nothing stops the variant from contradicting `#ifdef` conditions or KBUILD rules, both governing the conditional compilation of feature code. It is also possible that the rules are superfluous when considering the variant. These possibilities lead to *dead* or *undead* code that is never or always compiled, respectively, despite being annotated. It follows that manual work is required to keep the definitions of both models in sync to prevent these kinds of variability defects.

The lack of tools that support this task is what motivated the UNDERTAKER toolchain, introduced by Tartler et al. from the VAMOS¹ project, later the CADOS² project (see [Tar+11]). UNDERTAKER joins both models by extracting information from KCONFIG files, build scripts, and code files, resulting in a propositional formula for each fragment of annotated code which is then used to reason about what happens to the fragment during conditional compilation. The toolchain thus gives rise to several practical applications, first of all being its dead code analysis that automatically finds dead and undead CPP `#ifdef` blocks. Because this involves UNDERTAKER checking the propositional formulas for satisfiability, UNDERTAKER can, in the general case, provide only limited information on what led to the defect and how to fix it – problems that have been analyzed in [Nad+13]. Valentin Rothberg extended the UNDERTAKER toolchain with UNDERTAKER-CHECKPATCH in [Rot14] which can detect defects that got newly introduced by changes made to the Linux code. Being employed in an

¹Variability Management in Operating Systems

²Configurability Aware Development Of Operating Systems

experiment where a Linux development tree is periodically checked for new defects to inform the responsible developers, UNDERTAKER-CHECKPATCH has a feature that further analyzes defects so as to act as sort of a front end for the process.

However, the only information available at this point is still just a propositional formula and an automated analysis of it. If Linux developers received this in raw form in an e-mail, it would shift the burden of evaluating UNDERTAKER-CHECKPATCH's output to the recipients of the e-mails, which is not acceptable, even though the output would be understandable for them in many cases. Also, the e-mails would miss an explanation of relevant context, such as the portion of the patch that introduced the defect and the exact position of the defective `#ifdef` block. It is therefore vital for them to be useful to developers that these information are enriched with a manually assembled textual description. There are two problems with this: First, describing `#ifdef` blocks by naming line numbers and explaining nesting relationships between them is a tedious task that might become repetitive very quickly. Second, a text is very likely not the most optimal way to explain the context of a defect.

Much more interesting would be a graphical view that relieves the CADOS people of having to write the most basic of explanations and that can simultaneously provide a superior version of them. My work is aimed at developing the program WUNDERTAKER that implements such a Graphical User Interface (GUI), presenting developers with a familiar, syntax-highlighted version of a code file and overcoming the lack of intuitive context Linux developers face if the manual enhancements of defect reports are solely text-based. In the GUI, `#ifdef` blocks could be visualized with specialized techniques so that as much information about defects as possible is available at first glance. Designed as a web application, my GUI can help with understanding not only Linux code but arbitrary code that contains `#ifdef` directives. So developers can explore the distribution of defects found in a project on their own, it offers a browsable view of a GIT repository and so can be useful to any open source project using CPP-based variability and GIT.

Since variability defects arise on account of the way the Linux build system operates and is used, I will begin in Chapter 2 by explaining its three stages and how UNDERTAKER extracts propositional formulas from them. Next, in Section 2.2, I will discuss UNDERTAKER's dead code analysis, its defect classification, and that UNDERTAKER-CHECKPATCH's output is the foundation of WUNDERTAKER. Lastly, I will briefly reference and summarize literature that reports on relevant visualization techniques. Chapter 3 will discuss how WUNDERTAKER can fit technically into the UNDERTAKER toolchain and how it can support it, step by step gathering a list of requirements. Section 3.4 will show how a combined use of visualization techniques allows to fulfill these requirements, and Chapter 4 will evaluate them and demonstrate their feasibility.

FUNDAMENTALS

2

2.1 The Linux build process

2.1.1 KCONFIG

2.1.1.1 Mode of operation

The first step when compiling Linux into runnable binary files is to generate a configuration variant by selecting features from the KCONFIG model. The model is defined in `Kconfig` files that reside in the Linux tree and are written in the KCONFIG language. The following explanation of KCONFIG is not as exhaustive as the one given in [Hen15], but it covers the most important language constructs that influence UNDERTAKER's formulas. Many constructs not mentioned here can be eliminated by a pre-processing step [ZK10] or are not relevant to a dead code analysis.

Each KCONFIG feature has a type that determines the set of values it is allowed to take on. A feature of type `bool` represents a binary choice: whether to compile it into the kernel or not. Tristate features may assume a third state, designating the feature as a module that can be loaded at run

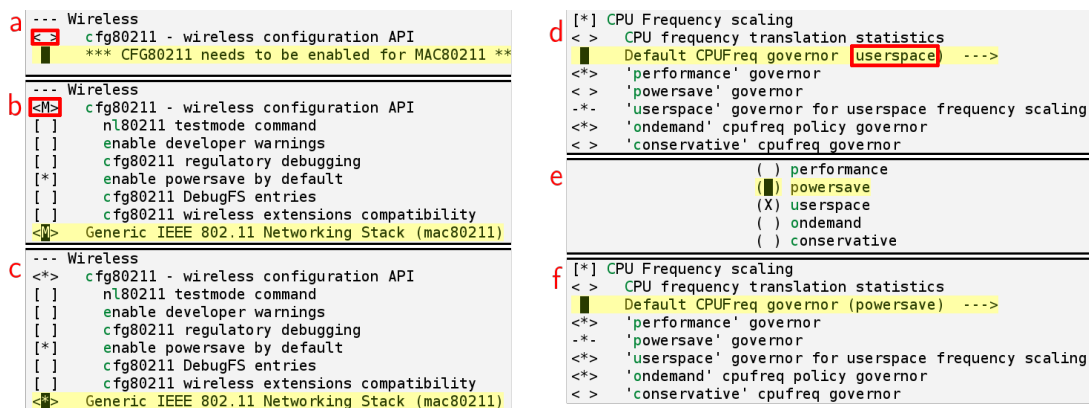


Figure 2.1 – `make menuconfig` TUI of Linux v4.3 – one of the KCONFIG front ends. Each entry represents a feature.

time – a Loadable Kernel Module (LKM). The three values are written as `y` or `*` for “selected”, `M` or `m` for “selected as an LKM”, and `n` or `_` for “not selected”. They can also be seen as numbers 2, 1, and 0, in this order. Types `string` and `int` (integer) exist, too, but these are impossible to process correctly with a purely boolean SAT checker as employed by UNDERTAKER, so I will ignore them.

Listing 2.1 shows the definition of the feature `MAC80211` in one of the `Kconfig` files. It is introduced in the first line by the `config` keyword, and the second line contains the `tristate` statement that designates it as a tristate feature. It becomes evident that `KCONFIG`’s syntax is generally line-based; each line represents the use of a statement, written as a keyword followed by a value, whose effect is influenced by certain statements, like `config`, that come before it.

```

1 config MAC80211
2     tristate "Generic IEEE 802.11 Networking Stack (mac80211)"
3     depends on CFG80211

```

Listing 2.1 – Lines 1–3 from `net/mac80211/Kconfig` of v4.3.

Figure 2.1 shows the `make menuconfig` front end of `KCONFIG`, which generates a hierarchical menu of features from the `KCONFIG` model and enforces its constraints by limiting a user’s actions. Figure 2.1a/b/c demonstrate the effects of the `depends on` statement; Figure 2.1d/e/f those of the `select` statement and the `choice...endchoice` block. The `depends on` statement, used in line 3 in Listing 2.1, disallows a feature to be selected as long as its dependencies are not selected. In Figure 2.1a, `CFG80211` is deselected and the menu is almost empty. In Figure 2.1b, where it is selected as an LKM, `MAC80211` and other features become visible because they depend on `CFG80211`. Here, `MAC80211` is chosen as an LKM as well. In fact, it could not be selected as `*` because if a dependency of a feature is to be compiled as an LKM, so must be the feature itself. In Figure 2.1c with `CFG80211` chosen as `*`, `MAC80211` can too finally take on both `*` and `M` as a value. A more precise definition of `depends on` is hence that A depending on B has the effect of the value of B imposing an upper bound on the value of A.

Listing 2.2 shows five features being defined within a single `choice...endchoice` block. They are there for changing the kernel’s default CPU frequency scaling governor. Although five governors are available, only one can be the default setting. Figure 2.1d shows how `make menuconfig` visualizes this mutual exclusivity: by showing the default governor as a single menu entry. Additionally, each of the default governor definitions includes a statement selecting the actual governor. The effect of this can be seen in Figure 2.1d and f, where the `userspace` and `powersave` governor features, respectively, have hyphens instead of brackets around their selected values, signaling that whatever governor is chosen as default in Figure 2.1e cannot be deselected. This “force on” acts as a reverse to the normal `depends on` option, imposing a *lower* bound on the value of a selected feature, and is employed when it is undesirable that features get hidden in the menu.

```

choice
    prompt "Default CPUFreq governor" [...]
config CPU_FREQ_DEFAULT_GOV_PERFORMANCE
    bool "performance"

```

```

select CPU_FREQ_GOV_PERFORMANCE [...]
config CPU_FREQ_DEFAULT_GOV_POWERSAVE [...]
config CPU_FREQ_DEFAULT_GOV_USERSPACE [...]
config CPU_FREQ_DEFAULT_GOV_ONDEMAND [...]
config CPU_FREQ_DEFAULT_GOV_CONSERVATIVE [...]
endchoice

```

Listing 2.2 – Lines 49–105 from `drivers/cpufreq/Kconfig` of v4.3.

In KCONFIG, it is possible to form expressions with arbitrary combinations of features and connectives based on tristate logic. For instance, given features A, B, and C, one could write `depends on B && !C` or make `select` conditional, e.g., `select A if B`.

2.1.1.2 Translation into a propositional formula

To connect the KCONFIG constraints explained so far with others from the build system, UNDERTAKER follows a procedure to translate them to a propositional formula ϕ_{KCONFIG} introduced by Zengler and Küchlin [ZK10]. First, tristate values are represented as two bits, (a_0, a_1) , as it is shown in Table 2.1, yielding four possible combinations. For a tristate feature, only three are needed, so one constructs a set \mathcal{C}_O of constraints that make the fourth possibilities illegal. If, for instance, there is a tristate feature A, encoded with (a_0, a_1) , then the set is

$$\mathcal{C}_O = \{\neg a_0 \vee \neg a_1\},$$

and it can be added to ϕ_{KCONFIG} as a conjunction $\bigwedge_{t \in \mathcal{C}_O} t$.

The second step is to find two projection functions $\pi_{1,2}$ that map from a tristate expression to the two-bit encoding of its value when all connectives are resolved. For simple symbols like A it is $\pi_{0,1}(A) = a_{0,1}$. Conjunctions of expressions $e = e_0 \ \&\& \ \dots \ \&\& \ e_n$ can be explained as follows. They cannot be y if any $e_i = n$ or if any $e_i = m$ and hence it would be $\pi_0(e_i) = 0$. Therefore, it must be

$$\pi_0(e) = \bigwedge_{e_i} \pi_0(e_i).$$

The conjunction can only be m if all $e_i \neq n$, and if any $e_i = m$, because otherwise e would then be y and its second bit 0. Hence, reflecting the conditions in this order, it must be

$$\pi_1(e) = \bigwedge_{e_i} (\pi_0(e_i) \vee \pi_1(e_i)) \wedge \bigvee_{e_i} \pi_1(e_i).$$

| a_0 | a_1 | Tristate meaning | Treatment in <code>autoconf.h</code> |
|-------|-------|------------------|---|
| 0 | 0 | n | No symbols defined |
| 0 | 1 | m | <code>CONFIG_A_MODULE</code> is defined |
| 1 | 0 | y | <code>CONFIG_A</code> is defined |
| 1 | 1 | illegal | — |

Table 2.1 – Encoding (a_0, a_1) of a tristate symbol A. The last column is relevant in Section 2.1.3.1.

With analogous functions $\pi_{1,2}$ for negations and disjunctions (see [ZK10]), one can now simulate a `depends on e` option declared for a symbol s encoded as (s_0, s_1) . Remember that the value of e is an upper limit for the value of s , so s is not constrained at all if e is `y`. In other cases, s can be

$$\begin{aligned} & \text{n or m if } e \text{ is m,} & \pi_1(e) \rightarrow \neg s_0; \\ & \text{and just n if } e \text{ is n,} & (\neg \pi_0(e) \wedge \neg \pi_1(e)) \rightarrow (\neg s_0 \wedge \neg s_1). \end{aligned}$$

As to `select` options, which impose a lower limit on the `selected` feature, s' selected by a symbol s under the condition that e is `y` would not be constrained if s is `n`. But it would have to be

$$\begin{aligned} & \text{m or y if } s = \text{m and } e \text{ is y,} & s_1 \wedge \pi_0(e) \rightarrow s'_0 \vee s'_1; \\ & \text{and only y if } s = \text{y and } e \text{ is y,} & s_0 \wedge \pi_0(e) \rightarrow s'_0. \end{aligned}$$

The constraints in the right columns can be added to ϕ_{KCONFIG} as a conjunction $\bigwedge_{d \in \mathcal{C}_{\text{depends on}}} d \wedge \bigwedge_{s \in \mathcal{C}_{\text{select}}} s$. Lastly, given mutually exclusive KCONFIG symbols s_1, \dots, s_n in a choice block, if one s_i is `y`, no other s_i may be `y`, which can be written in a straightforward way as

$$\bigwedge_{s_i} \left(\pi_0(s_i) \rightarrow \bigwedge_{s_j, j \neq i} \neg \pi_0(s_j) \right)$$

and added to ϕ_{KCONFIG} as a conjunction, as well.

Suppose now a variability model with a tristate feature F00 and a boolean feature BAR, encoded as (f_0, f_1) and (b_0, b_1) , where the former `depends on` the latter. The extended formula $\phi_{\text{KCONFIG}} \wedge f_0$ should then be satisfiable since it reflects that F00 can be chosen at all. Its satisfiability also demonstrates that an `#ifdef` block with condition `CONFIG_F00` is not dead because there are configuration variants where it is compiled. In contrast, $\phi_{\text{KCONFIG}} \wedge f_0 \wedge \neg b_0$, simulating an `#ifdef` block with condition `CONFIG_F00 && !CONFIG_BAR`, should *not* be satisfiable since there is no legal configuration variant where F00 is chosen but BAR is not.

2.1.2 KBUILD

2.1.2.1 Mode of operation

Connecting an `#ifdef` condition with the KCONFIG model is not enough to simulate conditional compilation in a propositional formula because it is not a given that a file will be compiled at all. Rather, KBUILD scripts in files called `Makefile` or `Kbuild` drive a build. They are written in the MAKE language and distributed across the Linux tree. Normally, the MAKE program acts upon a set of declarative rules composed of sources, a target, and a script. However, most of the Makefiles in Linux reside in subdirectories and are merely responsible for setting the MAKE variables `obj-y`, `obj-m`, and `obj-n`, thereby communicating to a top-Makefile the list of files that should be built statically, as LKMs, or not at all, respectively.

To see how this takes place, assume that a developer wants to add a new interface for wireless adapters called `fooadapter`. He places the code file `fooadapter.c` into `net/wireless`. Instructing KBUILD with `obj-$(CONFIG_F00_ADAPTER)+= fooadapter.o` in `net/wireless/Makefile`, he

wants to make `fooadapter.o`'s existence in the final build dependent on whether a feature `F00_ADAPTER`, in turn defined in `net/wireless/Kconfig`, is `y`, `m`, or `n`. The `Makefile` knows about this value because a file `auto.conf` is automatically generated by `KCONFIG` based on the user's feature selection and included beforehand. Besides files, the three lists may also contain directories, making `KBUILD` search them for `Makefiles`.

2.1.2.2 Translation into a propositional formula

To capture `KBUILD`'s implicit conditional compilation in a propositional formula ϕ_{KBUILD} , one needs to find those `KCONFIG` features X that appear in lines of the form `obj-$(CONFIG_X) += x.o`. All `#ifdef` blocks in the file whose build product `x.o` is, must then include X in their formula. However, Dietrich et al. explain in [Die+12] that the `MAKE` language is, in general, much more powerful, even Turing complete, because it has features such as command substitution. Under these circumstances, it is much more difficult to decide whether a certain fragment of `MAKE` code does or does not extend one of the lists `obj-y`, `obj-m`, and `obj-n`, given a configuration variant.

However, Dietrich et al. find a robust solution to this problem. In principle, it finds all configuration variants that cause a file to be compiled, and adds a disjunction of them all,

$$\bigvee_{v \in V} \left(\bigwedge_{\text{feature} \in v} \text{feature} \right),$$

as a conjunction to ϕ_{KBUILD} . This process is implemented as a tool called `GOLEM`, integrated into the `UNDERTAKER` toolchain, and works by actually running `MAKE` with a custom `Makefile` that traverses the Linux tree in the same way `KBUILD` would and evaluating the three `MAKE` lists afterwards. `GOLEM` does this “probing” first with an empty configuration variant in which no features are selected. Then it scans all sub-`Makefiles` for any variables they use that start with `CONFIG_`, i.e., the features the file potentially depends on, and extends the initial configuration variant with them, one after another, probing again after each extension and recursing into subdirectories. When a build product is found in `obj-y` or `obj-m` after probing, the variant v used in this step is known to belong to V .

The major downside of this approach is its long runtime. Andreas Ruprecht notes in [Rup15] that it takes over 3 hours to run `GOLEM` on a modern quad core machine on Linux v3.19 for a single architecture. This motivated him to again take up a method based on regular expressions and utilizing the fact that the majority of Linux `Makefiles` do after all use a safe subset of the `MAKE` language. As a result, his `MINIGOLEM` tool has a runtime of around a second which makes it much more usable for analyses where it must be invoked often, like in `UNDERTAKER-CHECKPATCH` (see Section 2.2.1).

2.1.3 CPP

2.1.3.1 Mode of operation

`KBUILD` can include or exclude entire compilation units, but Linux developers' need for finer-grained variability is met by the `CPP` which handles arbitrary fragments of code within a file. The `CPP` has a

rather disparate set of uses; only macro definition and the `#ifdef` family of directives is required for variability implementation [Sin+10]. By surrounding code with an `#ifdef` block, its existence in the final token stream to the C compiler is made dependent on the header of the block which contains a condition like `#ifdef CONFIG_FOO`, testing for a KCONFIG feature. So the CPP has access to the KCONFIG selection, KCONFIG generates an `autoconf.h` from a given configuration variant, analogous to the `auto.conf` it makes for KBUILD. Only here, tristate values are encoded like in ϕ_{KCONFIG} (see Table 2.1): Two CPP symbols act as bits by being defined or not. At its top, Listing 2.3 shows such a `autoconf.h` where MAC80211 is selected as an LKM, and the symbol hence suffixed with `_MODULE`. At the bottom is a variant where MAC80211 is set to `y`, and the symbol is not suffixed.

```
#define CONFIG_MAC80211_MODULE 1
#define CONFIG_ARCH_USES_PG_UNCACHED 1
```

```
#define CONFIG_MAC80211 1
#define CONFIG_ARCH_USES_PG_UNCACHED 1
```

Listing 2.3 – Snippets from `include/generated/autoconf.h` I had generated from the variants active in Figure 2.1b and Figure 2.1c, respectively. Note that in an `autoconf.h`, all symbols are prefixed with `CONFIG_`.

2.1.3.2 Translation into a propositional formula

At the end of Section 2.1.1.2, I already alluded to how `#ifdef` conditions can be incorporated into our formula. CPP `#ifdef` conditions can basically be taken and integrated straight into a formula ϕ_{CPP} which is then added to the existing formula ϕ_{KCONFIG} . However, presence conditions of blocks need not simply be the expression in its header; parents and `#ifdef` chains must be taken into account, as described in [Sin+10]. When `#ifdef` blocks are nested, the presence condition of a block is the conjunction of its header condition with those of all of its ancestor blocks. When blocks are chained, like the one in lines 2–6 in Listing 2.4, their presence condition is the conjunction of its header condition with the negations of its predecessors' conditions.

```
1 #ifdef CONFIG_A
2 #   if defined(CONFIG_B) && defined(CONFIG_C_MODULE)
3 #   elif defined(CONFIG_D)
4 #   else
5       a
6 #   endif
7 #endif
```

Listing 2.4 – When processed with the CPP, the presence of the character `a` in the output would depend on the parent block `#ifdef CONFIG_A` to be present and the first two chain blocks to *not* be present. Writing bits (a_0, a_1) for feature A, and so on, the presence condition of the `a` is $a_0 \wedge \neg(b_0 \wedge c_1) \wedge \neg d_0$.

Finally, the CPP allows symbols to be redefined and undefined anywhere in the code. The formula can reflect this by including a logical implication from the presence of a block that contains

a redefinition to the new value of the respective symbol. Although Sincero et al. noted in their publication [Sin+10] that their implementation did not yet support this, UNDERTAKER has been thus extended since then.

2.1.3.3 Problems with the CPP

The CPP might seem like a natural choice for the task at hand, given that most of Linux is written in C, but its deficiencies have been pointed out in the literature. Spencer [Spe92] describes that it often acts as a poor substitute for well-designed interfaces by making small bits of code conditional all over the place instead of deferring the variability to a central module. Lohmann et al. concern themselves in [Loh+06] especially with the use of the CPP in system software and propose a remedy to its inherent problems, which they call “*#ifdef-hell*”, in the form of aspect-oriented programming. Defects partly result from the unsuitability of the CPP for implementing variability.

First, CPP-annotated code does not get parsed by the CPP, only tokenized. For example, the CPP would read `a+b` as `a`, `+`, and `b`, whereas `"a+b"` would be read as a single string literal token `a+b`, ensuring a rudimentary syntactic correctness. However, Listing 2.5 shows that a code file which is compilable with the correct configuration of symbols `A` and `B` (see Table 2.2) can still very well elicit a syntax error under a different configuration. Concentrating on only one of the four variations of the code file, the developer may mistakenly think his program valid and tested because the other three, like the one where the semicolon in line 8 is missing, are never even sent to the compiler. Even if the syntax of a variation is correct, the function call in line 6 might be compiled while the definition in line two is not. Assuming definition and call to belong to a single feature, this demonstrates that feature-related annotations tend to be spread across the code and the conditions in lines 1 and 5 would have to be kept in sync.

```
1 #if defined(A) && !defined(B)
2 int foo(int a) { return a; }
3 #endif
4 int main(void) { return
5 #ifdef A
6     foo(1);
7 #elif defined(B)
8     foo(2)
9 #endif
10 }
```

Listing 2.5 – CPP-annotated C code resulting in variations illustrated in Table 2.2.

Is there a simple solution that only involves changing one’s coding style? Code in regular `if` statements does cause compiler errors when it calls undefined functions or has an invalid syntax. On the other hand, regular `ifs` cannot surround function declarations, nor can they eliminate the spread of features and so cannot be a general replacement for `#ifdefs`. One would either need to improve the CPP itself, or make tools that support developers working with it.

| A defined? | B defined? | Active block in chain | foo declared? | Compiler / Linker errors |
|------------|------------|-----------------------|---------------|--------------------------|
| no | no | none | no | syntax error |
| no | yes | second | no | syntax error |
| yes | no | first | yes | none |
| yes | yes | first | no | foo undefined |

Table 2.2 – The different variations of Listing 2.5 depending on the configuration of A and B.

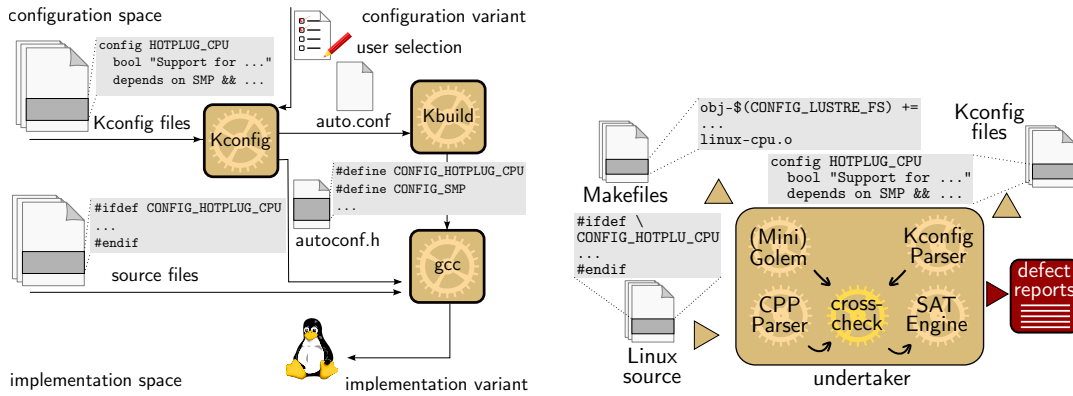
2.2 UNDERTAKER toolchain

UNDERTAKER’s dead code analysis does not type check or help test CPP-annotated code, but it eases the burden on developers of writing annotations by detecting redundant `#ifdef` conditions. Beginning with the presence condition of an `#ifdef` block ϕ_{CPP} , UNDERTAKER extends this formula step by step with ϕ_{KBUILD} and ϕ_{KCONFIG} until

$$\phi = \phi_{\text{CPP}} \wedge \phi_{\text{KBUILD}} \wedge \phi_{\text{KCONFIG}}$$

is reached (see Figure 2.2).

At each step, UNDERTAKER checks whether the current formula is satisfiable. If it is not, i.e., if it is a contradiction, the `#ifdef` block under scrutiny is deemed *dead* because it will never be compiled. On the other hand, if the negation of the formula is not solvable, then the formula is a tautology, and the block is *undead* (to be understood as the contrary of *dead*) because it will always be included in the build product whenever its parent is. In practice, not the entire formulas ϕ_{KCONFIG} and ϕ_{KBUILD} are used, but only the parts of them dealing with those features that actually appear in the block’s presence condition. This slicing algorithm was presented by Sincero in [Sin13].



(a) The Linux build system. Note how the configuration variant is communicated via the files `autoconf.h` and `auto.conf` to KBUILD and the CPP. Slightly modified version of Figure 1 from [Tar+11].

(b) UNDERTAKER’s mode of operation. Modified version of Figure 4 from [Tar+11].

Figure 2.2 – Summary of the Linux build system and how it is examined by UNDERTAKER.

2.2.1 UNDERTAKER-CHECKPATCH

Tartler et al. pointed out that UNDERTAKER could be useful when used as close as possible to a developer: “More importantly, we also aim at supporting programmers at development time [with undertaker] when only a few files are of interest. [...] we consider the efficient check for variability consistency during incremental builds essential.” [Tar+11]. UNDERTAKER-CHECKPATCH, implemented by Valentin Rothberg in [Rot14], was a step towards this goal. It runs UNDERTAKER’s static dead code analysis before and after applying a set of changes, called a *patch*, to the Linux tree, determines the differences between the analyses, and outputs a report as well as a supplementary `.analysis` file for each new defect.

The environment in which UNDERTAKER-CHECKPATCH is now mainly used is not, however, on a developer’s machine to check “incremental builds” but to check already published GIT commits in development repositories. The CADOS team chose to periodically run UNDERTAKER-CHECKPATCH on their own computers, evaluate and manually compile explanations and fixes for found defects, and send them off as e-mails to the responsible programmers. The target of this experiment, which evolved from the one described in [Tar+11], is the `linux-next`³ GIT repository.

2.2.2 Defect classes

Whenever UNDERTAKER finds a defect, it assigns it one flag from each of the three categories in Table 2.3. The distinction between *dead* and *undead* is the first and most basic form of classification. Second comes the part of the formula where the analysis stopped, which is very important to guide further analysis of a defect [Rup15; Die+12], which will be explored in the following sections. The third category is based on the architectures (x86, arm, etc.) a defect appears in. UNDERTAKER must test an `#ifdef` block for each architecture separately because each has its own main `Kconfig` file, perhaps producing differing results. Once the flags are assigned, UNDERTAKER writes the position of the defective block and the formula it used into a file whose name ends with the flags joined together with dots, e.g., `foo.c.B1.code.locally.dead`, where B1 is an identifier UNDERTAKER assigns to `#ifdef` blocks. UNDERTAKER-CHECKPATCH’s supplementary analysis would get written to `foo.c.B1.code.locally.dead.analysis`.

2.2.2.1 code defects

The first step of SAT checking examines ϕ_{CPP} alone, without ϕ_{KCONFIG} and ϕ_{KBUILD} , and this is where *code* defects can be detected [Rup15; Die+12]. This class is named as such because the formulas of defects belonging to it are unsatisfiable even when excluding `KCONFIG` and `KBUILD`. Its defects result solely from the presence condition of an `#ifdef` block. A very simple example might be a block whose header condition is `CONFIG_FOO && !CONFIG_FOO` – an obviously unsolvable formula. The

³The `linux-next` repository at <http://git.kernel.org/cgit/linux/kernel/git/next/linux-next.git> collects patches from a variety of development trees that are not yet in mainline Linux.

| Category | Flags | Description | UNDERTAKER-CHECKPATCH analysis |
|----------------|-------------------|--|---------------------------------|
| Basic | <i>dead</i> | Contradiction | |
| | <i>undead</i> | Tautology | |
| Part of ϕ | <i>code</i> | ϕ_{CPP} | Presence condition |
| | <i>kconfig</i> | $\phi_{\text{CPP}} \wedge \phi_{\text{KCONFIG}}$ | MUS and <i>always on/off</i> |
| | <i>kbuild</i> | $\phi_{\text{CPP}} \wedge \phi_{\text{KCONFIG}} \wedge \phi_{\text{KBUILD}}$ | MUS and <i>always on/off</i> |
| | <i>missing</i> | $\phi \wedge \bigwedge_{m_i} \neg \pi_0(m_i) \wedge \neg \pi_1(m_i)$ | Missing symbol and similar ones |
| | <i>no_kconfig</i> | ϕ contains symbols not beginning with CONFIG_ | Ignored |
| Architecture | <i>locally</i> | Less than all architectures | |
| | <i>globally</i> | All architectures | |

Table 2.3 – UNDERTAKER defect classes. The m_i are CPP symbols missing from KCONFIG, see Section 2.2.2.4. For a more detailed version of the two rightmost columns, see Sections 2.2.2.1 to 2.2.2.5. MUS stands for Minimal Unsatisfiable Subset.

very same formula ϕ_{CPP} , but in a more implicit way, would result for the inner block in Listing 2.6. UNDERTAKER-CHECKPATCH’s analysis file consists simply of the formula ϕ_{CPP} in this case.

```

1 #ifdef CONFIG_FOO
2 #   ifndef CONFIG_FOO
3 #       endif
4 #endif

```

Listing 2.6 – The outer block is not dead, but the inner one is dead, in combination with the outer one. `#ifndef` stands for “if not defined.”

2.2.2.2 *kconfig* defects

If no *code* defect could be found, UNDERTAKER adds ϕ_{KCONFIG} to the initial formula ϕ_{CPP} and checks the resulting formula $\phi_{\text{CPP}} \wedge \phi_{\text{KCONFIG}}$ [Rup15; Die+12]. If there is a defect now, it is flagged with *kconfig* as it arises no sooner than one connects a block’s presence condition with the KCONFIG model. Section 2.1.1.2 has shown an example for this class of defects. Consider boolean features F00 and BAR where the former depends on the latter. If the condition `CONFIG_F00 && !CONFIG_BAR` would now result from any combination of `#ifdef` chains and nestings, it would be unsolvable. Note that while ϕ_{CPP} is absolutely readable, $\phi_{\text{CPP}} \wedge \phi_{\text{KCONFIG}}$ commonly has lengths in excess of 500 lines. To overcome the limited usefulness of such an amount of text, UNDERTAKER-CHECKPATCH can generate a Minimal Unsatisfiable Subset (MUS) out of it and place it in its analysis. UNDERTAKER-CHECKPATCH also checks if KCONFIG features are involved that are *always on* or *always off* in every possible configuration variant. For instance, if F00 was *always on*, an `#ifdef` block testing for the definedness of `CONFIG_F00` would be *undead*, and UNDERTAKER-CHECKPATCH’s analysis would inform about this relationship.

2.2.2.3 *kbuild* defects

After $\phi_{\text{CPP}} \wedge \phi_{\text{KCONFIG}}$, UNDERTAKER finally tests the satisfiability of the full formula ϕ . If the test turns out negative, the defect gets assigned the class *kbuild*, introduced by Andreas Ruprecht [Rup15]. The presence condition of a defective `#ifdef` block of this kind is not contradictory itself, nor in combination with KCONFIG, only when taking the build system into account.

For example, assuming again that F00 and BAR are boolean features with F00 depending on BAR. A block `#ifndef CONFIG_BAR` would then be *kbuild dead* if it appeared in a file `foo.c` which is referenced in its `Makefile` with `obj-$(CONFIG_F00)+= foo.o`. Under these circumstances, the block would only be compiled if both F00 and BAR are selected, demonstrating how a dependency chain through all three formulas can give rise to a contradiction.

As to the UNDERTAKER-CHECKPATCH analysis, the entire formula ϕ can obviously only be more complicated than its extension $\phi_{\text{CPP}} \wedge \phi_{\text{KCONFIG}}$, so a MUS is generated here as well.

2.2.2.4 *missing* defects

If, even by use of the formula ϕ , no defect is found, UNDERTAKER gathers all KCONFIG symbols m_1, \dots, m_n in use in the formula and sets their propositional literals in the SAT checker to `False` [Rup15]. If the expression $\phi \wedge \bigwedge_{m_i} \neg \pi_0(m_i) \wedge \neg \pi_1(m_i)$, is then a contradiction or a tautology, the defect is of class *missing* as ϕ tests symbols that cannot possibly be defined in `autoconf.h`, i.e., are always missing from it. These defects can result from typos in `#ifdef` conditions, in build scripts, or in KCONFIG files [Rot14]; or from the use of symbols that do not exist in mainline Linux. A programmer might have a private development tree with its own KCONFIG features for testing purposes. If this developer submits a patch, it might still contain references to those features.

missing defects being the most common ones [Rot14], they are also the most easy ones to analyze. UNDERTAKER-CHECKPATCH outputs the names of missing symbols and a list of symbols that do exist and are similar in name to the missing ones, which helps when typos are involved.

2.2.2.5 *no_kconfig* defects

Rothberg introduced one last class, *no kconfig*, in [Rot14]. Whenever a defect is found, it gets assigned to this class, regardless of its previous classification, under the condition that the block's presence condition ϕ_{CPP} must reference symbols not beginning with `CONFIG_`. This is to catch defects resulting from blocks with conditions like `#ifdef DEBUG` which are most likely intended to be dead except during development.

2.3 Related work

Attempts to visualize one's way out of the `#ifdef` hell [Loh+06] have been made. The two most closely related to WUNDERTAKER seem to be FEATURECOMMANDER [Fei+11] and the *Colored IDE* (CIDE) [KTA08], a prototype of a stand-alone GUI and an Eclipse plug-in, respectively. They both

feature a folder view, giving an overview over the files in a project, and a code view which shows code in a way that is beneficial to understanding annotations around it. FEATURECOMMANDER does visualize actual `#ifdefs`; CIDE does not, but rather its own, special, way of annotating code. Both focus on showing *features* in connection with a feature model, not individual `#ifdef` blocks.

Techniques employed by them that are also interesting for WUNDERTAKER are hence those that are limited to within a single code file because `#ifdef` blocks, unlike features, do not span across files. First, CIDE has an elaborate functionality to hide code, helping focus the attention on code fragments that are more relevant than others. This technique is not unique to CIDE and is sometimes called ‘code folding’; Integrated Development Environments (IDEs), like Eclipse⁴, use it to hide bodies of function definitions, of classes, or of `#ifdef` blocks, etc.

Secondly, because nested `#ifdef` blocks can be seen as a tree, FEATURECOMMANDER places a bar next to the code of a block. The bars of children, that necessarily run within the line range of a parent, are stacked onto parent bars. [CPR07] presents another tool that uses stacked bars to make it clear at a glance where a block of code that is found in a tree begins and ends.

The most obvious visualization technique that both FEATURECOMMANDER and CIDE employ is probably the use of background colors to highlight annotated code. As noted in [KTA08], if colors are supposed to uniquely identify more than a handful of features, then they cannot scale well in large projects with many of them. Colors in CIDE are rather used for their contrast – to emphasize the boundaries of code fragments. FEATURECOMMANDER takes this notion one step further and does not assign any saturated colors to features unless the developer does it manually for a small subset.

Some of the authors of [Fei+11] and [KTA08], among others, have conducted a very comprehensive set of studies on the effect of using background colors to highlight `#ifdef` blocks [Fei+13]. In the first experiment, they tested two groups: the first had an uncolored and `#ifdef` annotated code view, and the second had a colored but unannotated one. Subjects of both groups were given a set of tasks from two categories: static tasks and maintenance tasks. Static tasks required subjects to associate code with features; in maintenance tasks, they had to fix bugs starting from bug reports.

The conclusion from the experiments is that colors are beneficial. Users from both groups estimated that working with the colored version is faster than working with the uncolored one. Users of the colored version were indeed faster with static tasks and took a comparable time for all maintenance tasks except one. The authors assumed that the color group was slower with this one task because a red color was used in it to highlight large swaths of code, probably negatively affecting performance. Whereas in this experiment users did complain about the badly chosen color, they did not in a second experiment with a single group that had the possibility to switch between `#ifdefs` and colors. Most subjects chose to use colors and were consequently slowed down in a task that involved red again by the same amount as the color group in the first experiment, but without realizing it. Finally, the third experiment confirmed that previous results also apply in large-scale software projects. However, it involved one task where users of the colored code had to work with 12 different colors at the same time, also resulting in a performance worse than that of the `#ifdef` users.

⁴http://www.eclipse.org/pdt/help/html/using_code_folding.htm

WUNDERTAKER

3.1 Environment

WUNDERTAKER is the front end of UNDERTAKER-CHECKPATCH's dead code analysis, visualizing the defect reports and fitting in with the rest of the toolchain, and its first and foremost use case is supposed to be as a didactic tool in the `linux-next` experiment (see Section 2.2.1). The following sections will answer how, where, and by whom WUNDERTAKER is intended to be run.

3.1.1 WUNDERTAKER as an UNDERTAKER GUI

Computations of the `linux-next` experiment are conducted centrally. The CADOS team analyze the generated reports and send out e-mails as sort of a service. If they had a GUI at their disposal, they could, after having checked a `linux-next` commit, update it with the respective reports and refer to it from within their e-mails.

In order to be up-to-date, WUNDERTAKER would therefore firstly need access to the CADOS server where the reports lie and secondly to the respective Linux code where the defects appear in, which can be any, up to the most recent, `linux-next` commit. Although an implementation as a desktop GUI is thinkable, I chose to keep both things in one place – on the CADOS server, with the reports being in a file system and the code in a GIT repository. WUNDERTAKER would then run as a web application, managed by the CADOS team and serving Hypertext Markup Language (HTML) pages from this server to Linux developers who received report mails.

Benefits of this approach are that WUNDERTAKER is automatically made platform-independent without any further effort, given the universality of web standards, and that it can be integrated into e-mails almost seamlessly by a link, being accessible via the Internet to the same degree the e-mails are as well. This last point is probably the most important one; if WUNDERTAKER needed set-up time on part of the recipients, its acceptance would likely suffer. *Ideally*, the visualization behind the link would be comprehensive to the point of making any manually written explanation of the defect redundant, so e-mail writers' lives would be made easier. We will see that this goal is only partially reachable within the scope of this work, but WUNDERTAKER should certainly be able to both relieve the writers of repetitive tasks and to improve and supplement the explanations.

3.1.2 WUNDERTAKER as a GIT repository view

As we found out, WUNDERTAKER already needs access to a GIT repository. It is then only a small step towards making the repository fully browsable via a folder view that presents clickable subfiles and subfolders, at least when limited to a given GIT commit. Navigation from one commit to another and along GIT branches is, however, more complex, and one needs to draw a line between what is and is not sensible to include as a functionality. WUNDERTAKER is not supposed to be a development platform like GITLAB⁵, for example, but a fully read-only online view of a single repository. Navigation features should be provided only as long as they are conducive to conveying a context to UNDERTAKER defect reports. If WUNDERTAKER not only visualizes files but also the folders that contain them up to the root of the tree of a given commit in a rudimentary way, like in a common file browser (see Figure 3.1), it should be sufficient. As one further enhancement, WUNDERTAKER could count the reports that exist beneath each item in the view and show this number, giving an impression of how defects are distributed.

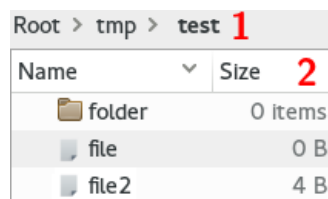
3.2 The didactics of defect reports

After UNDERTAKER-CHECKPATCH has found a defect, it supplies three bits of information: the formula, a part of ϕ ; the position of the defective block; and the .analysis file. In the following, we go through the different classes of defects and find suitable ways to explain them in a graphical form. A thorough examination of the general causes and fixes of variability defects without regard to GUIs has been conducted by Nadi et al. in [Nad+13], and by Rothberg in [Rot14].

3.2.1 missing

Defects of this class are easy to understand, given that the UNDERTAKER-CHECKPATCH analysis already names the symbol missing from KCONFIG (see Listing 3.7). It should not be hard for a developer who is accustomed to the defective code to grasp the problem, then. When an e-mail links to

⁵<https://about.gitlab.com/>



| Root > tmp > test 1 | |
|----------------------------|---------------|
| Name | Size 2 |
| folder | 0 items |
| file | 0 B |
| file2 | 4 B |

Figure 3.1 – A screenshot of the dolphin file browser showing the contents of a folder containing both folders and files. Things to note are the clickable path (1), the separation between folders and files, and the columns that supply further information on each item (2). WUNDERTAKER should try to stick close to this, as it is a tried approach, making WUNDERTAKER consistent with programs already known to users.

WUNDERTAKER, the latter should therefore show this exact analysis in a view of the relevant code file right next to the CPP annotations that caused the defect. As a result, the e-mail would not need to mention the defect class, the position of the `#ifdef` block, nor what symbol is missing, freeing the CADOS team from manually assembling this information over and over again whenever a *missing* defect arises.

```
New defect: arch/arm64/kernel/cpufeature.c:B1:66:74:missing.globally.dead
CONFIG_AS_LSE is referenced but not defined in Kconfig

Similar symbols: CONFIG_AFS_FS, CONFIG_USB_LED, CONFIG_AFFS_FS
```

Listing 3.7 – Analysis of a *missing* defect that UNDERTAKER-CHECKPATCH found in Linux v4.3.

3.2.2 code

A good visualization of *code* defects is perhaps best found when looking at one. GIT commit 73b341efd changed occurrences of `CONFIG_PPC_HAS_HASH_64K` to `CONFIG_PPC_64K_PAGES`, among the affected `#ifdef` blocks the one starting in line 704 in `arch/powerpc/mm/hash_low_64.S` (see Listing 3.8). The block happened to have already contained another block testing for the exact same symbol. As a consequence, the inner block is classified as *undead* and its `#else` branch as *dead*.

```
#ifdef CONFIG_PPC_64K_PAGES
[144 lines omitted]
#ifdef CONFIG_PPC_64K_PAGES
    oris r30,r30,_PAGE_HPTE_SUB0@h
#else
    ori r30,r30,_PAGE_HASHPTE
#endif
[142 lines omitted]
#endif /* CONFIG_PPC_64K_PAGES */
```

Listing 3.8 – Snippet from `arch/powerpc/mm/hash_low_64.S` from v4.3.

UNDERTAKER-CHECKPATCH’s analysis, the presence condition of the block, is shown in Listing 3.9. The formula is short but not self-explanatory for developers that are unfamiliar with UNDERTAKER. Firstly, block identifiers B00, B0, etc. are unique to UNDERTAKER, and not used by other programs, and take some time to get familiar with – something WUNDERTAKER can avoid entirely by not employing any textual notation at all but rather showing the blocks themselves and placing interactive shortcuts between them. So WUNDERTAKER knows where the shortcuts must point, it can run UNDERTAKER with options that make it print out the line numbers of the blocks. WUNDERTAKER can thus build on existing, but disconnected, bits of information and bring them together.

```
New defect: arch/powerpc/mm/hash_low_64.S:B5:849:851:code.globally.undead
Tautology in the block's precondition:
B5
&& ( B5 <-> B4 && (CONFIG_PPC_64K_PAGES) )
```

```

&& ( B4 <-> (CONFIG_PPC_64K_PAGES) )
&& B00

```

Listing 3.9 – UNDERTAKER-CHECKPATCH analysis of the *code* defect.

Secondly, the formula can become quite convoluted in cases such as in Listing 3.10. It deals with block 33 that is located within block 32 which is in turn located in block 31, which itself has an include guard as its parent, as indicated in line 4. The relevant bits are in lines 2 and 8: block 32 tests `STRICT_MM_TYPECHECKS` while block 31 `#undefs` it. So block 32 and hence block 33 cannot be enabled; the latter is marked as *dead*. This is hard to see at first glance from the formula. Therefore, an UNDERTAKER GUI should not depend on formulas if it does not need to. Rather, it should visualize the nesting of `#ifdef` blocks. Such a hierarchy is already easier to see if code not belonging to CPP directives is hidden in the same way I manually did in Listing 3.8. There, the inner block is located more than 140 lines – multiple pages – into its parent from both ends.

Also, if a block references a symbol which is redefined as constant beforehand, WUNDERTAKER should make this noticeable. In conclusion, *code* defects are the ones that might benefit the most from a GUI. Presence conditions of blocks are rather complicated to explain so a structured and automated way to show them would surely ease the writing of e-mails.

```

1 ( B33 <-> B32 && ((CONFIG_PPC_64K_PAGES) && (CONFIG_PPC_STD_MMU_64)) )
2 && ( B32 <-> B31 && (STRICT_MM_TYPECHECKS.) )
3 && ( B31 <-> B0 && (! __ASSEMBLY__) )
4 && ( B0 <-> (! _ASM_POWERPC_PAGE_H) )
5
6 && (B0 -> _ASM_POWERPC_PAGE_H.)
7 && (!B0 -> (_ASM_POWERPC_PAGE_H <-> _ASM_POWERPC_PAGE_H.))
8 && (B31 -> !STRICT_MM_TYPECHECKS.)
9 && (!B31 -> (STRICT_MM_TYPECHECKS <-> STRICT_MM_TYPECHECKS.))

```

Listing 3.10 – Presence condition of block 33 in `arch/powerpc/include/asm/page.h` of Linux commit 456fdb267.

3.2.3 *kconfig* and *kbuild*

Let us again begin with an example to find further requirements for WUNDERTAKER: in commit 4c477de14237, there is an *undead* block in the file `kernel/stop_machine.c`. The commit gave an `#ifdef` block the condition `CONFIG_SMP || CONFIG_HOTPLUG_CPU`. But it becomes apparent that this condition is none, really, since `stop_machine.c` will not ever be compiled without `CONFIG_SMP`, as defined in its `Makefile`. This is exactly what UNDERTAKER-CHECKPATCH says in its analysis shown in Listing 3.11. Again, a GUI should display this alongside the relevant code.

Listing 3.11 – UNDERTAKER-CHECKPATCH analysis of a *kbuild* defect.

```

File preconditions from build system create a tautology

```

```
File precondition for architectures ['alpha', 'arc', 'arm', [...]]: CONFIG_SMP
[...]
```

Given the complexity of the formulas involved in these classes, WUNDERTAKER is probably least useful here, mostly relying on UNDERTAKER-CHECKPATCH's MUS form of the propositional formula ϕ . $kconfig$ and $kbuild$ defects, by definition, span across multiple files, and WUNDERTAKER only visualizes a single one at a time – the C code file. Still, it is not a given that $\phi_{KCONFIG}$ or ϕ_{KBUILD} are complicated. Maybe only ϕ_{CXX} is – one of a block's many parents references a symbol that is already used in a Makefile, like SMP above. And, like above, the content of the .analysis file would suffice to see the problem. Also, showing the #ifdef hierarchy in a view resembling an IDE or text editor a developer is used to, could very well help to construct a mental context to the defect.

3.3 Requirements

From the preceding sections, one can now gather a list of basic requirements WUNDERTAKER should meet. WUNDERTAKER's code view, on the one hand, should (1) be visually similar to those found in IDEs, including syntax highlighting; (2) display the .analysis provided by UNDERTAKER-CHECKPATCH in the vicinity of the defective #ifdef block it belongs to; (3) be able to hide, or fold, any code not belonging to a feature annotation; (4) visualize the hierarchy of #ifdefs, i.e., the way they are nested; (5) have clickable shortcuts from #ifdef headers to #define directives if they share symbols; and (6) should *ideally* show the presence condition formulas it gets from UNDERTAKER only optionally. Its folder view, on the other hand, should (1) have a clickable breadcrumb path, (2) adhere to common file browser GUI design principles, and (3) show the number of reports that are in each item.

Being only a front end, WUNDERTAKER should re-use as much functionality from the UNDERTAKER toolchain as possible, such as the parsing of #ifdef conditions.

3.4 Implementation

3.4.1 Visualization techniques

3.4.1.1 The code view

Suppose one wants an all set-up and running WUNDERTAKER to display its visualization of the file `arch/powerpc/mm/hash_low_64.S` in the tree of Linux `v4.3`. Being a web application, the intended way to do this would be to send a request with a web browser, where the first component of the path is the version and the rest is the path of the file. So if this WUNDERTAKER instance was running on `www.example.com`, then the address bar of the browser would show something like `http://www.example.com/v4.3/arch/powerpc/mm/hash_low_64.S`. In fact, this path is the only information WUNDERTAKER only ever needs from a user; it does not store any cookies or

server-side session data, except for caching purposes. WUNDERTAKER then looks up the path in its Linux GIT repository, finds that it is indeed a code file, not a folder, generates its output, including information from relevant UNDERTAKER defect reports it reads from a file system, and returns a response to the browser.

After this response has been rendered, the user will see a page with the general layout depicted in Figure 3.2. The page header and footer are purely decorative elements. The defects list acts as sort of a table of contents linking from short descriptions of defects to defective blocks inside the code view. This, where the substance of the visualization can be found, is shown in Figure 3.3. The figure shows the visualization of the entire file; no parts have been cut from the screenshot. The reason for why it all fits on one page is that WUNDERTAKER shows only lines of code that are responsible for beginning or ending an `#ifdef` block. A number of lines are additionally shown after each `#ifdef` header in order to allow for a certain context when glancing over the view. Certain points of interest in the figure are numbered for further explanation.

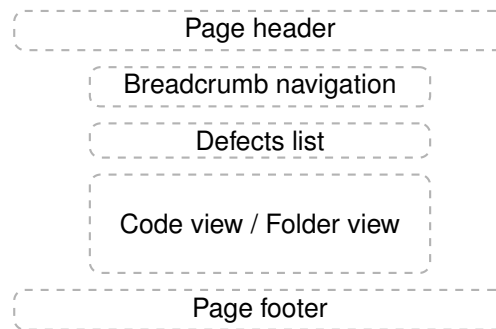


Figure 3.2 – General layout of a WUNDERTAKER page.

There is an ellipsis, numbered with 1, which stands for 291 lines of code that are not related to any `#ifdef` annotations. This is code I call *ancillary* code for it just happens to exist between code that is actually analyzed by the UNDERTAKER toolchain. Each ellipsis stands for a fragment of ancillary code, and is not just a static placeholder but can be interacted with by clicking on it, upon which the code underneath it will be revealed and the placeholder hidden in its stead – something that can also be done en masse by using the button above number 3, which expands all fragments at once. The ellipses implement the code folding technique that should augment the focus on feature annotations by hiding irrelevant code by default.

The left column, numbered with 2, contains boxes, one for each `#ifdef` block, with their top borders aligning with those of the `#ifdef` blocks in the code. The first line in a box shows the identifier of the `#ifdef` block it describes, numbered from zero onwards the same way UNDERTAKER does it. The rest is the presence condition of the block in which WUNDERTAKER recognizes block identifiers and links to the headers of the respective blocks. For example, if a user clicks on a B3 link in the example, the page would be scrolled in order to show line 459 if it was not already visible. Since the boxes may overlap, or not fit into the column, they are brought to the front when the user

The figure displays a code editor interface for the file `arch/powerpc/mm/hash_low_64.S` in Linux v4.3. The interface is divided into four main sections:

- Block Conditions (2):** A list of conditions defined in the code, such as `B0` (`CONFIG_PPC_64K_PAGES`), `B1`, `B2`, `B3`, and `B4`. Each condition is associated with a specific code block.
- Block Boundaries (4):** A column showing vertical bars that represent the boundaries of code blocks. The bars are color-coded to match the background of the corresponding code block.
- Code View (6):** The main area showing the assembly code with syntax highlighting. The code includes conditional compilation directives like `#ifdef` and `#endif`, and assembly instructions like `andc`, `andi`, and `rlwinm`.
- Defect View (5):** A section showing a new defect: `arch/powerpc/mm/hash_low_64.S:B6:851:853:code.globally.dead`. The defect description indicates a contradiction in the block's precondition, specifically involving conditions `B6`, `B5`, `B4`, and `B0`.

Figure 3.3 – Code view of `arch/powerpc/mm/hash_low_64.S` in Linux v4.3.

hovers over them, as shown with the box of block B0. Fulfilling another one of the requirements, the column can be removed from view with the button below 3.

The middle one of the buttons (3) is active in this example, meaning that `#ifdef` coloring is enabled in the figure. Indeed, the blocks are colored with one of five colors, where red is reserved for defective blocks. Furthermore, in the code view, the text itself is colored, not just the background. This is standard syntax highlighting as it can be found in IDEs or text editors. I will describe the exact choice of colors for both foreground and background in Section 3.4.2.

Next is the “Block Boundaries” column (4) whose purpose is to visualize the *hierarchy* of `#ifdef` blocks. Each bar in this column corresponds to a block and runs next to its lines in the code column with the exact same height as the block has there, growing and shrinking whenever ellipses are expanded or closed. Bars belonging to child blocks stack to the right onto the bars of their parents so that bars always have a smaller height than the bar on their left. The color of a bar is similar to the background color of the respective block; it is more saturated but has the same hue so as to make a connection to the code view. The bars are also clickable; each leads to the header of the block it belongs to, like when clicking on a block identifier.

The last point in this figure is not one of the requirements per se, but helpful for the user. When hovering above the column headers, like the one numbered with 6, a small help box is shown describing some of the details from the text above.

Please note that, in this figure, `#ifdef` highlighting is not switched on. As a result, block bars are here shown in alternating shades of gray rather than saturated colors. Merely the defective blocks are still in red. Also, block headers have a special, yellow, background color of their own which the lines surrounding them do not.

3.4.1.2 The folder view

```

279 #ifdef __ASSEMBLY__
280
281 #undef STRICT_MM_TYPECHECKS
282
283 #ifdef STRICT_MM_TYPECHECKS
284 /* These are used to make sure we are not checking. */
285
...

```

The following block (B33) is dead (code, globally).

```

294 #if defined(CONFIG_PPC_64K_PAGES) && defined(CONFIG_PPC_STD_MMU_64)
295 typedef struct { pte_t pte; unsigned long hidx; } real_pte_t;

```

The following block (B34) is dead (code, globally).

```

296 #else
297 typedef struct { pte_t pte; } real_pte_t;
298 #endif

```

Figure 3.4 – Code view of `arch/powerpc/include/asm/page.h` of Linux commit 456fdb267.

one made up from numbers 1 and 2 in Figure 3.5. Each of the path components in 1 is a link and so Figure 3.5 would be the result of the user clicking on `hash_low_64.S`'s parent.

WUNDERTAKER, upon receiving a request for the path `/v4.3/arch/powerpc/mm`, would recognize it as a folder and generate an appropriate view (3 in Figure 3.5). This view is a simple list of folders and files residing within the requested directory, together with a display of the number of defect reports found in each item.

Pairs of a version and a path, as used by WUNDERTAKER, can be converted to equivalent Uniform Resource Locators (URLs) naming the same file or folder in the GIT repository browser used at `www.kernel.org`⁶. WUNDERTAKER can, for this reason, generate a link, numbered with 2 in Figure 3.5, to the `linux-next` repository on `kernel.org`. As mentioned in Section 2.2.1, `linux-next` is the target of periodic invocations of `UNDERTAKER-CHECKPATCH` by the CADOS team. The repository browser at `www.kernel.org` has much more advanced features for navigating within the GIT history of a file and can thus complement WUNDERTAKER because, being a web application as well, it should be accessible from anywhere where WUNDERTAKER is also available.

3.4.2 Colors

In WUNDERTAKER's code view, there are several elements that benefit from coloring. First of all are the `#ifdef` blocks. Being highlighted with saturated background colors, as opposed to the color white, notifies a user reading their code that it is annotated. By contrasting with colors of neighboring blocks, their boundaries become clear, too. Colors achieve this without the need of even reading the text, since they are processed by the human brain first [Fei+13]. Second are the block bars whose coloring would serve the same purpose. Last are tokens of the program code that need to be syntax highlighted. However, the colors of tokens must also contrast with the background colors of blocks so as to keep the code readable.

Drawing from the experience gathered by Feigenspan et al. in [Fei+13] that is summarized in Section 2.3, I tried to distribute these colors as well as possible regarding the criteria stated above.

⁶`kernel.org` is the web page where official Linux GIT repositories are hosted, including Linus Torvalds's mainline repository.

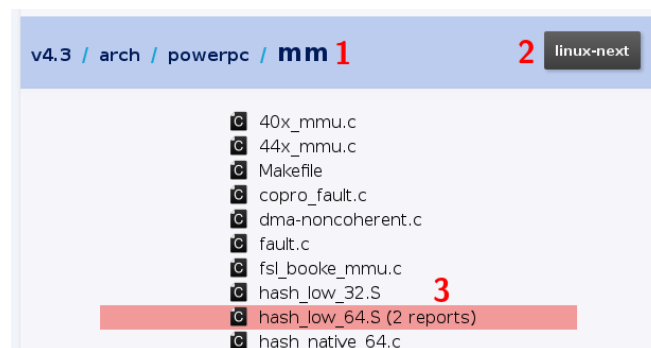
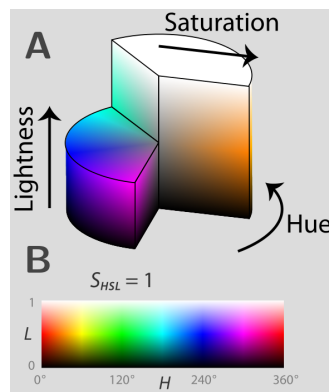


Figure 3.5 – The top of the view of folder `arch/powerpc/mm` in Linux v4.3.

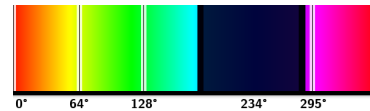
I used the three-dimensional Hue, Saturation, Lightness (HSL) coordinate system for describing colors (shown in Figure 3.6a). In short, a point in this system describes a color. Points with $L = 1$ are always white, regardless of the other parameters; points with $L = 0$ are black (see B in Figure 3.6a). Also shown in B is that the hue H is written as an angle from 0° to 360° . Fixing $L = 1/2$, points with $S = 0$ would appear as a gray exactly midway between black and white, and points with $S = 1$ and $L = 1/2$ would appear as the most saturated colors possible in HSL which are depicted on the imaginary horizontal line in the middle of Figure 3.6a B.

With this information, one can construct colors so that the intuitive contrast between them is maximized. Suppose S and L are constant and only H remains free. One then obviously gets the most contrast among n colors if the distances between their hues are maximized and all equal. When thus distributing, for instance, six values over the domain of H , one can start with 0° and then add 60° in successive steps until 0° is reached again, resulting in $0^\circ, 60^\circ, 120^\circ, 180^\circ, 240^\circ, 300^\circ$. However, when choosing foreground colors for text that is to appear upon a colored background, one must leave out a range around the background color's hue lest the text become unreadable. This extended process is illustrated in Figure 3.6b and can be seen as a function $\text{dist}(n, H_{\text{avoid}}, w)$ that results in n hues with maximum contrast, under the boundary condition of leaving out a range of width w around the supposed hue H_{avoid} .

As shown in Table 3.1, I choose to keep S and L constant among each of the three groups of colored elements, text, boundary bars, and block backgrounds. The reason for this is that variations of lightness or saturation among these elements would look misleading. Consider one block that has a green as a background color while the block following it is colored with a blue that is much paler. A user might then associate a valuation with the difference in saturation – that one block is somehow more or less important than another. Hues, except red, on the other hand, should be free of valuation with regard to `#ifdef` blocks.



(a) A cropped and slightly modified version of https://en.wikipedia.org/wiki/File:Hsl-hsv_models.svg by Jacob Rus under license CC BY-SA 3.0.



(b) Result of $\text{dist}(4, 234^\circ, 102^\circ)$. The prohibited range is overlaid with black. The hues found by dist are marked by white stripes.

Figure 3.6 – The HSL system.

| Visual Element | H | S | L |
|----------------|---|-----|------|
| Backgrounds | $0^\circ, 60^\circ, 120^\circ, 180^\circ, 240^\circ, 300^\circ$ | 0.5 | 0.85 |
| Bars | $0^\circ, 60^\circ, 120^\circ, 180^\circ, 240^\circ, 300^\circ$ | 0.6 | 0.5 |
| Text | $\text{dist}(8, H_{\text{Background}}, 90^\circ)$ | 1 | 0.25 |

Table 3.1 – Fixed S and L values and the distribution of hues in the different colored GUI elements. Saturation values become larger from row one to the last row while Lightness values become smaller in order to make text colors appear ‘stronger’ and darker than the light background. Bar colors are in between.

The six hues constructed above are used for the six background colors. The rationale for this number is that the graph coloring problem for a tree of `#ifdef` blocks, additionally assuming that siblings are connected, can be solved with four colors because of the Four Color Theorem [Wik15]. Therefore, four of the six hues are used for regular coloring: WUNDERTAKER alternates two sets of two colors between levels in the tree. So nodes at level 1 are colored with the first two colors, blocks at level 2 with the second two, and level 3 with the first set again, and so on. Within a level, the two colors in the set are alternated resulting in the desired property that each line that starts or ends a block will have another color than the line that precedes or follows it, respectively. The fifth color is red, which is reserved for defective blocks and not used as a regular block background respecting the insight from [Fei+13] that it would be a bad color for that. The sixth is an extra color used to highlight certain lines that have been selected by the user, for example the header of a block after being jumped to.

All in all, six background colors should be few enough to not overload the perception of users; users of the colored version in the experiments by Feigenspan et al. only started getting problems in tasks involving 12 colors. Also, this `#ifdef` coloring feature is optional and off by default. Rather than being permanent, it is intended as an option that is switched on by the user, once he has found a certain spot in a file, to discern between `#ifdef` blocks local to the file – to find their boundaries as quickly as possible, and to help the user decide what block a line of code belongs to.

3.4.3 Software that WUNDERTAKER uses

As a web application, WUNDERTAKER communicates with web browsers via the Hypertext Transfer Protocol (HTTP). An HTTP request is text-based and can be as simple as in Listing 3.12 which might be what a browser would send to `www.example.com` if one were to enter `www.example.com/foo` in its address bar. Of interest to WUNDERTAKER are only GET requests which, according to the HTTP standard, are intended for pure information retrieval. That is, WUNDERTAKER never changes any state in the wake of a request but loses all information about a connection the moment it is closed. The sole exception to this rule of read-only access and statelessness is the caching it does to enhance performance.

WUNDERTAKER does not handle all the protocol details itself but builds on a stack of software responsible for this. I chose Ruby⁷ as the language to write WUNDERTAKER in because it provides a

⁷<https://www.ruby-lang.org/en/>

useful combination of libraries and frameworks for this task, which is demonstrated in this section. How WUNDERTAKER interacts with these dependencies can perhaps best be explained by dissecting it into two dimensions: Vertically, it is placed on top of a stack of HTTP-related software where requests travel upwards and responses downwards. Horizontally, it gathers all the data it needs from several sources to produce HTML documents once it has got a request.

```
GET /foo HTTP/1.1
User-Agent: Mozilla/5.0
If-Modified-Since: Sun, 6 Dec 2015 12:00:00 GMT
```

Listing 3.12 – A HTTP request.

3.4.3.1 Vertical Design

Imagine a web browser sending a request, as sketched in Figure 3.7. The request would travel across the internet to the web server WUNDERTAKER is running on. WEBRICK⁸ is a simplistic web server integrated into the Ruby standard library and so makes it easy to interface to it from Ruby programs. But WUNDERTAKER can also run on the APACHE HTTP Server⁹ as long as the Phusion Passenger¹⁰ module is installed and loaded into APACHE. This is necessary because before passing the request on to the Ruby-based part of the stack, it needs to be encoded in a rudimentary way as a Ruby object, as defined by the Rack interface¹¹, and Passenger can do this.

If it is so configured, the RACK¹² program itself can now directly pass on the request to the second level. Usually however, it first wraps a certain set of middleware around the call, like `Rack::ShowException` that catches any Ruby exception occurring further up and generates an error page with traceback based on this. SINATRA¹³, upon getting the request, goes through a list of routing rules, both SINATRA's own and ones that have previously been registered by WUNDERTAKER, and calls a function associated with the first one that matches. For example, one of WUNDERTAKER's rules is defined as a match of the pattern `/*/*` against the path of the request. The path `/v4.3/kernel/sched/core.c` matches this pattern – a request for `kernel/sched/core.c` as it was when `v4.3` of Linux was released. The response then travels all the way back down to the web server, where it is encoded in HTTP, and sent back across the Internet.

3.4.3.2 Horizontal Design

Figure 3.8 illustrates WUNDERTAKER's mode of operation after a request has been passed on from SINATRA. In this diagram, arrows indicate that data are moved from one component to another. The vertical structure below WUNDERTAKER is hidden in this diagram (Figure 3.7 rotated by 90 degrees).

⁸<http://ruby-doc.org/stdlib-2.2.2/libdoc/webrick/rdoc/WEBrick.html>

⁹<https://httpd.apache.org/>

¹⁰<https://www.phusionpassenger.com/>

¹¹<http://www.rubydoc.info/github/rack/rack/master/file/SPEC>

¹²<https://rack.github.io/>

¹³<http://www.sinatrarb.com/>

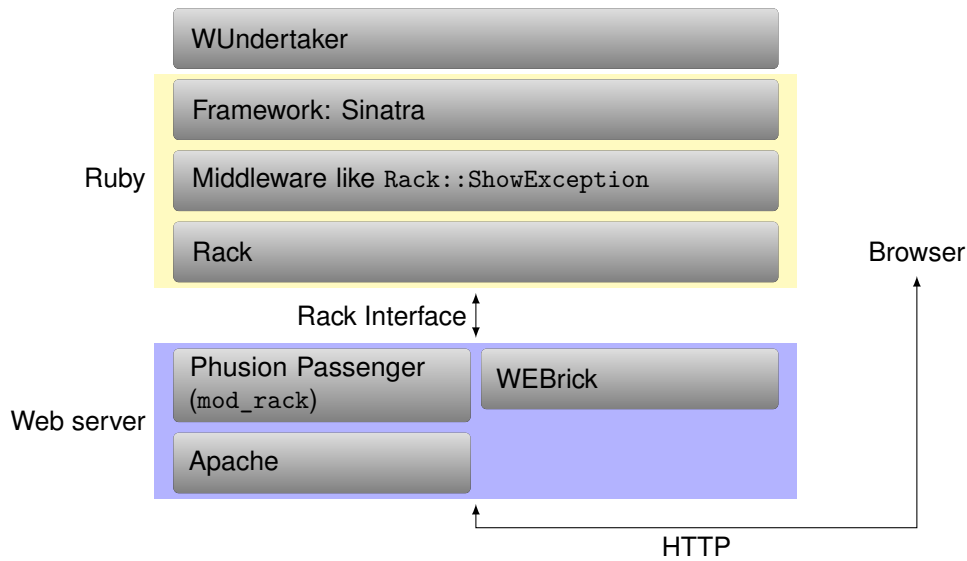


Figure 3.7 – The vertical design of WUNDERTAKER.

The first important thing that catches the eye is the file tree on top of the diagram. This is the folder WUNDERTAKER “runs on”; it contains the Linux GIT repository and another folder `commits`. `Makefile` and other files are checked out from the repository in the drawing, but it is not a requirement. It merely illustrates the fact that the repository is the sole place where the actual code of Linux needs to be stored.

Let us again go through the sequence of steps leading from an HTTP request to a response, indicated by the numbers inside circles. After the request for a file is received and routed at ❶, WUNDERTAKER fetches the specified version, indicated in red, of the file at the specified path, written in blue, from the repository ❷.

Next, a simultaneously running instance of MONGODB¹⁴ is consulted whether it already has defects reports and information about `#ifdef` annotations regarding this file in its storage ❸. This is to avoid the possibly costly step ❹ and also step ❺. Step ❹ involves executing UNDERTAKER as a separate process on the retrieved repository code, once to parse the position of `#ifdef` blocks in it, and once for each block to extract its presence condition formula. Step ❺ is more straightforward since it only comprises finding and reading UNDERTAKER defect reports in the `commits` folder. Notice how the red part of the request, the version, is matched against the names of the direct subfolders of the `commits` folder. This is done by normalizing it as the SHA1 checksum of the GIT commit it describes since, for the sake of convenience, it may be any uniquely identifying prefix of the actual checksum as well as a pointer or shortcut, called tags in GIT, to the commit. WUNDERTAKER then tries to find UNDERTAKER defect reports and their `.analysis` files in the matched subfolder under the path of the file.

¹⁴MONGODB is a document-oriented database management system, see <https://www.mongodb.org/>.

4

EVALUATION

Linux is a large software project, with the tree of v4.3 consisting of 3439 folders containing 51556 files, many of them with over 1000 lines of code. When counting defect reports, WUNDERTAKER must traverse parts of this tree. When visualizing files, it must tokenize the code for syntax highlighting, parse `#ifdef` directives, extract presence conditions, and finally construct an HTML document. Performance is therefore a major concern for WUNDERTAKER; having to wait lengthy periods of time just for one page to generate would severely diminish its usefulness and acceptance among developers, especially so when it would make the use case of “clicking around” in the repository unreasonably tedious. This section shows that WUNDERTAKER, while meeting its design requirements, is a usable program by measuring and presenting several numbers that arise when it operates, first and foremost being the speed and memory consumption of WUNDERTAKER.

4.1 Targets and Procedure

All measurements were done on a machine with an i5-3570K CPU with four cores, 16 GBs of RAM, and with WUNDERTAKER running on WEBRICK. The Linux GIT repository and all other files WUNDERTAKER needs to access, including the MONGODB database, were moved onto a `tmpfs`, which is a file system residing entirely in the RAM of a machine without requiring any disk to perform reads or writes¹⁵. This is to prevent delays caused by input / output (IO) from affecting the results too much because the delays are hard to predict. I also make sure during testing that enough RAM remains free so as not to risk the kernel swapping out any files from the `tmpfs`.

I invoke WUNDERTAKER by sending a request with Ruby’s `Net::HTTP` module¹⁶ via a method call `Net::HTTP.get_response('localhost', path, 4567)` from a separate script, where `path` is the request path, WUNDERTAKER runs on port 4567, and is accessed via `localhost`, which means that no data have to be sent over a network but only through the kernel. The time that elapses during this call is measured with Ruby’s `Time` class (see Listing 4.13). This total time is further partitioned into four split times, three of which are measured in WUNDERTAKER’s code. The fourth

¹⁵<https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt>

¹⁶<http://ruby-doc.org/stdlib-2.2.0/libdoc/net/http/rdoc/Net/HTTP.html>

is determined by subtracting the sum of the three first times from the total time. The split times are defined as:

init *init* is reached as soon as WUNDERTAKER takes over from SINATRA. It then reads its configuration file and finds the requested object in its GIT repository.

preprocessing If caching is enabled, the *preprocessing* stage always follows the pattern of connecting to MONGODB, checking there if some information is already available, and generating and writing new data if not. When visualizing folders, there is only a single bit of information: the report counts in the subfolders and subfiles. In case of the code view, WUNDERTAKER finds and reads contents of relevant defect reports, makes UNDERTAKER parse `#ifdef` blocks, and further processes these information.

html *html* is the last stage before WUNDERTAKER hands back control to SINATRA. Here, code is tokenized and syntax highlighted on the fly. After that, the final response HTML document is generated by assembling a long string in memory. Together with the previous two steps this one adds up to the time that passes while WUNDERTAKER's own code is running.

frame This is the difference between the total time and the sum of the three times above and illustrates how much is lost to the web server stack and the framework WUNDERTAKER runs on and in.

```
start_time = Time.now
#[code whose performance is to be tested]
duration_in_seconds = Time.now - start_time
```

Listing 4.13 – Measuring time with Ruby.

The pages I target with my measurements can be summarized in three groups. The first group ❶ comprises the root folders of the trees of the Linux commits 456fdb267, v4.3, and v4.2. In the test setup, there are 3208 folders in the folder containing defect reports for 456, 36 in the folder for v4.3, and none at all in the one for v4.2, testing the effect of the size of the reports database on WUNDERTAKER's performance. The second group ❷ are code views of the files `crypto/testmgr.h`, `kernel/sched/core.c`, `drivers/char/rtc.c`, and `drivers/memory/of_memory.c`, all from Linux v4.3, because they exemplify files of different sizes (1.2MiB, 206KiB, 34KiB, and 4.8KiB, in this order).

The third group ❸ is a measurement where I requested each of the 51556 files from the tree of Linux v4.3 (called 'all in v4.3') and computed the mean split times, in order to stress test and determine the average performance of WUNDERTAKER. Unfortunately, 132 files produced error responses and were omitted for this reason. In all cases, the error resulted from the UNDERTAKER `#ifdef` parsing interface being unable to process the respective files. 19 of the problematic files contain text that is not CPP-annotated and could very well include invalid C tokens. However, the remaining 119 are `.S`, `.c` and `.h` files so I assume a bug on the part of UNDERTAKER. WUNDERTAKER produces HTTP 200 OK responses for all 51424 files that are left.

Every request, or set of requests, was measured four times, resulting in a pair of results. The first result is the mean time elapsed during the first request, and the second one is the mean across the other three measurements. Results suffixed with u or c originate from measurements done with an ‘uncached’ or a ‘cached’ WUNDERTAKER, respectively. The first result is not averaged because a cached WUNDERTAKER fills its cache during the first request to a page, influencing all subsequent requests. The second results tagged with c should be much lower than the first ones if caching is beneficial. When comparing them to the first result, the averaged second results will show that the measured times are constant and hence reproducible.

4.2 Results

The results of the performance measurements are shown in Figure 4.1. One can see that the *init* phase never takes a significant amount of time – it has a constant runtime of less than 5ms. The first group, depicted on the left of the chart, is dominated by the *preprocessing* stage, that is, by the report counting. Note that caching is extremely effective here, with the second result of 456 c and v4.3 c not even visible any more on the chart. However, their first measurements take longer than

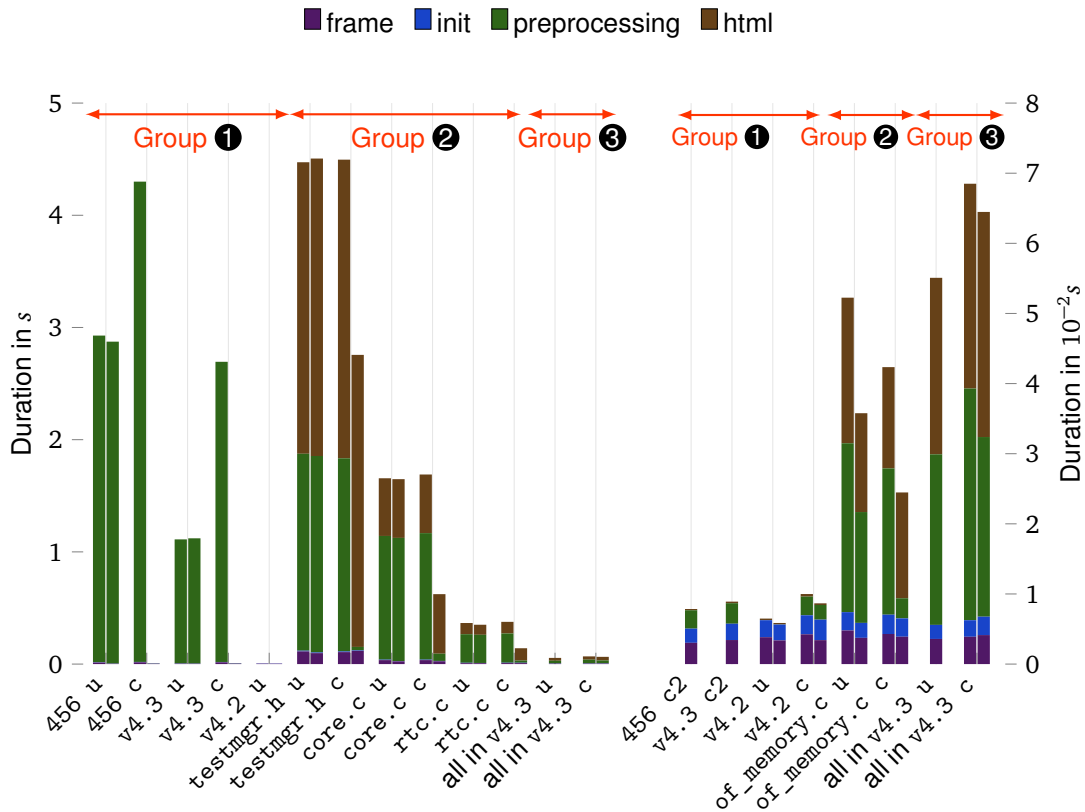


Figure 4.1 – WUNDERTAKER performance measurements. c means ‘cached’, u means ‘uncached’.

the uncached counterparts, in the case of v4.3 more than twice as long. I assume this is because the tests represent a worst-case scenario where the cache is empty and then the root folder is suddenly requested. WUNDERTAKER then traverses the tree and checks for each subdirectory whether its report counts are already stored in the cache, which, in this case, is for naught. Note that when caching is disabled, 456 takes longer than v4.3, which in turn takes longer than v4.2, despite all having roughly equally many files and folders in their GIT tree. This is because WUNDERTAKER does not traverse the `commits` folder containing the reports but rather fetches paths of its subdirectories from the GIT repository and then “probes” `commits` for the existence of reports. 456’s reports database contains the most folders, v4.3’s less, and v4.2 none. So the more folders there are to probe the slower WUNDERTAKER is.

In the second group, the *html* stage, doing syntax highlighting, is much more pronounced. The right bar of a measurement with cache is basically the left one with *preprocessing* much shortened, so the ratio of *preprocessing:html* determines how well the code view of a file can benefit from caching – the greater the better, since *html* stays constant. While caching is generally useful in folder views, the situation here depends on the structure of the files. The visualization of the largest of them, `testmgr.h`, which has 33595 lines of code and 88 `#ifdefs`, is sped up by a factor of almost 2 if caching is enabled. `core.c`, on the other hand, is quite a bit shorter, but it has 133 `#ifdef` blocks and a greater ratio, resulting in a visibly greater speedup when cached. `rtc.c` is even shorter but still contains 54 blocks and has the largest ratio of *preprocessing:html*. It can hence be summarized that heavily annotated files have the largest *preprocessing:html* and are therefore most suitable for caching.

The third group, the two rightmost measurements (‘all in v4.3’), show that the average Linux file does not contain any `#ifdef` blocks at all, like of `_memory.c`. The speedup of ‘all in v4.3’ is hardly visible. One could react to this situation by disabling caching for non-code files. Also in contrast to the folder view, during generation of code view pages, the interaction with MONGODB is negligible – the first bars of the cached measurements look the same as the uncached bars. As a last remark, note that *frame* is about constant everywhere – except in `testmgr.h u/c` where it is quite visible. Most probably, the sheer size of the response HTML document (4MiB) is responsible for this delay. When WUNDERTAKER is accessed over the Internet, *frame* would depend heavily on the connection between client and server.

WUNDERTAKER’s memory consumption, which I measured with the `USED` column of the `top` program, becomes apparent if we look at the ‘all in v4.3’ experiment again. WUNDERTAKER used 35MiB before it started, 1.9GiB after one pass, and 3.2GiB after the second one, suggesting the existence of a serious memory leak. If caching is disabled, WUNDERTAKER ends up with 140MiB of RAM, indicating that the MONGODB interface, or the code that uses it, is probably the root cause here. However, WUNDERTAKER is completely written in garbage collected Ruby and it does not keep any global state that might grow with each request in its own code. Although 3.2GiB is a lot of memory, WUNDERTAKER’s RAM usage becomes so high only after having served over 100,000 code views and can be zeroed by simply restarting the WUNDERTAKER process, which does not incur any

data loss. The MONGODB cache on disk, empty at the beginning, grows from pre-allocated 17MiB to 269MiB, which is reasonable.

Seeing that WUNDERTAKER works, is it useful, too? A promising approach to answer this question would be to conduct a study as so much subjectivity is involved. At this point, I can only reference [Fei+13], which I discussed in Section 2.3, proving that the coloring techniques employed by WUNDERTAKER are effective. Then, Valentin Rothberg and I have received dedicated and helpful feedback regarding WUNDERTAKER from two Linux developers. Greg Kroah-Hartman’s response was purely positive, saying that WUNDERTAKER indeed makes it easier to see redundant CPP annotations that ought to be removed and generally praised the value of WUNDERTAKER as a didactic tool, saying, “This is great stuff, I really like it, it makes it easy to see and understand.”

Paul Bolle was “quite impressed” with three examples of WUNDERTAKER sent to him, all still development versions without `.analysis` display. His first bit of criticism is interestingly in accord with my design principle mentioned in Section 3.2.2, namely that block presence condition formulas should be optional. However, I still chose to show them by default in a column next to the code (Figure 3.3), something he says makes the GUI “a bit cluttered.” Secondly, he points out that WUNDERTAKER does not show any relationships between code and `Makefiles`. Although WUNDERTAKER now includes the `.analysis` files that inform about ϕ_{KBUILD} , this is correct and further work would be needed to thus extend WUNDERTAKER. He also mentions that the `#else` branch of a defective `#ifdef` block is shown as a separate defect, all while correctly assuming that connecting both is difficult to achieve – indeed, I think a sensible implementation that features this functionality would need to introduce heuristics at a lower level than WUNDERTAKER. Finally, an interesting suggestion of his is a mode that shows only defective blocks and their parents and hides everything else.

CONCLUSION

In this work, I presented the program WUNDERTAKER implemented with techniques that ensure a performance and robustness that allows to process almost any Linux code file within a reasonable time frame, even if WUNDERTAKER runs as a much-accessed web application. Its visualization follows carefully designed and proven principles and has received praise from actual Linux developers. Suggestions for improvement so far, apart from proposals for further work, consist of minor points or target the underlying UNDERTAKER toolchain.

WUNDERTAKER's main purpose was as a didactic tool in the `linux-next` experiment. It fits into the experiment's workflow by being a web application e-mails can link to. It incorporates UNDERTAKER-CHECKPATCH's output, visualizes the hierarchy of `#ifdefs` with bars that illustrate their boundaries, also seen in [Fei+11]. It uses code folding, as seen in [KTA08], and follows the conclusions drawn from the studies in [Fei+13] to construct a set of colors that ease the understanding of an annotated code file. It can thus serve as a substitute for the basic portions of manual explanations of *code* and *missing* defects in the `linux-next` experiment. The intricacies of *kbuild* and *kconfig* defects still need to be described, though, because WUNDERTAKER can only visualize a single file at a time. In Section 4.2, WUNDERTAKER's maximum response time was around 4s and its average response time for a code view around 60ms. Moreover, its RAM usage stays within a reasonable limit, if it is restarted periodically, and its use of MONGODB for caching purposes is especially useful in folder views, where a speedup of over 250 was measured. In conclusion, WUNDERTAKER is also technically suitable to be used in a production environment like the `linux-next` experiment.

Possible subjects for further work arise from the challenges that were met during this work, as well as from comments by Linux developers. Taking up Paul Bolle's suggestions from the preceding section, WUNDERTAKER has a few places where it could be improved. The defective-blocks-only mode he mentioned could be implemented in a straightforward way, but it would need some work. Then, within the framework of the UNDERTAKER toolchain, the next logical step in WUNDERTAKER's development would probably be a dedicated visualization of Makefiles and Kconfig files with interactive shortcuts to and from `#ifdef` conditions, thereby completing a triad of visualizations for the three stages of the Linux build system.

There are reasons for WUNDERTAKER being a web application (see Section 3.1.1), but it may also be advantageous to implement it as an Eclipse plug-in or as an extension of UNDERTAKER's emacs mode¹⁷ in the future. Given an UNDERTAKER-CHECKPATCH that is fast enough, it could then highlight defective `#ifdef` conditions by constructing a patch file on-the-fly while changes to code are typed into the computer even before they are published. Another topic for further development, which is already closer to implementation, would be a WUNDERTAKER that works more like GitLab, contrary to the design requirement in Section 3.1.2. It could be an online platform visualizing arbitrary software projects – this is already possible with the current version of WUNDERTAKER – and providing graphs and statistics about feature models, how features are distributed across files, and, most importantly, about defects.

¹⁷see <http://vamos.informatik.uni-erlangen.de/trac/undertaker/wiki/UndertakerElDocumentation>

LIST OF ACRONYMS

| | |
|-------------|------------------------------------|
| CPP | C preprocessor |
| GUI | Graphical User Interface |
| LKM | Loadable Kernel Module |
| IDE | Integrated Development Environment |
| URL | Uniform Resource Locator |
| HSL | Hue, Saturation, Lightness |
| IO | input / output |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| MUS | Minimal Unsatisfiable Subset |

LIST OF FIGURES

| | | |
|-----|---|----|
| 2.1 | <code>make menuconfig</code> TUI of Linux v4.3 – one of the KCONFIG front ends. Each entry represents a feature. | 3 |
| 2.2 | Summary of the Linux build system and how it is examined by <code>UNDERTAKER</code> | 10 |
| 3.1 | A screenshot of the dolphin file browser showing the contents of a folder containing both folders and files. Things to note are the clickable path (1), the separation between folders and files, and the columns that supply further information on each item (2). <code>WUNDERTAKER</code> should try to stick close to this, as it is a tried approach, making <code>WUNDERTAKER</code> consistent with programs already known to users. | 16 |
| 3.2 | General layout of a <code>WUNDERTAKER</code> page. | 20 |
| 3.3 | Code view of <code>arch/powerpc/mm/hash_low_64.S</code> in Linux v4.3. | 21 |
| 3.4 | Code view of <code>arch/powerpc/include/asm/page.h</code> of Linux commit 456fdb267. | 22 |
| 3.5 | The top of the view of folder <code>arch/powerpc/mm</code> in Linux v4.3. | 23 |
| 3.6 | The HSL system. | 24 |
| 3.7 | The vertical design of <code>WUNDERTAKER</code> | 27 |
| 3.8 | The horizontal design of <code>WUNDERTAKER</code> . Boxes with thick borders stand for programs that produce information on their own. The HTML generator is in fact a part of <code>WUNDERTAKER</code> and therefore has a dashed border. Databases have rounded corners. | 28 |
| 4.1 | <code>WUNDERTAKER</code> performance measurements. c means ‘cached’, u means ‘uncached’. | 31 |

LIST OF TABLES

| | | |
|-----|---|----|
| 2.1 | Encoding (a_0, a_1) of a tristate symbol A. The last column is relevant in Section 2.1.3.1. | 5 |
| 2.2 | The different variations of Listing 2.5 depending on the configuration of A and B. | 10 |
| 2.3 | UNDERTAKER defect classes. The m_i are CPP symbols missing from KCONFIG, see Section 2.2.2.4. For a more detailed version of the two rightmost columns, see Sections 2.2.2.1 to 2.2.2.5. MUS stands for Minimal Unsatisfiable Subset. | 12 |
| 3.1 | Fixed S and L values and the distribution of hues in the different colored GUI elements. Saturation values become larger from row one to the last row while Lightness values become smaller in order to make text colors appear ‘stronger’ and darker than the light background. Bar colors are in between. | 25 |

LIST OF LISTINGS

| | | |
|------|--|----|
| 2.1 | Lines 1–3 from <code>net/mac80211/Kconfig</code> of v4.3. | 4 |
| 2.2 | Lines 49–105 from <code>drivers/cpufreq/Kconfig</code> of v4.3. | 5 |
| 2.3 | Snippets from <code>include/generated/autoconf.h</code> I had generated from the variants active in Figure 2.1b and Figure 2.1c, respectively. Note that in an <code>autoconf.h</code> , all symbols are prefixed with <code>CONFIG_</code> | 8 |
| 2.4 | When processed with the CPP, the presence of the character <code>a</code> in the output would depend on the parent block <code>#ifdef CONFIG_A</code> to be present and the first two chain blocks to <i>not</i> be present. Writing bits (a_0, a_1) for feature A, and so on, the presence condition of the <code>a</code> is $a_0 \wedge \neg(b_0 \wedge c_1) \wedge \neg d_0$ | 8 |
| 2.5 | CPP-annotated C code resulting in variations illustrated in Table 2.2. | 9 |
| 2.6 | The outer block is not dead, but the inner one is dead, in combination with the outer one. <code>#ifndef</code> stands for “if not defined.” | 12 |
| 3.7 | Analysis of a <i>missing</i> defect that UNDERTAKER-CHECKPATCH found in Linux v4.3. . . . | 17 |
| 3.8 | Snippet from <code>arch/powerpc/mm/hash_low_64.S</code> from v4.3. | 17 |
| 3.9 | UNDERTAKER-CHECKPATCH analysis of the <i>code</i> defect. | 18 |
| 3.10 | Presence condition of block 33 in <code>arch/powerpc/include/asm/page.h</code> of Linux commit 456fdb267. | 18 |
| 3.11 | UNDERTAKER-CHECKPATCH analysis of a <i>kbuild</i> defect. | 18 |
| 3.12 | A HTTP request. | 26 |
| 4.13 | Measuring time with Ruby. | 30 |

REFERENCES

- [CPR07] David Coppit, Robert R Painter, and Meghan Revelle. “Spotlight: A prototype tool for software plans.” In: *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society. 2007, pp. 754–757.
- [Die+12] Christian Dietrich et al. “A robust approach for variability extraction from the Linux build system.” In: *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM. 2012, pp. 21–30.
- [Fei+11] Janet Feigenspan et al. “FeatureCommander: Colorful #Ifdef World.” In: *Proceedings of the 15th International Software Product Line Conference, Volume 2*. SPLC ’11. Munich, Germany: ACM, 2011, 48:1–48:2. ISBN: 978-1-4503-0789-5. DOI: 10.1145/2019136.2019192. URL: <http://doi.acm.org/10.1145/2019136.2019192>.
- [Fei+13] Janet Feigenspan et al. “Do background colors improve program comprehension in the# ifdef hell?” In: *Empirical Software Engineering* 18.4 (2013), pp. 699–745.
- [Hen15] Stefan Hengelein. “Analyzing the Internal Consistency of the Linux KConfig Model.” MA thesis. University of Erlangen, Dept. of Computer Science, 2015. URL: <https://www4.cs.fau.de/Ausarbeitung/MA-I4-2015-04-Hengelein.pdf>.
- [KTA08] Christian Kästner, Salvador Trujillo, and Sven Apel. “Visualizing Software Product Line Variabilities in Source Code.” In: *SPLC (2)*. 2008, pp. 303–312.
- [Loh+06] Daniel Lohmann et al. “A quantitative analysis of aspects in the eCos kernel.” In: *ACM SIGOPS Operating Systems Review*. Vol. 40. 4. ACM. 2006, pp. 191–204.
- [Nad+13] Sarah Nadi et al. “Linux variability anomalies: What causes them and how do they get fixed?” In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press. 2013, pp. 111–120.
- [Rot14] Valentin Rothberg. “Years of Variability Bugs in Linux - How to Avoid Them.” MA thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2014. URL: <https://www4.cs.fau.de/Ausarbeitung/MA-I4-2014-07-Rothberg.pdf>.
- [Rup15] Andreas Ruprecht. “Lightweight Extraction of Variability Information from Linux Make-files.” MA thesis. University of Erlangen, Dept. of Computer Science, 2015. URL: <https://www4.cs.fau.de/Ausarbeitung/MA-I4-2015-01-Ruprecht.pdf>.

-
- [Sin+10] Julio Sincero et al. “Efficient extraction and analysis of preprocessor-based variability.” In: *ACM SIGPLAN Notices*. Vol. 46. 2. ACM. 2010, pp. 33–42.
- [Sin13] Julio Sincero. “Variability Bugs in System Software.” PhD thesis. Erlangen: Friedrich-Alexander-Universität Erlangen-Nürnberg, 2013. URL: <http://opus4.kobv.de/opus4-fau/files/3317/diss.pdf>.
- [Spe92] Henry Spencer. “#ifdef Considered Harmful, or Portability Experience with C News.” In: *In Proc. Summer’92 USENIX Conference*. 1992, pp. 185–197.
- [Tar+11] Reinhard Tartler et al. “Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem.” In: *Proceedings of the sixth conference on Computer systems*. ACM. 2011, pp. 47–60.
- [Wik15] Wikipedia. *Four color theorem* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 31-December-2015]. 2015. URL: https://en.wikipedia.org/w/index.php?title=Four_color_theorem&oldid=695588105.
- [ZK10] Christoph Zengler and Wolfgang Küchlin. “Encoding the Linux kernel configuration in propositional logic.” In: *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration*. Vol. 2010. 2010, pp. 51–56.