Analyzing the Impact of Feature Changes in Linux*

Andreas Ziegler
Friedrich-Alexander-Universität
Erlangen-Nürnberg
ziegler@cs.fau.de

Valentin Rothberg
Friedrich-Alexander-Universität
Erlangen-Nürnberg
rothberg@cs.fau.de

Daniel Lohmann
Friedrich-Alexander-Universität
Erlangen-Nürnberg
lohmann@cs.fau.de

ABSTRACT

In a software project as large and as rapidly evolving as the Linux kernel, automated testing systems are an integral component to the development process. Extensive build and regression tests can catch potential problems in changes before they appear in a stable release. Current systems, however, do not systematically incorporate the configuration system KCONFIG. In this work, we present an approach to identify relationships between configuration options. These relationships allow us to find source files which might be affected by a change to a configuration option and hence require retesting. Our findings show that the majority of configuration options only affects few files, while very few options influence almost all files in the code base. We further observe that developers sometimes value usability over clean dependency modelling, leading to counterintuitive outliers in our results.

CCS Concepts

•**Software and its engineering** → **Software product lines;** *Software defect analysis;*

Keywords

Configurability, Linux, Kconfig, CADOS

1. INTRODUCTION

The Linux kernel is a highly dynamic software project. On average, close to 200 changes are merged into its central repository every day [14] – and this does not include the patches which developers propose for submission on the mailing lists. This speed in development requires a high amount of automation in the process of finding possible issues with proposed changes.

The "usual" process a change has to go through involves several people: After writing the patch, the developer has to send it to the *maintainer* of the modified files in question – each file (or subsystem) in the Linux kernel is assigned to a designated developer from the respective domain. The *maintainer* will have a look at the proposed changes, try to clear up questions which might arise and recommend amendments if the patch does not meet his or her requirements. If the maintainer decides that the change should be included in the next Linux release, he or she will pick up the patch into their own repository, and ultimately write a *pull request* to Linus Torvalds, who will then integrate the patch into the next mainline Linux kernel.

Before a patch makes it into a Linux kernel release, it will be subject to some automated testing: Experienced developers, such as Jim Davis or Stephen Rothwell, the maintainer of the `linux-next` tree – which is updated daily with proposed changes – run automated compile tests with randomly generated configurations on their trees, and if the build fails, send a mail with the problem they discovered and the offending configuration to the Linux Kernel Mailing List (LKML). Note that in order to properly test the compilation with a sequence of randomly generated configurations, the object files have to be cleaned from the source tree and every file possibly needs to be recompiled with every new configuration.

Some more sophisticated testing is done by the so-called "0day" infrastructure by the Intel Open Source team [9, 13]. Their infrastructure runs build, boot, and performance tests for a number of commits in `linux-next`, the mainline and developer's repositories using over 140 different configurations. Half of these configurations are randomly generated, while the other half are configurations custom-built for testing. After compiling individual commits with all these configurations and statically checking the code using `sparse` [15] and `Coccinelle` [17], the resulting kernels are booted in a virtual machine and checked for any regressions resulting in performance degradation or unstable behaviour.

The current testing systems already uncover some bugs which are caused by inconsistencies in the configuration system; they are, however, only found by chance. Using randomly generated configurations might at some times lead to an error, while at other times the random selection will have picked a (seemingly) good configuration. In other words: While different configurations are already used, the testing infrastructures do not use a *systematic* approach to find adverse effects resulting from faulty uses of configurability inside the code as they do not employ any variability-aware sampling strategies.

To check Linux for such variability-related defects, we have

*This work was partly supported by the German Research Council (DFG) under grant no. LO1719/3-1

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

VaMoS '16, January 27-29, 2016, Salvador, Brazil
© 2016 ACM. ISBN 978-1-4503-4019-9/16/01…$15.00
DOI: http://dx.doi.org/10.1145/2866614.2866618

created a tool called undertaker-checkpatch. This tool uses the UNDERTAKER suite [24], which transforms the structure of #ifdef blocks inside the source files into a boolean formula. This file-internal formula is then enriched with dependency information from the configuration system KCONFIG. UNDERTAKER then applies a SAT solver to the formula to find dead or undead #ifdef blocks in the source code. We run undertaker-checkpatch on a daily basis, checking all new commits in the linux-next tree. For that, we use the information from the commit itself about the files it modifies, and add these files to a worklist. By comparing the output of UNDERTAKER for the states of all these files before and after the patch is applied, we can derive if a patch introduces (or eliminates) any variability-related defects.

One crucial point, however, is missing in all mentioned testing procedures: Up to now, no mapping exists between a configuration option and the files it has an influence on (directly or indirectly through other options from the KCONFIG system). Thus, when we encounter a commit which changes the definition of a configuration option, we currently do not know which files might possibly be affected by that change, and consequently do not know which files must be rechecked for build errors or variability defects.

In this paper, we present an algorithm that allows us to quickly identify the impact of a change in KCONFIG on the source code. To accomplish this, we leverage the information about dependencies between options from KCONFIG to build a mapping of *related* configuration options. From this mapping, it is possible to determine a list of source files which *might* be affected by a change to a given configuration option.

The remainder of the paper is structured as follows: In Section 2, we present an overview of KCONFIG, the configuration system of Linux, showing how configuration options can be specified and connected with each other. Then, we describe our algorithm to find *related* symbols as well as the corresponding source files in Section 3. We present an evaluation of our findings in Section 4, giving an insight into the structure of dependencies inside the Linux feature model as well as their influence on the source code. Subsequently, we discuss the results in Section 5, and elaborate on some extraordinary cases we encountered during our analysis. After presenting related work in Section 6, the paper concludes with an outlook on future possible applications of and improvements to our algorithm.

2. BACKGROUND

In the following, we shall describe how configurable options are defined and used in KCONFIG, and how they can influence each other.

The KCONFIG language allows developers to specify configuration options which can have a short description of the option, a type, for example, int or hex for numbers, bool for options that can either be enabled or disabled, or tristate for options which allow the corresponding code to be built as a loadable kernel module (LKM). KCONFIG options can also entail dependencies, thus allowing the user to enable them only if some other options – which might provide some necessary functionality – have already been enabled earlier. The following example for a configuration option is taken from drivers/hid/usbhid/Kconfig, Linux v4.3.

```
4  config USB_HID
5      tristate "USB HID transport layer"
6      default y
7      depends on USB && INPUT
8      select HID
9      ---help---
10       Say Y here if you want to connect USB
11       keyboards, mice, joysticks, graphic
12       tablets, or any other HID based
13       devices to your computer via USB,
14       [...]
```

The developer defines the configuration option USB_HID which depends on USB and INPUT, meaning it is only possible to enable this configuration option if both dependencies have been enabled previously – if USB or INPUT have not been enabled, the user simply is not prompted to configure USB_HID by KCONFIG. The default setting for USB_HID is y, meaning it is enabled; note that the default value of a configuration option can also be a boolean expression entailing multiple other configuration options.

Additionally, KCONFIG supports select statements, which are also called "reverse dependencies". With select, enabling an option forces another configuration option to become enabled as well – in the example, enabling USB_HID will unconditionally enable HID. select statements can further have conditions, meaning that the corresponding option is only forced to be enabled if the configurational conditions are met. In general, the dependencies between configuration options can become arbitrarily complex and span across different subsystems of the kernel (see [22]).

When a user configures the Linux kernel, all enabled configuration options are collected in automatically generated header files which are made available to the build system and the source code. The kernel's build system KBUILD uses the information about configured options in its Makefiles to make a coarse-grained selection about which source files are included into the compilation process. As the generated header file is included into every source file by the compiler, the configuration options (and their configured values) are available to the C preprocessor. Therefore, developers can use #ifdef statements for a more fine-grained selection inside the source files, as to include or exclude blocks of code depending on the selected configuration options.

3. APPROACH

In this section, we present our approach to find relationships among configuration options from the KCONFIG system, and explain how we map these options to files which need to be retested due to a change made to a configuration option.

3.1 Preparations

First, we need the information about all configuration options defined by KCONFIG including their respective dependencies and select statements. We use a modified standalone version of the conf parser used by Linux itself, which is also part of the UNDERTAKER suite. The result of this step is a feature model containing all information about the KCONFIG features in the Rigi Standard Format (RSF). Due to the structure of the configuration process of Linux, which requires the user to define a target architecture before starting the actual configuration process, one model is gen-

erated per architecture, representing the information which would also be available for the user configuring Linux for that architecture from KCONFIG.

After reading the model for an architecture, we preprocess it for the later steps. Initially, we clean up the feature model: As the `conf` parser only reads all configuration files but does not check the dependencies on a semantic level, the feature model for an architecture might also contain features which can never be enabled on the analyzed architecture (e.g., if the configuration option for a driver `depends on MIPS`, the driver can never be enabled when configuring a Linux kernel for the `X86` architecture but is still present in the feature model for `X86`). Including configuration options into our analysis which are not even available on the currently analyzed architecture would distort the results about relationships between configuration options. Thus, we use a SAT solver to eliminate such unselectable configuration options before running the further steps.

Additionally, we need a mapping that allows us to quickly look up the set of files that reference and use a given configuration option. To generate this mapping, we perform a scan of the entire source code for uses of configurable options. In order to include the influence of coarse-grained variability, we also extract information from the build system, that is, which configuration options have to be enabled for a file to be included into the compilation process and thus, the resulting kernel image. For this purpose we developed a text-based extractor using regular expressions which we have shown [21] to be as precise as the previously employed robust, but slow GOLEM extractor by Dietrich et al. [3].

3.2 Worklist algorithm

From the cleaned-up feature model, we can now construct the set of configuration options Rel_C which are transitively *related* to a given initial option C. We use a worklist algorithm to process all relevant options. The worklist is initialized with the configuration option in question (i.e., the option which has been modified by a patch). We then take one element from the worklist and add it to the result set Rel_C until there are no more items left to process. For the element *cur* we just removed from the worklist, we add the following new configuration options to the worklist:

- All configuration options which are `select`ed by *cur*.
- All configuration options which `depend on` *cur*.
- All configuration options which can not manually be selected in KCONFIG (i.e., which have no prompt in the configuration interface), but have *cur* as their `default` value.
- All configuration options which are `select`ed under a condition involving *cur* (for example, the presence of a statement like "`select` *FOO* `if` *cur*" would add *FOO* to the worklist when processing *cur*).

To speed up processing, we cache the constructed set of *related* configuration options for a symbol and use this precalculated information when we encounter that symbol in the worklist during the analysis of another option.

3.3 Finding affected files

Once we know the set of configuration options which are related to a given option, we can use the information from the build system and the code to find all files which might be affected through direct references to configuration options in the related set.

The set of affected files is constructed by collecting all files which reference a configuration option contained in the set of *related* configuration options. More formally: Given the set of configuration options *related* to C as $Rel(C)$, the set of files which directly reference a configuration option C in the code as $F(C)$, the set of files which contain C in their build system condition as $B(C)$, the set of files affected by a change of the configuration option C, $affected_directly(C)$ is constructed as follows:

$$affected_directly(C) = \bigcup_{c \in Rel(C)} \{F(c) \cup B(c)\}$$

However, due to the way Linux uses `#ifdef` to provide a different implementation depending on if the corresponding configuration option has been set or not, this is not sufficient to detect certain classes of errors.

3.4 The #include graph

Developers are encouraged to avoid using the C preprocessor and `#ifdef` statements in .c files by the central coding guidelines.[1] Instead, developers should "use such conditionals in a header file defining functions for use in those `.c` files, providing no-op stub versions in the `#else` case, and then call those functions unconditionally from `.c` files". This structuring rule leads to the identification of many headers (which directly use the configuration option) as relevant files, but causes our simple approach from above to miss the `.c` source files which `#include` these headers. The code inside the `.c` files, however, might very well be subject to variability-related errors when it is compiled with all headers included.

In order to additionally find the source files which might be affected by a modification of a configuration option only used inside a header file, we need to construct the (directed) *include graph* for all source files. With the *include graph*, we can generate the transitive set of included header files for each source file, and more importantly, the "reverse mapping" $included_by(h)$ of the `#include` structure: Given a header file h, $included_by(h)$ constitutes the set of source files which (transitively) include that header.

Thus, we can extend the list of files in $affected_directly(C)$ to find all files possibly affected by a change in a configuration option C: For every header file h in $affected_directly(C)$, also consider the source files in $included_by(h)$ as relevant.

$$all_affected(C) = affected_directly(C) \cup \bigcup_{\substack{f \in affected_directly(C),\\ is_header(f)}} included_by(f)$$

While this set might be an overapproximation (as not all source files use all conditional and unconditional functionality from every header they include), it provides a good starting

[1] Located at `Documentation/CodingStyle` in the Linux source code and considered the official reference for coding style in Linux.

point for further analysis. Note that we use a simple text-based scan to find #include statements – this might not be 100 percent accurate due to the presence of computed includes or macros; however, we found these special cases to be the exception rather than the rule. Therefore, we chose to favor the much better runtime (around 60 seconds) over the higher accuracy offered by tools like SuperC [8] or TypeChef [11], which have a total runtime of over 12 and 52 hours, respectively, for all files in the Linux kernel tree.

4. EVALUATION

In the following, we present the results we obtained using the approach given in Section 3. We use the latest stable Linux release available at the time of writing (v4.3, released on November 2nd, 2015). Note that for this work, we restrict the analysis to the X86 architecture – while there are no technical restrictions to do so, it simply would not be possible to present the data for all thirty architectures as part of this paper. All experiments were conducted on a machine equipped with a quad-core Core i7 CPU with 3.4 GHz and 8 GiB RAM.

4.1 Performance

Calculating the relationships between all configuration options and mapping the relationships to affected files takes around 9.5 minutes. The process of eliminating unselectable configuration options from the model accounts for the majority of runtime, taking roughly 8 minutes, or 85 percent of the total time, due to the fact that the SAT solver needs to be invoked once for every configuration option present in the original model – this equals to 11,067 invocations.

The worklist algorithm described in Section 3 computes all relationships between configuration options in 1.5 seconds. Finding direct references of configuration options in the code requires one simple pass over all source files in the kernel tree and takes about 2 seconds to complete. The remainder of the runtime is spent constructing the include graph from the 28,916 source files available for X86[2] and building the resulting mapping from the configuration options to all affected files, which takes roughly one minute.

4.2 Relationships between options

First, we show statistics about the relationships between configuration options as determined by the worklist algorithm. Out of the original 11,067 configuration options in the model, only 8,731 remain after eliminating unselectable options. Figure 1 contains an excerpt of a histogram which is grouped by the number of configuration options contained in the resulting set of related options. Here, we see that 4,787 configuration options, or 54.8 percent of all options, only have one configuration option they are related to – which is the respective option itself. The *arithmetic mean* value of related options, however, is 149.9, which stems from the fact that some configuration options have many related options. When we look at the data computed for all configuration options, we see that there are 281 configuration options, or 3.2 percent, which have more than 1,000 related options.

[2]The number of files available for the chosen architecture is calculated using git ls-files and excluding the Documentation/ directory as well as all directories under arch/ except arch/x86/ - the latter directories represent the separate hardware abstraction layers for the individual target architectures. Furthermore, we prune all files which have a build system condition that can not be enabled on X86 by building the conjunction of CONFIG_X86 and the build system condition and checking solvability using a SAT solver.

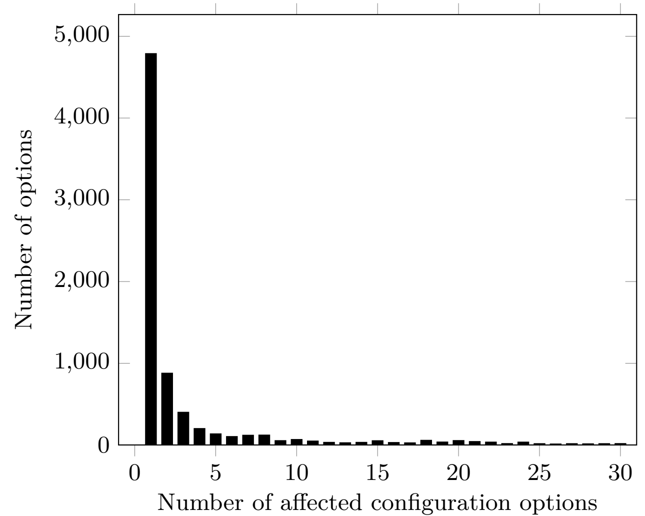

Figure 1: Histogram of configuration options grouped by the number of options they are related to as calculated by the worklist algorithm (Linux v4.3, x86 architecture). The plot does not show data for configuration options related to more than 30 options.

Table 1: The ten configuration options in the X86 architecture on Linux v4.3 which have the largest scope, that is, the largest number of affected configuration options by them.

Option	Number of affected options
X86_32	6,623
BLOCK	6,737
X86	6,754
NET	6,814
HAS_IOMEM	7,638
PCI	7,651
UNISYSSPAR	7,654
STAGING	7,773
X86_64	7,805
64BIT	7,984

In Table 1, we show the ten configuration options with the largest scope along with the number of options they affect, that is, the ten configuration options with the largest set of affected options. The options with the largest scope are mostly directly associated to the architecture (X86, X86_32, X86_64 as well as 64BIT) or to the largest subsystems in the Linux kernel (PCI, NET and BLOCK). One interesting exception is UNISYSSPAR – representing a single driver for secure partitioning in mainframes – which we will discuss in greater detail in Section 5.

4.3 Files with direct references

We will now extend the results to affected files, but without taking the include graph into account (corresponding to *affected_directly* presented in Section 3.3). The results show

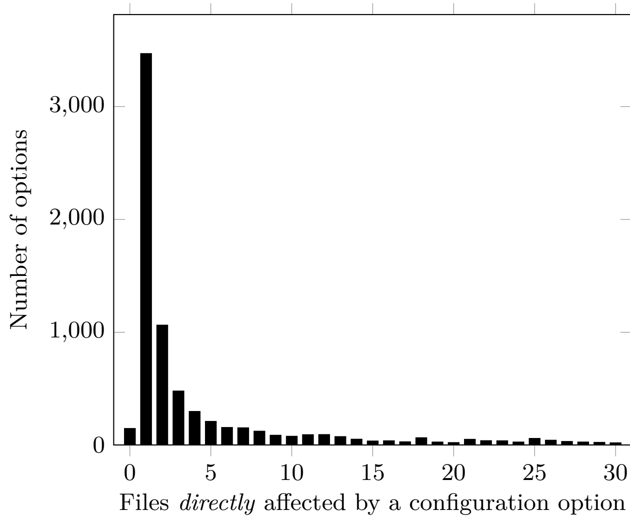

Figure 2: Histogram of configuration options grouped by the number of source files they *directly* affect (Linux `v4.3`, `x86` architecture). The plot does not show data for configuration options affecting more than 30 files.

Table 2: The ten configuration options in the `X86` architecture on Linux `v4.3` which affect the most files by means of direct references to configuration options contained in their set of related options.

Option	**Number of affected files**
`MMU`	10,225
`NET`	10,483
`BLOCK`	10,672
`HAS_IOMEM`	11,161
`X86`	11,413
`PCI`	12,447
`UNISYSSPAR`	12,452
`X86_64`	12,875
`STAGING`	12,876
`64BIT`	13,180

us how many files reference configuration options that we found to be related to a given option (and thus may need checking after that option has changed).

From the 8,731 selectable configuration options, 3,469 options (or 39.7 percent) only affect *one* source file; in addition, we found 146 configuration options which are not referenced from any source file – these options mostly change compiler flags or are used by KCONFIG internally to model more complicated dependencies. The *median* number of affected files is 2, meaning that half of all configuration options only influence one or two files. The *arithmetic mean* value is, as before, much higher, with 240.1 influenced files. These results resemble the observations made when we looked at just the configuration options alone – there are many configuration options with an influence on only few files, but there are also a few configuration options which affect many files. Consequently, the histogram in Figure 2 looks very similar to Figure 1. Again, we list the ten most "influential" options, that is, the ten configuration options which (through their relationship in KCONFIG) affect the most files in Table 2. Except for `MMU` – which replaces `X86_32` at the tenth place – we see the same configuration options at the top as in Table 1.

4.4 Files affected through the #include graph

As stated earlier, the Linux developers are encouraged to use `#ifdef` statements in headers instead of the source code, providing empty implementations in case a configuration option is turned off. Source files which `#include` such headers might then have build or other errors without directly referencing the offending configuration option. Our approach handles this structure by using the `#include` graph to find all source files which transitively `#include` a header and including those source files into the result (see Section 3.4).

In Figure 3, we see that the majority of configuration options (3,276, or 37.5 percent of all configuration options) still only affects one single source file. Furthermore, we observe that all values in the displayed range are lower than in the previous measurement – this indicates that there are now more configuration options affecting more than 30 files. Consistently, the *median* value of affected files is now 3 (as opposed to 2 when not considering `#include`s). The *arithmetic mean* value increases more than tenfold to 2,940 possibly affected files per configuration option. While these numbers surely are an overapproximation – no source file uses *all* functionality offered by *every* header it transitively includes – this, in addition to the coding guidelines which every developer should follow, shows us how important it is to take the `#include`d header files into the analysis.

Once more, in Table 3 we show the ten most "influential" configuration options. The architectural options (`X86, X86_32, X86_64` and `64BIT`) still have the largest scopes, as well as the `PCI` subsystem, `STAGING` and `UNISYSSPAR`, with the options `EMBEDDED` (allowing the configuration of certain options that are reasonable to consider on embedded systems), `EXPERT` (which allows tweaking additional standard kernel configuration options for expert users), and `SMP` (for

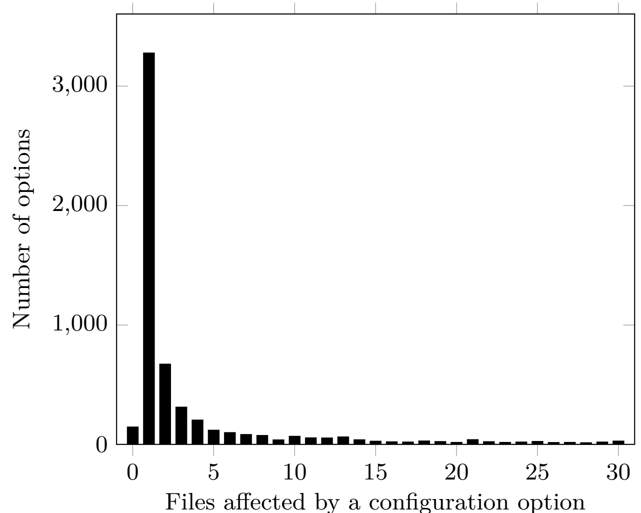

Figure 3: Histogram of configuration options grouped by the number of source files they affect when also considering `#include`d headers (Linux `v4.3`, `x86` architecture). The plot does not show data for configuration options affecting more than 30 files.

Table 3: The ten configuration options in the X86 architecture on Linux v4.3 which affect the most files (including references from #included headers) through their set of related configuration options.

Option	Number of affected files
SMP	21,342
EXPERT	21,374
EMBEDDED	21,374
X86	21,392
X86_32	21,417
PCI	21,433
UNISYSSPAR	21,433
STAGING	21,434
X86_64	21,495
64BIT	21,615

symmetric multiprocessing support) newly showing in the listing.

Having 64BIT as the most "influential" configuration option – with and without the #include graph – is not a surprise: The sub-architectures (X86_32 and X86_64) both depend on 64BIT to be enabled or disabled, respectively. Hence, through our approach we also mark all configuration options as relevant which have a KCONFIG dependency on the sub-architectures – which is the case for all architecture-specific device drivers. Similarly, when looking at the #include graph, we found that the file most often #included is include/asm-generic/bitsperlong.h (which is transitively included by 23,161 other files). This file contains the type definition for the long data type and is thus required by every file in the Linux kernel which uses that data type.

5. DISCUSSION

In this section, we will discuss the results and elaborate on their implications.

5.1 Under- vs. Overapproximation

First, we will take another look at the difference between the results for *directly* affected files and files affected through the reverse #include mapping.

Due to the fact that developers are encouraged to implement variability in header files instead of the source code, some configuration options might not be referenced directly at all from .c files. A simple approach – such as looking for direct references only – will then not find all source files possibly affected: The files might still contain defects related to configuration options only mentioned in the header files, for example, through contradictory KCONFIG dependencies of another #ifdef block surrounding a call to a conditionally defined function in the code. In this case, our results represent an *under*approximation of the real number of affected files.

On the other hand, when we take all files into account which (transitively) #include a header file containing the configuration option in question, we create an *over*approximation in terms of actually affected files. Often, a header file is included to be able to use only a few functions or data types from that header. As some header files in Linux are very large and #include many other header files themselves, this effect is amplified for every additional level of #includes.

We argue that the overapproximation does not adversely affect our approach: If some file which in fact does not use any #ifdef-guarded functionality from the headers it #includes is handed to a downstream analysis, for example to the UNDERTAKER tool or the TYPECHEF [12, 16] tool, these tools will not report variability-related defects. On the contrary, if our approach does *not* find a source file just because it does not directly reference a configuration option, downstream analysis tools might miss more complicated types of defects which are caused by interactions between the source code and header files – precisely the kinds of defects we need those tools for.

5.2 Usability vs. Clean Modelling

We will now take another look at one of the most influential configuration option presented in Section 4, UNISYSSPAR. This configuration option controls the compilation of a single driver which provides support for a secure partitioning of multiple operating-system instances running on the same machine [26].

It seems surprising that a specific driver appears among the most influential configuration options – to understand these results, we need to take a look at the definition of the corresponding KCONFIG option (located at drivers/staging/unisys/Kconfig in Linux v4.3):

```
 8 menuconfig UNISYSSPAR
 9     bool "Unisys SPAR driver support"
10     depends on X86_64 && !UML
11     select PCI
12     select ACPI
13     ---help---
14     Support for the Unisys SPAR drivers
```

Line 10 tells us that UNISYSSPAR is only available if the X86_64 option has been enabled earlier, and the kernel is not compiled as a User Mode Linux (UML). More interestingly, the lines 11 and 12 select the configuration options PCI and ACPI. These statements *force* PCI and ACPI to be turned on when the user enables the UNISYSSPAR configuration option; in KCONFIG terms, the UNISYSSPAR driver has a *reverse dependency* on PCI and ACPI.

In our algorithm, the select statements lead to the inclusion of all configuration options that depend on PCI into the result set for UNISYSSPAR – as PCI is one of the largest subsystems in the kernel, we end up with more than 6,700 configuration options detected as related to UNISYSSPAR. Note that UNISYSSPAR is not the only driver making such use of select statements; among others, the configuration options SCSI_CXGB3_ISCSI and SCSI_CXGB4_ISCSI also have more than 5,900 configuration options related to them, due to selects on NETDEVICES and ETHERNET, which are top-level configuration options for all networking devices in Linux.

The possibility to *force* other configuration options to a desired value is convenient for users configuring the kernel – they do not need to manually enable PCI and ACPI before the UNISYSSPAR configuration option becomes visible, rather they only need to enable UNISYSSPAR. From a modelling perspective, however, this is questionable: If a driver depends on some functionality like PCI or ACPI, these dependencies should also be modelled using depends on statements

– utilising `select` instead inverts the original intention. Furthermore, the neglection of dependencies for the target of a `select` statement is not intuitive at all and, among other problems, complicates ensuring consistency of the configuration [1].

The kernel's documentation for KCONFIG[3] states that `select` should be only be used for "non-visible symbols (no prompts anywhere) and for symbols with no dependencies", as the `select` statement will "force a symbol to a value without visiting the dependencies". It is therefore possible to generate invalid configurations in which dependencies between some configuration options – which are even formulated in KCONFIG – are not met correctly. Among other issues with the handling of recursive dependencies in KCONFIG, this problematic behaviour is one argument for an integration of a SAT solver directly into KCONFIG itself, as recently proposed by kernel developers and researchers [20].

6. RELATED WORK

Many researchers are working on modelling and analyzing variability, often in the context of the Linux kernel. In the following, we would like to highlight some of these works and relate them to our contribution.

In previous work [25], our group presented VAMPYR, a tool that generates a set of configurations that must be applied and checked by existing static checkers in order to maximize the *configuration coverage*. VAMPYR, however, requires the set of `.c` files modified by a patch as input; by design, it can not track the influence of modified configuration options on `#ifdef` blocks inside unmodified files. This can lead to false negatives, as not all relevant files will be checked.

In a detailed analysis of the `driver` subsystem of Linux, Passos et al. [18] study the scattering of feature code. Their findings suggest that driver features are often scattered across several locations in the code. The effects of such feature scattering on code quality – and thus, the likelihood of defects in scattered code – are studied by Eaddy et al. [6]. In their work, the researches find evidence for a correlation between high scattering degrees and the number of bugs in the scattered code; as many multiple locations might be of concern for the same feature, it gets harder to fully understand all possible interactions between the scattered parts of the code, in turn making it difficult to correctly and consistently change the source code. These observations underline the importance of our approach which allows us to find scattered locations possibly affected by a change in a single feature (or configuration option) as well as the influence on other configuration options – which in turn might be scattered. Furthermore, Queiroz et al. [19] report that the degree of scattering in Linux follows a power-law distribution, which aligns with the shape of our result plots in Section 4.

In order to recover relationships between configuration options, She et al. [23] present heuristics to find parent features from the variability model in order to reconstruct the feature hierarchy. Their evaluation shows that features in Linux have complicated dependencies, with most dependencies coming from transitive implications. This confirms the importance of taking transitive `select`s and dependencies into account, which is covered by the design of our worklist algorithm.

[3]Located at `Documentation/kbuild/kconfig-language.txt`

Our algorithm relies on an accurate representation of the configurable options and their dependencies. As the semantics of the KCONFIG language are sometimes unclear, El-Sharkawy, Krafczyk, and Schmid [7] systematically evaluate the behaviour of all available extractors and show that none of the generated models fully represents the actual behaviour of KCONFIG. For this paper, we consider the adverse effects of our extractor's insufficiencies as not practically relevant since they mostly affect corner cases in choices – which are rarely used and changed in Linux anyway [4]. However, our design would also allow an easy substitution of the extractor with more accurate tools like KCONFIGREADER[10] by Kästner et al.

Our approach aims at the detection of files which are affected by changes to configuration options – hence, to apply our algorithm, we first need to know if and which configuration options have changed compared to an earlier state. For this purpose, Dintzner, Van Deursen, and Pinzger [5] present `FMDiff`, a tool which can efficiently compute the changes between two KCONFIG feature models and categorize the types of changes. We plan to integrate `FMDiff` into our daily tests of `linux-next` in the future.

7. CONCLUSION AND FUTURE WORK

As Linux is a large and rapidly evolving software system, its developers require tool support in order to verify the integrity of the system as a whole. Companies and developers already have some automated testing systems in place which can detect build errors and performance regressions. Current systems, however, do not incorporate variability-related tests – errors caused by inconsistencies in the configuration system KCONFIG are only found "by accident" when compiling randomly generated kernel configurations. Furthermore, developers currently lack a mechanism to find out what configuration options are needed to run a complete set of tests [2].

In order to better understand the effects of configurable options on the source code, this paper presents a method which allows us to quickly compute relationships between configuration options from a feature model. Additionally, we show how the relationship information can be used to find the set of files which might be influenced by a given configuration option, while also taking the `#include` structure into account.

Our results show that many configuration options only affect few others, but also that some configuration options have up to 8,000 options related to them, and might affect up to 21,600 files. We also found that some configuration options misuse KCONFIGS `select` feature for usability reasons, leading to these options being associated with many others by our algorithm.

In the future, we will integrate the presented approach into our daily bot which searches for variability defects in `linux-next` in order to possibly uncover more complicated interactions between configuration options and their use in the source code. Additionally, we would like to use the data to optimize the use of configurations during testing: as we now know which files are affected by a change in Kconfig, we can analyze all these files for uses of configurable options and subsequently generate a minimal set of configurations in order to maximize configuration coverage inside the affected files. In the long term, this might eliminate the need for randomly generated configurations for testing and replace them with configurations tailored to the change in question.

Furthermore, we would like to investigate the relationships between header files and source files in a more fine-grained manner; for example, we could evaluate which definitions in a header file really are *used* by the code. This would allow us to reduce overapproximation, as we can then only mark source files as affected that actually *use* some `#ifdef`-guarded functionality provided by the header.

> "There is little disagreement about the need for better tests – and the need for developers to actually run those tests. This is an area that should continue to progress quickly in the coming year." [2]
>
> (Jonathan Corbet, Linux kernel developer)

References

[1] Thorsten Berger, Steven She, Rafael Lotufo, and Andrzej Wasowski und Krzysztof Czarnecki. "Variability Modeling in the Real: A Perspective from the Operating Systems Domain". In: *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. 2010.

[2] Jonathan Corbet. *Kernel testing session at Kernel Summit 2015*. 2015. URL: https://lwn.net/Articles/662882/ (visited on 11/05/2015).

[3] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "A Robust Approach for Variability Extraction from the Linux Build System". In: *Proceedings of the 16th International Software Product Line Conference (SPLC '12)*. 2012.

[4] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. "Analysing the Linux kernel feature model changes using FMDiff". In: *Software & Systems Modeling* (2015).

[5] Nicolas Dintzner, Arie Van Deursen, and Martin Pinzger. "Extracting feature model changes from the Linux kernel using FMDiff". In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS '14)*. 2014.

[6] Marc Eaddy, Thomas Zimmermann, Kaitlin D Sherwood, Vibhav Garg, Gail C Murphy, Nachiappan Nagappan, and Alfred V Aho. "Do crosscutting concerns cause defects?" In: *IEEE Transactions on Software Engineering* 34.4 (2008).

[7] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. "Analysing the Kconfig Semantics and Its Analysis Tools". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '15)*. 2015.

[8] Paul Gazzillo and Robert Grimm. "SuperC: parsing all of C by taming the preprocessor". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. (Beijing, China). 2012.

[9] Shelby Ineson. *0-Day Linux Kernel Testing: The Inside Story of an Intel Software Quality Award Winner*. 2015. URL: https://01.org/lkp/news/0-day-linux-kernel-testing-team-won-2015-intel-software-quality-awards (visited on 11/05/2015).

[10] Christian Kästner. *kconfigreader*. 2015. URL: https://github.com/ckaestne/kconfigreader (visited on 11/05/2015).

[11] Christian Kästner, Paolo G. Giarrusso, and Klaus Ostermann. "Partial Preprocessing C Code for Variability Analysis". In: *Proceedings of the 5th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '11)*. Jan. 2011.

[12] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. "Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation". In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*. Oct. 2011.

[13] Michael Kerrisk. *KS2012: Kernel build/boot testing*. 2012. URL: https://lwn.net/Articles/514278/ (visited on 11/05/2015).

[14] Greg Kroah-Hartman. *Linux kernel history logs and stats*. 2015. URL: https://github.com/gregkh/kernel-history (visited on 11/05/2015).

[15] Christopher Li. *Sparse - a Semantic Parser for C*. 2015. URL: https://sparse.wiki.kernel.org (visited on 11/05/2015).

[16] Jörg Liebig, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. "Scalable Analysis of Variable Software". In: *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '13)*. Aug. 2013.

[17] Gilles Muller. *Coccinelle: A Program Matching and Transformation Tool for Systems Code*. 2015. URL: http://coccinelle.lip6.fr/ (visited on 11/05/2015).

[18] Leonardo Passos, Jesús Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Tulio Valente. "Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers". In: *Proceedings of the 14th International Conference on Modularity (MODULARITY '15)*. 2015.

[19] Rodrigo Queiroz, Leonardo Passos, Marco Tulio Valente, Sven Apel, and Krzysztof Czarnecki. "Does Feature Scattering Follow Power-law Distributions?: An Investigation of Five Pre-processor-based Systems". In: *Proceedings of the 6th International Workshop on Feature-Oriented Software Development (FOSD '14)*. 2014.

[20] Luis R. Rodriguez. *Linux Kconfig SAT integration*. 2015. URL: http://kernelnewbies.org/KernelProjects/kconfig-sat (visited on 11/05/2015).

[21] Andreas Ruprecht. "Efficient, Yet Robust Extraction of Variability Information from Linux Makefiles". FOSD Meeting '15. 2015. URL: http://mevss.jku.at/wp-content/uploads/2014/08/Andreas-Ruprecht-Efficient-Yet-Robust-Extraction-of-Variability-Information-from-Linux-Makefiles.pdf.

[22] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. "The Variability Model of the Linux Kernel". In: *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '10)*. Jan. 2010.

[23] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wøsowski, and Krzysztof Czarnecki. "Reverse engineering feature models". In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. 2011.

[24] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem". In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys '11)*. Apr. 2011.

[25] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue". In: *Proceedings of the 2014 USENIX Technical Conference (USENIX '14)*. 2014.

[26] Unisys Corporation. *Secure Partitioning (s-Par) for Enterprise-Class Consolidation*. Tech. rep. Feb. 2014. URL: http://www.unisys.co.nz/offerings/security-solutions/Whitepaper/Secure-Partitioning-sPar-for-Enterprise-Class-Consolidation-id-637 (visited on 11/05/2015).