Finding Broken Linux Configuration Specifications by Statically Analyzing the Kconfig Language

Jeho Oh*
University of Texas at Austin
jeho.oh@utexas.edu

Necip Fazıl Yıldıran*
University of Central Florida
yildiran@knights.ucf.edu

Julian Braha
University of Central Florida
julianbraha@knights.ucf.edu

Paul Gazzillo
University of Central Florida
paul.gazzillo@ucf.edu

ABSTRACT

Highly-configurable software underpins much of our computing infrastructure. It enables extensive reuse, but opens the door to broken configuration specifications. The configuration specification language, Kconfig, is designed to prevent invalid configurations of the Linux kernel from being built. However, the astronomical size of the configuration space for Linux makes finding specification bugs difficult by hand or with random testing. In this paper, we introduce a software model checking framework for building Kconfig static analysis tools. We develop a formal semantics of the Kconfig language and leverage automated theorem proving to model Kconfig behavior and find bugs. We design and implement a bug finder for unmet dependencies called `kismet`. `kismet` is evaluated for its precision, performance, and impact on kernel development for a recent version of Linux, which has over 140,000 lines of Kconfig across 28 architecture-specific specifications. Our evaluation finds 781 bugs (151 when considering sharing among Kconfig specifications) with 100% precision, spending between 37 and 90 minutes for each Kconfig specification, although it misses some bugs due to underapproximation. Compared to random testing, `kismet` finds substantially more true positive bugs in a fraction of the time.

KEYWORDS

software configuration, Kconfig, formal verification, static analysis

1 INTRODUCTION

Highly-configurable software product lines underpin much of our computing infrastructure, because configurability enables reuse without having to reprogram the software for new devices or applications. The Linux kernel is one such example of a software product line, which is used in billions of computing devices. With over 15,000 configuration options controlling everything from drivers, architecture, memory management, and more, there over than $2^{15{,}000}$ combinations, if only considering Boolean options. This extreme configurability makes its widespread use possible, but also opens the door to invalid configurations, which produce broken variations of the software.

To mitigate the chance of misconfiguration, developers provide *configuration specifications*, which can define the intended combinations of configuration options. These specifications may be as simple as a text file describing configuration instructions, or as sophisticated as a machine-readable specification enforced at build time. These specifications, if only implicitly, define a software product line's *feature model*, i.e., the legal configurations of the software. In Linux and other systems software, such as BusyBox and coreboot, developers use the Kconfig language to specify configuration options, as well as their dependencies.

While Linux is effectively a software product line, its specification language, Kconfig, is unlike typical feature modeling languages [45]. Kconfig has complex semantics and additional language features, such as invisible variables, automated option selection, and user-interface support. With over 140,000 lines of Kconfig specifications in the Linux kernel, its complex behavior makes maintenance challenging. One example is the common pitfall highlighted in Kconfig's manual [29], the *unmet dependency bug*. These bugs lead to illegal configurations when developers unwittingly mix conflicting constructs in dependency specifications. With thousands of potentially vulnerable constructs and an ever-changing specification, finding such bugs by hand is a practically impossible task.

Existing work on the analysis of Kconfig is focused on extracting a feature model, rather than checking for Kconfig bugs. Having a Linux feature model has been useful for applications outside of Kconfig, including measuring feature model hierarchies [10, 40, 43], supporting variability-aware analysis of C [11, 19, 21, 25, 27, 28, 32, 49, 54, 56], and dead code elimination [50, 51]. However, these tools make assumptions about Kconfig semantics that, while appropriate for their respective applications, make them less amenable to bug finding. For instance, KconfigReader explicitly omits modeling the language semantics that lead to unmet dependencies, leaving a checker as a separate problem [26, 31]. The other tools do the same and have less support for Kconfig semantics, omitting some parts of the Linux feature model altogether [17, 40]. Moreover, by focusing on feature modeling, prior tools bake in decisions about the analysis domain, i.e., a feature model, which limits the feasibility of repurposing the work for Kconfig bug finding and other source level tools.

In this paper, **we introduce a software model checking framework for building Kconfig static analysis tools**. Inspired by model checking frameworks for program code [1], we base our static

*Co-first authors.

https://doi.org/10.1145/nnnnnnn.nnnnnnn

analysis on our newly-developed formal semantics of the Kconfig language and leverage automated theorem proving to model Kconfig behavior and find bugs. Of existing descriptions of Kconfig semantics, all but one are either informal or example-based [16, 17] which, having no formalization, are not amenable to automated reasoning. The one prior formal semantics uses an idealized Kconfig syntax rather than Kconfig's actual grammar, is missing language behavior (including that leading to unmet dependencies), and uses an abstract domain designed for feature modeling [42]. In contrast, we develop a formal semantics of the concrete behavior of Kconfig when it checks a configuration file's validity. We define our semantics over the syntax derived from Kconfig's actual implementation, which contains a `bison` grammar specification.

This approach to Kconfig semantics has three key benefits over prior efforts. First, it is formal, making it possible to use automated reasoning tools. Second, it is concrete, which decouples the description of Kconfig semantics from any particular analysis objective. This leaves decisions about how to abstract Kconfig behavior to specific applications and should reduce future effort to design new Kconfig analyses. Third, it simplifies modeling Kconfig since, as we show, we can methodically derive an abstraction of Kconfig behavior from this concrete semantics.

To demonstrate the utility of our approach, we design and implement an analysis that finds the unmet dependency bugs highlighted in Kconfig's manual and is, to our knowledge, the first static analysis for finding such bugs. We first define the bug as a formal property in terms of the semantics, then show how a checker can be methodically derived from the semantics. We underapproximate non-Boolean options and use aggressive optimization to yield a bug-finder that is both fast and very precise. Moreover, it can also automatically localize and generate test cases for the unmet dependency bugs it finds. The trade-off is decreased recall due to false negatives, although we show that these are less common due to the rarity of non-Boolean options.

We implement the bug-finder and evaluate it on a recent version of the Linux kernel source, which contains over 140,000 lines of Kconfig describing 28 architecture-specific Kconfig specifications. Our results show that our bug finding is both precise and fast. The bug-finder finds 781 alarms (151 when considering sharing among Kconfig specifications) over all Linux kernel architectures' Kconfig specifications, all of which are verified true positives, for a precision of 100%. While such precision might be unusually high for a programming language analyzer, the Kconfig language has no iteration or recursion that would require overapproximation. With our optimizations, our bug-finder takes an average of 40 minutes for one Kconfig specification, checking over 10,000 constructs.

While we are still in the process of reporting all bugs found by our tool, **Linux maintainers have so far already confirmed 38 of our reports and committed 15 of our patches into the mainline Linux kernel repository**, demonstrating the utility of our tooling. Committing patches is a manual process, requiring discussion with kernel maintainers, so we are still investigating, reporting, and submitting patches for the remaining alarms.

Since, to our knowledge, no other static bug finder for unmet dependencies exists, we compare against random testing using Kconfig's built-in random configuration generator frequently used in kernel testing [5]. Given the same amount of time to find bugs, random testing yields only 98 alarms compared to our tool's 781. Even letting random testing run for over 4 days for each Kconfig specification, 135 times longer than our tool's total time, the testing approach still only finds 175 bugs, far fewer than our tool. The random testing approach did find eight bugs missed by our tool, reflecting the tradeoff in performance gained by underapproximation. Even with this limitation, our tool finds many more bugs, while taking far less time, a useful complement to testing.

In summary, this paper makes the following contributions:

- A formal semantics of Kconfig's concrete behavior (Section 3).
- An efficient design of a bug-finder and localizer for unmet dependency bugs (Section 4).
- An implementation of the bug finder, along with reusable components for creating Kconfig analyzers (Section 5).
- An experimental evaluation of the bug finder's precision, performance, and impact (Section 6).

2 OVERVIEW

In this section we introduce the Kconfig language, illustrate an unmet dependency bug, and summarize how our formal semantics enables the design of a static analysis to find such bugs.

2.1 Introduction to the Kconfig Language

Figure 1 is a simplified snippet of Kconfig from Linux v5.4.4. Configuration options are defined with the `config` construct (lines 1, 7, and 12). Inside each `config` declaration is a block of constructs that define the option's type (e.g. Boolean), constraints on its use, and text used by Kconfig to populate a user interface.

Lines 2, 8, and 13 are the type declarations. A `bool` option (line 8) has two possible settings, `y` or `n`. `y` means the feature is on and compiled into the kernel, and `n` means the feature is off and omitted from the kernel. A `tristate` option (lines 2 and 13) adds an additional setting, `m`. `m` is like `y` except that the build system compiles a loadable kernel module instead of linking to the main kernel binary [29].

`tristate` and `bool` are the most common configuration options, representing more than 95% of options in the Linux Kconfig specifications. The other possible types are `string` for strings, `int` for decimal integers, and `hex` for hexadecimal numbers.

Constraints on options are defined using `depends on` (lines 4, 10, and 15) and `select` (line 5), but Kconfig language prohibits circular dependencies. `depends on` defines a *direct dependency*, which provides requirements that should hold before the option can be enabled. The dependency is expressed with a Boolean expression of other options. For instance, line 4 means that `TOUCHSCREEN_ADC` may not be enabled unless `IIO && INPUT_TOUCHSCREEN` is true, i.e., when both `IIO` and `INPUT_TOUCHSCREEN` are also enabled.

A *reverse dependency*, given by the `select` construct, inverts the dependency relationship by forcing the target of the select to be enabled. For instance, line 5 means that whenever `TOUCHSCREEN_ADC` is enabled, `IIO_BUFFER_CB` is forced to be enabled. A reverse dependency can only enable another option, not disable it, and only applies to `bool` or `tristate` options. *Kconfig permits reverse dependencies to override direct dependencies*, which can lead to unmet dependency bugs.

```
1  config TOUCHSCREEN_ADC
2          tristate
3          prompt "Generic ADC based touchscreen"
4          depends on IIO && INPUT_TOUCHSCREEN
5          select IIO_BUFFER_CB
6
7  config IIO_BUFFER
8          bool
9          prompt "Enable buffer support within IIO"
10         depends on IIO
11
12 config IIO_BUFFER_CB
13         tristate
14         prompt "IIO callback buffer"
15         depends on IIO && IIO_BUFFER
```

Figure 1: An example Kconfig specification that allows an unmet dependency violation and leads to a build error. Adapted from Linux source: drivers/input/touchscreen/Kconfig, drivers/iio/Kconfig, and drivers/iio/buffer/Kconfig.

Options with a prompt are *visible* options that a user may enable. The `prompt` construct defines the prompt string for use in a user interface (lines 3, 9, and 14). Non-visible options, i.e., those with no prompt construct, can only be set by a `select` construct or take a specification-defined default value (not shown in this example). The visibility of an option affects the behavior of a `config` construct in subtle ways, which we describe in the formal semantics of Kconfig.

An Unmet Dependency Bug in the Wild. Figure 1 has an unmet dependency bug found by our automated analysis. All three of the configuration options defined in this example control specific C compilation units that are only built into the kernel when the options are enabled. `IIO_BUFFER_CB` controls `industrialio-buffer-cb.o` and `IIO_BUFFER` control `industrialio-buffer.o`.

`industrialio-buffer-cb.o` calls functions that are defined in `industrialio-buffer.o`, so the former cannot be built without the latter, otherwise there would be a build error. The developers capture this build dependency with a direct dependency in the definition of the `IIO_BUFFER_CB` option (line 15). This constraint, by itself, would prevent a user from giving a configuration that leads to the build error.

The `select IIO_BUFFER_CB` construct on line 5, however, can override this direct dependency under certain conditions. Specifically, if a user (or another `select`) enables `TOUCHSCREEN_ADC`, the `select` automatically force-enables `IIO_BUFFER_CB`. Kconfig permits such a configuration to proceed to build, albeit with a warning. Still, the build will fail, and the user will have to manually correct their configuration file in order to avoid the unmet dependency.

While the Kconfig documentation warns of `select`'s pitfalls and recommends not using it to override dependencies, it is difficult to check by hand whether any of its 17,000+ uses have an unmet dependency bug.

2.2 An Unmet Dependencies Bug Finder

We create a formal model of the unmet dependency bug according to the semantics of Kconfig.

We tackle the problem of finding unmet dependencies in Kconfig with formal verification. First, we model the space of valid Kconfig configurations in formal logic automatically with our symbolic evaluator `kclause`. Next, `kismet` generates verification conditions to prove the absence of an unmet dependency for each `select` construct in the Kconfig specification. Not all reverse dependencies can cause unmet dependencies, so `kismet` needs to consider constraints from all configuration options to rule out infeasible ones.

The resulting verification conditions are discharged to the Z3 SMT solver [14]. When an unmet dependency cannot be ruled out, `kismet` raises an alarm. It then switches to test case generation, converting any counterexamples to Linux configuration files. `kismet` uses these tests on Kconfig and the build system to expose real bugs.

To see how `kclause` models dependencies, take Figure 1's definition of `TOUCHSCREEN_ADC` (line 1). Since it has no reverse dependencies, it can only be enabled when its direct dependencies hold, i.e., enabling `TOUCHSCREEN_ADC` implies that `IIO` and `INPUT_TOUCHSCREEN` are also both enabled:

$$\texttt{TOUCHSCREEN\_ADC} \longrightarrow \texttt{IIO} \wedge \texttt{INPUT\_TOUCHSCREEN}$$

When an option has reverse dependencies, its direct dependencies do not have to hold if its reverse dependencies already do. For instance, enabling `IIO_BUFFER_CB` (line 12) implies that its direct *or* reverse dependencies hold:

$$\texttt{IIO\_BUFFER\_CB} \longrightarrow (\texttt{IIO} \wedge \texttt{IIO\_BUFFER} \vee \texttt{TOUCHSCREEN\_ADC})$$

An unmet dependency happens when an option's reverse dependencies hold but its direct dependencies do not. For instance, an unmet dependency happens when `TOUCHSCREEN_ADC` force-enables `IIO_BUFFER_CB` even though `IIO_BUFFER_CB`'s direct dependencies are infeasible. This unmet dependency can be formalized as follows:

$$\texttt{TOUCHSCREEN\_ADC} \wedge (\texttt{IIO} \wedge \texttt{INPUT\_TOUCHSCREEN}) \wedge \texttt{IIO\_BUFFER\_CB} \wedge \neg(\texttt{IIO} \wedge \texttt{IIO\_BUFFER})$$

`kismet` tries to prove the *negation* of this condition, since it verifies the absence of unmet dependencies. If the proof fails, `kismet` raises an alarm and switches to test case generation.

3 THE SEMANTICS OF KCONFIG

The Kconfig language is a declarative configuration specification language. At its core, Kconfig takes a configuration file, which is a mapping from configuration options to their concrete values, and determines whether the configuration file is valid according to the developer's specifications. Developers use Kconfig language constructs to define configuration options, declaring their names, types, and any dependencies they have on other configuration options. Kconfig also supports user interfaces, and the language has additional constructs, such as `help`, to specify text elements of the interface. These do not affect the buildability of configuration files and act as comments.

We developed this formal semantics from reading the Kconfig manual, inspecting its source code, and informal descriptions and

```
kconfig    ::= statement+
statement  ::= config | choice
config     ::= config symbol type constrnts select*
choice     ::= choice type constrnts config+ endchoice
type       ::= bool | tristate
constrnts  ::= prompt depends+ default+
prompt     ::= prompt word if expr
default    ::= default val if expr
depends    ::= depends on expr
select     ::= select symbol if expr
expr       ::= expr && expr | expr || expr | ! expr | symbol
val        ::= y | n
```

Figure 2: Formal syntax of a core fragment of Kconfig.

examples from prior work [16, 17]. To check the fidelity of the semantics, we used new and existing benchmarks [16, 17], generated random configuration files fed to Kconfig as input, and evaluated this paper's bug-finder, which requires a correct semantics for its analysis to be precise. Given the size of the semantics, having dozens of rules and still more syntactic sugar rules, we highlight a core fragment of the language here, and document the remaining rules in the supplemental material[1].

3.1 Configuration Declarations

Figure 2 shows the syntax of a core fragment of the Kconfig language for `bool` configuration options. A *kconfig* file contains a list of *statement*s, which are either a configuration option declaration or a choice construct. A configuration option, *config*, has a *type*, constraints for direct dependencies, and zero or more *select* constructs for reverse dependencies.

Figure 3a defines the semantic valuation function S for statements. S functions take an immutable configuration file σ as input and return whether the configuration is `valid` or `invalid`, i.e., buildable or not. S_1 evaluates a Kconfig specification's list of statements by checking whether all statements are valid according to the input.

S_2 is the valuation function for `config` statements. The number of cases reflects the complexities of Kconfig's validity checking. The first covers reverse dependencies, using the valuation function R, which searches the entire *kconfig* file. (In practice, the Kconfig implementation memoizes reverse dependencies during parsing to avoid repeatedly traversing the syntax tree.) If a reverse dependency holds, that means the option must be enabled, i.e., $\sigma(sym) = y$, otherwise the configuration file does not match the specification.

The second case of S_2 handles a direct dependency when the reverse dependency does not hold. In this case, an option is valid regardless of its setting, because a user is free to enable or disable it. The third case covers non-visible configuration options, which have no `prompt`, so the option's value must match the specified default. The fourth case covers when none of the option's dependencies hold. Lastly, if none of these conditions are met, the configuration file is not valid.

[1]A blinded version is included as supplementary material with the submission.

Dependencies for non-Boolean types (string, int, and hex) behave similarly to `bool` and tristate, but there are additional constraints and expressions such as `range` and inequalities. The full semantics in the supplemental material describes these differences.

3.2 Reverse Dependencies

To find any reverse dependencies for an option, Kconfig needs to search the entire specification for a `select` that can enable the option. This is partly why tracking down unmet dependencies is so difficult.

Figure 3b defines the valuation function R for reverse dependencies. It takes both the configuration file σ and a symbol name s and returns a Boolean value: `true` if that symbol is selected by some option or `false` if not. R_1's disjunction reflects the need for only one `select` to force-enable an option.

R_2 checks to see if an option is enabled and its dependencies are met, then calls R_3 to evaluate any select constructs. R_3 checks whether there is a select for the input symbol s. R_4 checks whether any configuration option within a `choice` block selects symbol s.

Options other than `tristate` and `bool` cannot be the selector or selectee of a reverse dependency.

3.3 Choice Constructs

A `choice` construct defines a mutually-exclusive set of configuration options. Choices are useful in configuration specifications, because expressing them with Boolean logic alone is verbose. Figure 4 is an example of a `choice` from the Linux kernel that allows only one of several compression algorithms for a file system be enabled. A choice block starts with a `choice` keyword (line 1) and ends with an `endchoice` (line 10). It contains a list of configuration options which, besides the mutual-exclusion rule, behave mostly like any other options, except that they cannot have reverse dependencies or default values.

The S_3 function in Figure 3a describes the `choice` block's semantics. The first case covers the mutual exclusion property, requiring that only one of the configuration options is enabled. This condition also recursively checks that the all nested `config`s' dependencies are valid.

Choice constructs also have their own direct dependencies, so Kconfig permits no options to be enabled when the choice dependencies are not met. The second case of S_3 covers this situation. The third case covers the situation when none of the nested `config` options' dependencies are met, in which case Kconfig also permits the choice to have no options enabled. `choice` constructs can also take the `optional` keyword to allow for no options to be enabled even if its direct dependencies are met. The rules are slightly different from a regular choice, and we present them in the supplemental material.

The `choice` statement described above has `bool` type. The only other type a choice can take is `tristate`, in which case its behavior is the same as `bool`, except that more than one choice may be set to `m`.

3.4 Constraint Expressions

Figure 3e defines rules for evaluating constraint expressions, which return a Boolean `true` or `false`. E_1, E_2, and E_3 are the `prompt`,

$$(S_1)\quad \mathrm{S}[[kconfig]]\sigma \triangleq \begin{cases} \texttt{valid} & \textbf{if } \bigwedge_{statement_i \in kconfig} (\mathrm{S}[[statement_i]]\sigma = \texttt{valid}) \\ \texttt{invalid} & \text{otherwise} \end{cases}$$

$$(S_2)\quad \mathrm{S}[[\textbf{config } sym \textbf{ bool } constrnts\ select^*]]\sigma \triangleq \begin{cases} \texttt{valid} & \textbf{if } (\sigma(sym) = \texttt{y}) \wedge \mathrm{R}[[kconfig]](\sigma, sym) \\ \texttt{valid} & \textbf{if } \mathrm{E}[[depends+]]\sigma \wedge \mathrm{E}[[prompt]]\sigma \wedge \neg\, \mathrm{R}[[kconfig]](\sigma, sym) \\ \texttt{valid} & \textbf{if } ((\sigma(sym) = \texttt{y}) \wedge \mathrm{E}[[default+]]\sigma \vee (\sigma(sym) = \texttt{n}) \wedge \neg \mathrm{E}[[default+]]\sigma) \\ & \quad \wedge\, \mathrm{E}[[depends+]]\sigma \wedge \neg \mathrm{E}[[prompt]]\sigma \wedge \neg\, \mathrm{R}[[kconfig]](\sigma, sym) \\ \texttt{valid} & \textbf{if } (\sigma(sym) = \texttt{n}) \wedge \neg \mathrm{E}[[depends+]]\sigma \wedge \neg\, \mathrm{R}[[kconfig]](\sigma, sym) \\ \texttt{invalid} & \text{otherwise} \end{cases}$$

$$(S_3)\quad \mathrm{S}[[\textbf{choice bool } constrnts\ config+ \textbf{ end}]]\sigma \triangleq \begin{cases} \texttt{valid} & \textbf{if } (\textsc{Enabled}[[config+]]\sigma = 1) \wedge (\mathrm{S}[[config+]]\sigma = \texttt{valid}) \\ & \quad \wedge\, \mathrm{E}[[depends+]]\sigma \wedge \mathrm{E}[[prompt]]\sigma \\ \texttt{valid} & \textbf{if } (\textsc{Enabled}[[config+]]\sigma = 0) \wedge \neg(\mathrm{E}[[depends+]]\sigma \wedge \mathrm{E}[[prompt]]\sigma) \\ \texttt{valid} & \textbf{if } (\textsc{Enabled}[[config+]]\sigma = 0) \wedge \bigwedge_{constrnts_i \in config+} (\neg \mathrm{E}[[constrnts_i]]\sigma) \\ \texttt{invalid} & \text{otherwise} \end{cases}$$

(a) Direct dependency rules.

$$(R_1)\quad \mathrm{R}[[kconfig]](\sigma, s) \triangleq \bigvee_{statement_i \in kconfig} \mathrm{R}[[statement_i]](\sigma, s)$$

$$(R_2)\quad \mathrm{R}[[\textbf{config } sym \textbf{ bool } constrnts\ select^*]](\sigma, s) \triangleq \begin{cases} \mathrm{R}[[select^*]](\sigma, s) & \textbf{if } (\sigma(sym) = \texttt{y}) \wedge \mathrm{E}[[depends+]]\sigma \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$(R_3)\quad \mathrm{R}[[\textbf{select } sym \textbf{ if } expr\ select^*]](\sigma, s) \triangleq \begin{cases} \texttt{true} & \textbf{if } (sym = s) \wedge \mathrm{E}[[expr]]\sigma \\ \mathrm{R}[[select^*]](\sigma, s) & \textbf{if } select^* \neq \emptyset \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$(R_4)\quad \mathrm{R}[[\textbf{choice bool } constrnts\ config+ \textbf{ end}]](\sigma, s) \triangleq \begin{cases} \mathrm{R}[[config+]](\sigma, s) & \textbf{if } \mathrm{E}[[depends+]]\sigma \wedge \mathrm{E}[[prompt]]\sigma \\ \texttt{false} & \text{otherwise} \end{cases}$$

(b) Reverse dependency rules.

Σ : *Symbols* $\rightarrow \{\texttt{y}, \texttt{n}\}$
S : *Statements* $\rightarrow (\Sigma \rightarrow \{\texttt{valid}, \texttt{invalid}\})$
R : *Statements* $\rightarrow (\Sigma \times \mathit{Symbols} \rightarrow \{\texttt{true}, \texttt{false}\})$
E : *Constraints* $\rightarrow (\Sigma \rightarrow \{\texttt{true}, \texttt{false}\})$

(c) Types for input (Σ) and the valuation functions.

$$\textsc{Enabled}[[config+]]\sigma \triangleq \sum_{\substack{sym_i \in config+ \\ \sigma(sym_i) = y}} (1)$$

(d) Counting how many config options are enabled.

$$(E_1)\quad \mathrm{E}[[\textbf{prompt } word \textbf{ if } expr]]\sigma \triangleq \mathrm{E}[[expr]]\sigma$$

$$(E_2)\quad \mathrm{E}[[depends+]]\sigma \triangleq \bigwedge_{expr_i \in\, depends+} (\mathrm{E}[[expr_i]]\sigma)$$

$$(E_3)\quad \mathrm{E}[[\textbf{default } val \textbf{ if } expr\ default^*]]\sigma \triangleq \begin{cases} val = \texttt{y} & \textbf{if } \mathrm{E}[[expr]]\sigma \\ \mathrm{E}[[default^*]]\sigma & \textbf{if } default^* \neq \emptyset \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$(E_4)\quad \mathrm{E}[[expr_1 \texttt{ \&\& } expr_2]]\sigma \triangleq \mathrm{E}[[expr_1]]\sigma \wedge \mathrm{E}[[expr_2]]\sigma$$

$$(E_5)\quad \mathrm{E}[[expr_1 \texttt{ || } expr_2]]\sigma \triangleq \mathrm{E}[[expr_1]]\sigma \vee \mathrm{E}[[expr_2]]\sigma$$

$$(E_6)\quad \mathrm{E}[[\texttt{ ! } expr]]\sigma \triangleq \neg \mathrm{E}[[expr]]\sigma$$

$$(E_7)\quad \mathrm{E}[[sym]]\sigma \triangleq (\sigma(sym) = y)$$

(e) Expression evaluation rules.

Figure 3: Formal semantics of a core fragment of Kconfig.

```
1  choice
2      prompt "Decompressor parallelisation options"
3      depends on SQUASHFS
4  config SQUASHFS_DECOMP_SINGLE
5      bool "Single threaded compression"
6  config SQUASHFS_DECOMP_MULTI
7      bool "Use multiple decompressors"
8  config SQUASHFS_DECOMP_MULTI_PERCPU
9      bool "Use percpu multiple decompressors"
10 endchoice
```

Figure 4: An example of a choice construct.

depends on, and default constructs, respectively. Each is a carrier for a logical expression, and it is their interaction with config and choice that gives them distinct meaning. The rest of the rules are typical Boolean operators (E_4–E_7).

3.5 Syntactic Sugar

Kconfig has three additional statements that can be desugared to config and choice: if, menu, and menuconfig. Unlike the control-flow construct in programming languages, Kconfig's if is just syntactic sugar for adding constraints in bulk to its nested statements. The menu statement behaves like an if block, but also adds text to the user interface. menuconfig is a combination of config and menu. The supplemental material contains the desugaring rules for these.

The Kconfig language also has a great deal of flexibility in its syntax. Most of the behavior of a Kconfig specification is insensitive to the ordering of options and constraints. Therefore, our syntax defines ordering on constraints to reduce the number of syntactic sugars rules needed.

Kconfig has limited metaprogramming facilities via preprocessor constructs for file inclusion and macro expansion [2], which we do not model. Our implementation runs the preprocessor before symbolic evaluation to ensure that all files are included and macros are expanded.

4 DESIGNING THE BUG FINDER

Our bug-finder, called kismet works by generating a formula for each select describing the configurations under which it triggers an unmet dependency bug. This requires both syntax analysis, to identify select constructs, as well as semantic analysis, to construct a formal model of the bug automatically. kismet discharges the formal conditions to an SMT solver to check satisfiability. The main challenge to designing kismet are ensuring both scalability while preserving enough precision to identify the exact constructs causing the unmet dependency alarm.

4.1 Identifying Select Constructs

The first challenge for kismet is to identify select constructs in the Kconfig specifications. Walking over each config construct syntactically, it records all pairs of options involved in a select operation. For instance, in Figure 1, kismet identifies the pair (TOUCHSCREEN_ADC, IIO_BUFFER_CB) which contains the *selector* and *selectee*, respectively. In order to verify whether the select is free from an unmet dependency bug, kismet needs to account for all of the dependencies that constrain both the selector and the selectee.

```
1  config X
2          select A if D_X
3          K_X // other constraints for X
4
5  config A
6          depends on D_A
7          K_A // other constraints for A
8  K_other // constraints from other configuration options
```

Figure 5: Components of an unmet dependency condition.

The schematic in Figure 5 highlights what conditions kismet uses from the Kconfig specification to construct a verification condition. The configuration option X (line 1) selects A (line 5) with the select construct on line 2. The select construct itself is constrained by some if dependency, captured by a logical formula D_X (line 2). Additionally, X has its own dependencies controlling when it can be enabled (line 3). A's direct dependencies are D_A (line 6), while K_A (line 7) represents any prompt or default constraints. K_{other} represents the constraints from all other configuration options.

4.2 Modeling Unmet Dependency Bugs

X's select construct only causes an unmet dependency if the select overrides A's direct dependencies, i.e., when A's dependencies are unsatisfied. If we just consider the constraints of the selector and the selectee, the formula for unmet dependency is as follows:

$$\phi_{\text{unmet}} = X \wedge D_X \wedge K_X \tag{1}$$

$$\wedge\ A \wedge \neg(D_A \wedge K_A) \tag{2}$$

ϕ_{unmet} means the following: the *selector* option X is enabled and its select and other constraints $D_X \wedge K_X$ are met (subexpression 1); and the *selectee* option A is enabled while its dependencies $D_A \wedge K_A$ are *not* met (subexpression 2).

ϕ_{unmet} is an overapproximation, however, because it only accounts for the constraints from two configuration options, the selector and selectee. Constraints from other configuration options (K_{other}) can make ϕ_{unmet} unsatisfiable. Without accounting for those, we can expect more false positive alarms. A precise condition would contain these constraints as well:

$$\phi_{\text{unmet (precise)}} = \phi_{\text{unmet}} \wedge K_{\text{other}}$$

Optimizing the bug-finder. The Linux Kconfig specification has thousands of configuration options, so $\phi_{\text{unmet (precise)}}$ is a substantially more expensive formula to solve; it has the constraints from thousands of configuration options instead of just the two in ϕ_{unmet}. To make solving more efficient, we use two techniques. First, if a selectee option has no direct dependencies, then an unmet dependency bug is not possible. Second, we first check the ϕ_{unmet} condition and only check $\phi_{\text{unmet (precise)}}$ if the first check is satisfiable. This optimization is safe, because if ϕ_{unmet} is unsatisfiable, we

know that $\phi_{\text{unmet (precise)}}$ is also unsatisfiable, i.e. $\neg\phi_{\text{unmet}}$ entails $\neg\phi_{\text{unmet (precise)}}$:

$$\begin{aligned}
\phi_{\text{unmet}} &\longrightarrow \phi_{\text{unmet}} && \text{tautology}\\
\phi_{\text{unmet}} \wedge K_{\text{other}} &\longrightarrow \phi_{\text{unmet}} && \text{strengthening}\\
\phi_{\text{unmet (precise)}} &\longrightarrow \phi_{\text{unmet}} && \text{substitution}\\
\neg\phi_{\text{unmet}} &\longrightarrow \neg\phi_{\text{unmet (precise)}} && \text{contrapositive}
\end{aligned}$$

This simple optimization has a substantial impact on precision and performance as we show in the evaluation section.

4.3 Modeling Kconfig Semantics

Until now, we have described ϕ_{unmet} with placeholders for configuration option constraints. But interpreting these constraints as logical formulas requires modeling Kconfig's semantics. In this section we show how we methodically derive these from our formal semantics (Section 3).

Recall that Kconfig takes a configuration file as input and determines its validity according to the specifications. As with prior Kconfig feature modeling tools, we represent configuration options as symbolic Boolean options, collapsing `tristate` option's `y` and `m` to `true`. While this underapproximates `tristate`, it greatly reduces the space of possible configurations, improving solver performance. Similarly, we approximate non-Booleans with a finite range of options, as in prior work [31]. Less than 5% of options are non-Boolean in Linux Kconfig specifications.

To derive the model from Kconfig semantics, recall that our concrete semantics describes each case in which a Kconfig statement describes a valid configuration given an input configuration file. For each Kconfig syntactic construct in the specification under analysis, our bug finder automatically constructs a symbolic formula ϕ_i corresponding to its valuation function from the semantics in Figure 3. The formula is the disjunction of each condition leading to a `valid` result. For instance, a `config` statement, defined by semantic rule S_2 in Figure 3, has four `valid` cases. `kclause` constructs ϕ_{config} as a disjunction of each of these case conditions, where C is the symbolic value of the option:

$$\begin{aligned}
\phi_{\text{config}} = \quad & (C \wedge \phi_{\text{reverse}})\\
\vee\ & (\phi_{\text{depends}} \wedge \phi_{\text{prompt}} \wedge \neg\phi_{\text{reverse}})\\
\vee\ & ((C \wedge \phi_{\text{default}} \vee \neg C \wedge \neg\phi_{\text{default}})\\
& \quad \wedge \phi_{\text{depends}} \wedge \neg\phi_{\text{prompt}} \wedge \neg\phi_{\text{reverse}})\\
\vee\ & (\neg C \wedge \neg\phi_{\text{depends}} \wedge \neg\phi_{\text{reverse}})
\end{aligned}$$

Each of the four disjunctive terms corresponds to each of the four `valid` conditions from S_2. Accesses to the concrete configuration option value $\sigma(sym)$ are replaced with a symbolic Boolean value C. Calls to other valuation functions are replaced with the symbolic formulas for that syntax, e.g., ϕ_{depends} for the `depends on` construct.

For the configuration option `IIO_BUFFER_CB` in Figure 1, ϕ_{config} is constructed from the following symbolic formulas:

$$\begin{aligned}
C &= \texttt{IIO_BUFFER_CB}\\
\phi_{\text{reverse}} &= \texttt{TOUCHSCREEN_ADC} \wedge \texttt{IIO} \wedge \texttt{INPUT_TOUCHSCREEN}\\
\phi_{\text{depends}} &= \texttt{IIO} \wedge \texttt{IIO_BUFFER}\\
\phi_{\text{prompt}} &= \texttt{true}\\
\phi_{\text{default}} &= \texttt{false}
\end{aligned}$$

ϕ_{reverse} and ϕ_{direct} are the reverse and direct dependencies respectively. The option is always visible since it has an unconditional `prompt` (ϕ_{prompt}), and the default value is false since it has no `default` specification (ϕ_{default}).

Substituting the symbolic formulas into ϕ_{config} and simplifying the disjunctive terms, we get the following:

$$\begin{aligned}
& (\texttt{IIO_BUFFER_CB} && (3)\\
& \quad \wedge \texttt{TOUCHSCREEN_ADC} \wedge \texttt{IIO} \wedge \texttt{INPUT_TOUCHSCREEN})\\
\vee\ & (\texttt{IIO} \wedge \texttt{IIO_BUFFER} && (4)\\
& \quad \wedge \neg(\texttt{TOUCHSCREEN_ADC} \wedge \texttt{IIO} \wedge \texttt{INPUT_TOUCHSCREEN}))\\
\vee\ & (\texttt{false}) && (5)\\
\vee\ & (\neg\texttt{IIO_BUFFER_CCB} \wedge \neg(\texttt{IIO} \wedge \texttt{IIO_BUFFER}) && (6)\\
& \quad \wedge \neg(\texttt{TOUCHSCREEN_ADC} \wedge \texttt{IIO} \wedge \texttt{INPUT_TOUCHSCREEN}))
\end{aligned}$$

In summary, this formula means that `IIO_BUFFER_CB` is legal to enable if its reverse dependency holds (subexpression 3), is legal to either enable or disable if its direct dependency holds (subexpression 4), never takes a default value (subexpression 5), and can otherwise only be disabled when its direct and reverse dependencies do not hold (subexpression 6). The rest of the symbolic evaluator's valuation functions are similarly derived the formal semantics and can be found in the supplemental material.

The benefit of this approach is that is removes guesswork from designing Kconfig analysis tools. Instead, tool writers can rely on a common semantics to mechanically derive an analysis for whatever abstraction they would like to use for analysis, tailoring the choice of formalism for configuration options based on their specific application. We demonstrate just one possible set of choices for deriving the Kconfig analysis. Moreover, future extensions to Kconfig by developers can be captured by updates to the semantics, easing adoption for Kconfig tools that mechanically derive their analyses from the semantics.

5 IMPLEMENTATION

The bug finding tool, `kismet`, is implemented in about two thousand source lines of Python, and about one thousand source lines of C. It consists of four components. The `kextract` tool wraps the parser from the Linux implementation of Kconfig [52] with a C extension to desugar the Kconfig specification into an intermediate language. The `kclause` tool, written in Python, reads in the intermediate format and constructs logical formulas for each configuration options' constraints, outputting them in the SMTLIB2 [9] format. `kismet`, also written in Python, finds each `select` construct from the `kextract` output and calls `kclause` to generate the unmet dependency condition for each construct. `kismet` finally passes this condition in SMTLIB2 format to the `klocalizer` tool, which uses the Z3 SMT solver [14] to check for satisfiability. For satisfiable conditions, `klocalizer` can also generate solutions to the condition in the Linux `.config` file format, which we use to test the solution against the actual Kconfig implementation. All source code is available online as free and open-source software[2].

[2]Link omitted for blind review.

6 EXPERIMENTAL EVALUATION

We evaluate our bug finding approach for precision, performance, impact on real-world code.

6.1 Experimental Setup

We use Kconfig specifications from a recent version (v5.4.4) of the Linux kernel source code[3] as the target of our study. With over 140,000 lines of specifications and over 15,000 configuration options, Linux represents, to our knowledge, the largest user of Kconfig.

The Linux kernel not only provides a large Kconfig specification, but multiple ones as well, due to its support for multiple hardware platforms. Each of its 28 architecture families[4] has its own Kconfig specification, effectively providing 28 separate Kconfig specifications to use for evaluation. Because of the hardware abstraction layer, however, these architectures share at least some portion of the codebase in common, and therefore also share a large portion of their Kconfig specifications; about 100,000 lines, two-thirds, are architecture-independent. Each architecture has between 10,014 and 12,744 select constructs for a total of 289,202. Deduplicating these, there are 17,006 unique select constructs, although the constraints due to architecture-specific Kconfig files may differ. Due to this sharing, we not only report results for each architecture's Kconfig specifications but also the aggregate and deduplicated alarms across architectures.

All experiments were executed on a server with an AMD EPYC 7401 24-Core Processor with 512GB of RAM running Ubuntu 18.04, where we measured performance using the UNIX `time` utility. Since this machine allows for high parallelism, we ran the experiments for the 28 architectures' Kconfig specifications in parallel on separate copies of the Linux kernel source code. Replication scripts are available with the source code repository[5]

6.2 Data Availability

All experimental data are available as blinded, open data for submission[6]. We plan to turn our data into archived open data.

6.3 Research Questions

Our evaluation seeks to answer the following research questions:

RQ1 (Precision) How precise is our analysis when finding unmet dependencies? To measure bug-finding effectiveness, we run our tool on all 28 Linux Kconfig specifications and collect the alarms reported. We also automatically validate whether the alarms are true positives by generating and building test cases automatically. We expect that, if our semantics reflect real Kconfig behavior, that our symbolic model of unmet dependencies and Kconfig behavior should yield high precision, i.e., few false positives.

RQ2 (Performance) How fast is bug-finding?. We record the running time of our bug-finder when applied to all 28 Linux Kconfig specifications, i.e., the experiment from RQ1. We report the distribution of running times per architecture, the aggregate time, as well as the breakdown between desugaring, generating bug conditions, and solving. We expect that our design choices and optimization will yield a fast enough analysis to make running `kismet` feasible for developers to use regularly.

RQ3 (Impact) How useful are the resulting alarms to developers? We evaluate the impact of our bug-finding approach by manually submitting some reports and patches to the kernel maintainers. We expect that, if the resulting alarms are correct and provide value to the kernel maintainers, they will confirm the reports and accept our patches.

RQ4 (Comparison) How does our approach compare to random configuration testing? To our knowledge, no related tool for finding unmet dependencies in Kconfig exists. To provide a baseline time to search for bugs, we use random configuration testing with Kconfig's built-in `randconfig` tool. We compare the bugs found, given the same amount of time as `kismet` and also allow `randconfig` generation to run for several days. We expect that our static approach will perform better, given the enormity of the search space of configurations, but we also expect to find new bugs missed by `kismet`'s underapproximation of non-Booleans.

Table 1: `kismet`'s bug-finding results across all 28 architecture Kconfig specifications.

	Percentiles				
Metric	**Max**	**75th**	**50th**	**25th**	**Min**
Constructs	12,744	10,386	10,108	10,044	10,014
Alarms raised	53.00	31.25	25.00	22.75	10.00
Precision	100%	100%	100%	100%	100%

6.4 RQ1: Precision

We run `kismet` on each of the 28 architectures' Kconfig specifications and collect the resulting alarms. `kismet` reports the pair of configuration options involved in the unmet dependency, i.e., the *selector* and the *selectee*. Finally, we validate whether the alarm is a true positive by generating a test case. This works by querying the Z3 SMT solver for a satisfying solution to $\phi_{\text{unmet (precise)}}$, the bug's logical formula, then converting the solution into the Linux `.config` configuration file.

Table 1 summarizes the analysis results of our experiments. The rows list the number of constructs analyzed, the number of alarms raised by `kismet`, and the precision, i.e., the percent of all alarms that are true positives. The columns show the distribution of these metrics across the 28 architectures' Kconfig specifications as percentiles. `kismet` checks between 10,014 and 12,744 select constructs for each architecture, finding between 10 and 53 alarms per Kconfig specification, for a total of 781 alarms over 289,202 constructs. All alarms are confirmed to be true positives by generating test cases that trigger the alarm, for a precision of 100%. While such high precision would be unusual for static analysis, the core fragment of Kconfig that we model requires no over-approximation that could lead to false positives. Since the ground truth number of bugs in real-world Linux Kconfig specifications is unknown, we do not compute recall, but we address false negatives in RQ4.

Although each architecture has its own Kconfig specification, they all share a large common set of Kconfig files. The consequence

[3] https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.4.4.tar.xz

[4] alpha, arc, arm, arm64, c6x, csky, h8300, hexagon, i386, ia64, m68k, microblaze, mips, nds32, nios2, openrisc, parisc, powerpc, riscv, s390, sh, sh64, sparc, sparc64, um, unicore32, x86_64, and xtensa

[5] Link omitted for blind review.

[6] https://zenodo.org/record/4563310

Table 2: `kismet`'s bug-finding time in minutes for all 28 Kconfig specifications, broken down by each phase of analysis.

	Analysis Phase	Time Percentiles (minutes) Max	75th	50th	25th	Min
1.	`kclause`	7.21	5.52	5.35	5.21	5.03
2.	Syntax check	0.15	0.12	0.12	0.11	0.11
3.	ϕ_{unmet}	2.35	2.00	1.94	1.88	1.62
4.	$\phi_{\text{unmet (precise)}}$	79.31	33.79	32.16	31.23	29.08
5.	Confirmation	2.01	1.06	0.81	0.72	0.38
	Total Time	90.21	42.12	40.30	39.41	37.13

is that fixing a bug in one architecture's Kconfig specification can fix it for several others. Deduplicating these bug yields 151 total alarms for unique `select` constructs across all architectures. In some cases, the same `select` construct was a true unmet dependency in one architecture's Kconfig specification but not others, which is possible because of architecture-specific constraints. In these cases, we counted the construct as a true alarm in the deduplicated set.

Summary: our approach is precise, yielding 100% precision on Linux's very large, real-world Kconfig specification, and finds many new bugs: 781 true positive bugs or 151 if we deduplicate common constructs across architectures.

6.5 RQ2: Performance

To evaluate performance, we measure `kismet`'s running time, broken down by each phase of its analysis. Table 2 is the distribution of running times across each of the 28 architecture-specific Kconfig specifications. Each row is the phase of analysis, with the total time in the last row, while each column is percentiles in the distribution of running times.

`kismet` takes between 37 and 90 minutes on one Kconfig specification file, for a total of 20 hours in all, including the time spent generating a test case to automatically confirm true positives. We break down the timing into five phases: (1) *kclause* is the time spent modeling Kconfig constructs, which we perform at the beginning of analysis to cache the results. (2) *Syntax check* includes both identifying each select construct and the optimization that rules out selectees with no dependencies. As discussed in Section 4.2 on optimization, (3) ϕ_{unmet} is the time spent checking the imprecise bug formula, and (4) $\phi_{\text{unmet (precise)}}$ is the time spent checking the precise bug formula, if the imprecise one does not rule out the bug. (5) *Confirmation* is the time spent generating a test case for the bug and checking it against the actual Kconfig implementation; this is not part of the static analysis, per se, but it only takes a comparatively small amount of time.

In most cases, `kismet` takes less than hour for an architecture, making it fast enough for use on each commit of the Kconfig specification. The largest amount of time is spent on the precise formula check, which shows the importance of our optimization in avoiding making that check. Checking ϕ_{unmet} is fast: it takes less than an hour for hundreds of thousands of `select` constructs, albeit with low precision (less than 2%). 85% of the constructs are ruled out, however, reducing the time needed to solve the precise condition.

Summary:* `kismet` *is fast, taking between 37 and 90 minutes to analyze between 10,014 to 12,744 select constructs in a Kconfig specification, enabling frequent bug finding runs.

6.6 RQ3: Impact

We evaluate the impact of our bug-finder, and the semantics on which it is based, by reporting alarms to the kernel developers and submitting patches to the mainline Linux repository, specifically via the Linux kernel mailing list [48] and the kernel.org Bugzilla website [3]. Developer confirmation of bugs provides confidence in the utility of the alarms, beyond precision. Moreover, acceptance of patches by official maintainers reflects the beneficial impact of the results on this prevalent and frequently used codebase.

While our bug-finder is fully automated, submitting reports and patches is a manual process, requiring time to create them and communicate with human Linux maintainers. Moreover, maintainers may opt to not patch even true alarms, may not respond immediately, or may request different changes than what we proposed in the patch. Since the Kconfig specification gradually changes over time with the rest of the codebase, prior bugs may no longer occur, due to manual fixes, removal of options, etc. We believe it is feasible to use `kismet` in continuous integration, but we leave such infrastructure development as future work. For these reasons, we have not yet submitted all alarms; repairing them all is an ongoing process, and we report current state of the bug repairs in progress.

As of writing, we have submitted 38 reports or patches, 19 have been confirmed with the remainder pending, and 15 of our patches have already been committed to the Linux kernel codebase.

Knowing the effect of unmet dependencies on the kernel is difficult to measure. Such a configuration is not supposed to be feasible, and developers have been so far highly receptive to patches of unmet dependency bugs. While we do not know all the effects of an unmet dependency, one common result is a broken build, e.g., Figure 1, which is undesirable for any software product. We measured how often a broken build results on the bugs we found by attempting to build the generated `.config` from `kismet` and hand-checking the reason for the broken build. Build errors account for 68% of all tests. 29% of configuration files trigger build errors whose root cause is the unmet dependency bug from which the configuration file was generated. 27% fail due to bugs other than the one used to generate the test case. Since a build error halts the build process, we cannot easily determine whether the build would have encountered an error related to the unmet dependency, so we conservatively assume these are not caused by unmet dependencies.

Summary: The bug finding results have resulted in 38 reports and 15 committed patches to the Linux kernel so far, with further patch submission and discussion ongoing.

6.7 RQ4: Comparison

While `kismet` is 100% precise for its fragment of the Kconfig semantics, its underapproximation of non-Boolean leaves it susceptible to false negatives. To gather a set of unmet dependency benchmarks that include bugs not findable by `kismet`, we use a built-in Kconfig utility for generating random configurations. Generating random configurations for over four days for each architecture

Table 3: Percent of the bugs found by `kismet` compared to `randconfig` given both the same amount of time as and 135x more time than `kismet`.

	Percentiles				
Tool	**Max**	**75th**	**50th**	**25th**	**Min**
`kismet`	100.00%	100.00%	100.00%	100.00%	87.10%
`randconfig`					
Same time	62.86%	12.94%	6.80%	2.68%	0.00%
135x time	77.14%	22.55%	17.42%	10.54%	0.00%

in parallel (a combined time of more than three months), we generated over 11,000,000 configuration files, which raised 2,857,938 unmet dependency alarms, yielding 175 unique unmet dependency bugs. Comparing these to `kismet`'s results, `kismet` adds 614 unique unmet dependencies not found in this random testing.

Since no other tools to our knowledge analyze unmet dependencies, we compare the performance of `kismet` against a random testing approach, to see whether there is a benefit in running time and bugs found to using `kismet`. Using the combined set of bugs from months of `randconfig` and `kismet`'s results, we compare the percent of bugs found given the same amount of time. Table 3 shows the results of this comparison of the percentage of bugs found from the benchmark set. The columns show the distribution of these percentages across all architectures' Kconfig specifications. `kismet` finds 100% for almost all architectures, reflecting the fact that even after months of compute time, very few additional bugs were found by random testing compared with `kismet`. `randconfig` (row "Same time"), given the same amount of time that `kismet` took, finds on average only a small fraction of the set of bugs, 6.80%, with a maximum of only 62.86%. Even given several days to run (row "135x time"), `randconfig` still only finds a fraction of the benchmark bugs. In contrast, there were only eight bugs not found by `kismet`, leading to a worst-case of 87.10% benchmark coverage by `kismet`.

While our benchmark is not the ground truth of Linux's complete set of bugs, which is not feasible to find by hand, given the months of compute time to generate configuration files, it provides at least an estimate of the relative performance of `kismet` versus random testing. The results show the large performance benefit of using `kismet` compared to random testing. In the same amount of time, `kismet` finds many more bugs than random testing, providing a fast and precise complement to random testing that can be run regularly against new commits to the Kconfig specification.

Summary:* `kismet` *finds many more true positives bugs in far less time than random testing, although there are also false negatives as expected by deliberate underapproximation.

7 THREATS TO VALIDITY

Internal Threats. Our formal semantics needs to match the actual behavior of Kconfig, otherwise, any analyses based on it may yield incorrect results. We mitigated this using the Kconfig documentation, reviewing its actual C implementation, and collecting a Kconfig test suite. Moreover, the 100% precision of the bug-finder, validated with generated test cases and some developer confirmation, testifies to the accuracy of the semantics. `kismet` is deliberately underapproximate for non-Boolean options, however, so this part of the semantics is not supported by the bug-finding results, but by the documentation, implementation, and test suite only.

External Threats. While Kconfig is used by several popular, low-level systems software (BusyBox, coreboot, etc), our evaluation only applies to Linux. Linux, however, is the largest user of Kconfig that we know of, and has multiple Kconfig specifications. We evaluate our bug-finder on one recent version of the Linux source code, but Kconfig specifications change gradually with each kernel version. Different versions may yield different numbers of alarms. We leave a long-term study of Kconfig bugs across versions and projects as future work. Our bug-finder currently checks for one kind of bug. The performance of the bug-finder could vary for different bug types or analysis tasks. Our work is specific to the Kconfig specification language, so we do not show applicability to other specification languages. Given the large time investment in creating and evaluating accurate formal semantics and a corresponding analysis infrastructure, we leave generalizing the approach to other specification languages as future work.

8 RELATED WORK

Modeling Kconfig specifications. There are several prior efforts that convert Kconfig to logical formulas for various applications. Zengler et al. and Walch et al. modeled Kconfig in the DIMACS SAT format with the goal of finding Kconfig language metrics, including the number of options, types, and mandatory configuration options [55, 59]. She et al describe a formal semantics [42] and a tool called LVAT that converts Kconfig specifications to the DIMACS SAT solver format [10, 40, 43]. It was designed for collecting statistics about the Kconfig language such as the number of options, the hierarchy of dependencies, and other metrics [10], rather than for precise formal verification of configuration specifications. Tool development appears to have stopped for LVAT in 2013 [41]. The undertaker project has a tool to convert Kconfig's `dumpconf` output to the DIMACS SAT format for use in identifying dead code blocks in unpreprocessed C code [4, 51]. The `kconfigreader` tool converts the output of a Kconfig tool called `dumpconf`, which dumps each configuration options' constraint expressions, into the DIMACS SAT solver format [26, 31]. El-Sharkawy et al., describes an informal semantics of Kconfig, provides illustrative examples, and evaluates the limitations of other tools [16]. Fernandez et al. described informal semantics for Kconfig constructs that they identified as incorrectly supported in prior conversion tools [17]. They provide a set of example Kconfig constructs that illustrate these limitations, which we have incorporated into `kclause`'s test suite. Fernandez et al. also describe a new conversion tool that produces Binary Decision Diagrams but has not been evaluated on Linux Kconfig specifications.

Analyses of other configuration languages. Shambaugh et al. [39] perform formal verification of the Puppet deployment configuration language to detect non-deterministic system state updates and other undesirable system configurations. Weiss et al. [57] automate Puppet configuration repair using formal reasoning over a propositional model of the language. Anderson et al. [8] formally

verify the SmartFrog infrastructure deployment language to prove properties such as termination of compilation, comparing multiple implementations of SmartFrog compilers. Sotiropulous et al. formally modeled the system call trace of the Puppet tool to find faults from ordering violations on resource usage [47]. Horton and Parnin infer system dependencies from Python code in order to generate Docker specification files [23]. They also inferred from Python code snippets to check if their package dependencies are out of date [24]. Bouchet et al. use formal verification to check for inadvertent public access to Amazon S3 instances [13]. Chenygyuan et al. mined frequently used dependencies between entities from deployment descriptors for Java EE platform based applications to validate if a new deployment descriptor is violating mined dependencies [58]. Hanappi et al. formally modeled the configuration scripts and their resource usage to test if a system can recover from failures such as network outages and reach a stable state [22].

Studies on variability bugs. Some prior work extracted variability information from Makefiles and source code for finding bugs, dead code blocks, or inconsistencies between variability specification and implementation. [12, 15, 20, 37, 38, 46, 50]. Prior work also analyzed bugs or warnings raised from sampled configurations to classify them and understand how they are introduced [34, 35]. Similar analyses were performed on the bugs or vulnerabilities reported in the bug database or source commits [6, 7, 18, 36]. Others studied configuration sampling algorithms to find more variability bugs with fewer samples [30, 33, 44, 53].

9 CONCLUSION

We have introduced a new formal semantics and model checking infrastructure for analyzing Kconfig specification files and methodically derived a bug-finder, called `kismet`, for unmet dependencies, a common pitfall for Kconfig maintainers. Our results show that our bug-finder is precise, fast, and has resulted in patches to the mainline Linux kernel source code confirmed and accepted by maintainers. Future work includes continuing to repair all bugs found by `kismet`, applying it to ongoing kernel development and other software, and applying our analysis techniques to other languages.

REFERENCES

[1] 2008. Boogie: An Intermediate Verification Language. https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/.
[2] 2020. Kconfig macro language. https://www.kernel.org/doc/html/latest/kbuild/kconfig-macro-language.html, last accessed on 11/19/20.
[3] 2020. Kernel.org Bugzilla page. https://bugzilla.kernel.org/, last accessed on 11/19/20.
[4] 2020. Undertaker Project Page. https://vamos.informatik.uni-erlangen.de/trac/undertaker, last accessed on 11/19/20.
[5] 2021. 0-Day Test Service. https://01.org/lkp/documentation/0-day-test-service.
[6] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 421–432.
[7] Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability bugs in highly configurable systems: A qualitative analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26, 3 (2018), 1–34.
[8] Paul Anderson and Herry Herry. 2016. A formal semantics for the SmartFrog configuration language. *Journal of Network and Systems Management* 24, 2 (2016), 309–345.
[9] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening (Eds.).
[10] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640.
[11] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *PLDI*. ACM.
[12] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. Spllift: Statically analyzing software product lines in minutes instead of years. *ACM SIGPLAN Notices* 48, 6 (2013), 355–364.
[13] Malik Bouchet, Byron Cook, Bryant Cutler, Anna Druzkina, Andrew Gacek, Liana Hadarean, Ranjit Jhala, Brad Marshall, Dan Peebles, Neha Rungta, Cole Schlesinger, Chriss Stephens, Carsten Varming, and Andy Warfield. 2020. Block public access: trust safety verification of access control policies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 281–291.
[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[15] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. A robust approach for variability extraction from the Linux build system. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM, 21–30.
[16] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the Kconfig semantics and its analysis tools. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 45–54.
[17] David Fernandez-Amoros, Ruben Heradio, Christoph Mayr-Dorn, and Alexander Egyed. 2019. A kconfig translation to logic with one-way validation system. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. 303–308.
[18] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. 2016. Do# ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel. In *Proceedings of the 20th International Systems and Software Product Line Conference*. 65–73.
[19] Alejandra Garrido and Ralph Johnson. 2005. Analyzing Multiple Configurations of a C Program. In *ICSM*. 379–388.
[20] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 279–290. https://doi.org/10.1145/3106237.3106283
[21] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. ACM, New York, NY, USA, 323–334. https://doi.org/10.1145/2254064.2254103
[22] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. 2016. Asserting reliable convergence for configuration management scripts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 328–343.
[23] Eric Horton and Chris Parnin. 2019. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 328–338.
[24] Eric Horton and Chris Parnin. 2019. V2: fast detection of configuration drift in Python. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 477–488.
[25] Alexandru Florin Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2017. Effective Analysis of C Programs by Rewriting Variability. *CoRR* (2017).
[26] Christian Kästner. 2020. kconfigreader. https://github.com/ckaestne/kconfigreader, last accessed on 11/19/20.
[27] Christian Kästner, Paolo G Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 805–824.
[28] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A Variability-aware Module System. In *OOPSLA*. ACM, 773–792.
[29] The kernel development community. 2020. Kconfig Language. https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html, last accessed on 11/19/20.
[30] Chang Hwan Peter Kim, Don S Batory, and Sarfraz Khurshid. 2011. Reducing combinatorics in testing product lines. In *Proceedings of the tenth international conference on Aspect-oriented software development*. 57–68.
[31] Christian Kästner. 2017. Differential Testing for Variational Analyses: Experience from Developing KConfigReader. arXiv:1706.09357 [cs.SE]
[32] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 81–91.
[33] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *2016*

IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 643–654.

[34] Jean Melo, Elvis Flesborg, Claus Brabrand, and Andrzej Wąsowski. 2016. A Quantitative Analysis of Variability Warnings in Linux. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems* (Salvador, Brazil). ACM, New York, NY, USA, 3–8.

[35] Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. 2019. An empirical study of real-world variability bugs detected by variability-oblivious tools. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 50–61.

[36] Raphael Muniz, Larissa Braz, Rohit Gheyi, Wilkerson Andrade, Baldoino Fonseca, and Márcio Ribeiro. 2018. A qualitative analysis of variability weaknesses in configurable systems with# ifdefs. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*. 51–58.

[37] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where do configuration constraints stem from? an extraction approach and an empirical study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 820–841.

[38] Sarah Nadi and Ric Holt. 2012. Mining Kbuild to detect variability anomalies in Linux. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 107–116.

[39] Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 416–430.

[40] Steven She. 2013. Feature model synthesis. (2013).

[41] Steven She. 2013. LVAT Archive. https://code.google.com/archive/p/linux-variability-analysis-tools/, last accessed on 11/19/20.

[42] Steven She and Thorsten Berger. 2010. Formal semantics of the Kconfig language. *Technical note, University of Waterloo* 24 (2010).

[43] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*. 461–470.

[44] Jiangfan Shi, Myra B Cohen, and Matthew B Dwyer. 2012. Integration testing of software product lines using compositional symbolic execution. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 270–284.

[45] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. 2007. Is the linux kernel a software product line?. In *Proceedings of the International Workshop on Open Source Software and Product Lines* (Kyoto, Japan) *(SPLC-OSSPL)*. 134–140.

[46] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2010. Efficient extraction and analysis of preprocessor-based variability. In *Proceedings of the ninth international conference on Generative programming and component engineering*. 33–42.

[47] Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. Practical fault detection in puppet programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 26–37.

[48] Jasper Spaans. 2020. Linux Kernel Mailing List. https://lkml.org/, last accessed on 11/19/20.

[49] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90, 000# ifdefs Issue.. In *USENIX Annual Technical Conference*. 421–432.

[50] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. 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, 47–60.

[51] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 531–551.

[52] Linux Tovalds. 2020. Linux Kconfig Source Code. https://github.com/torvalds/linux/tree/master/scripts/kconfig, last accessed on 11/19/20.

[53] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A classification of product sampling for software product lines. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*. 1–13.

[54] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* 27, 4 (2018), Article No. 18. https://doi.org/10.1145/3280986

[55] Martin Walch, Rouven Walter, and Wolfgang Küchlin. 2015. Formal analysis of the Linux kernel configuration with SAT solving.. In *Configuration Workshop*. 131–138.

[56] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. 2014. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Onward!* ACM, 213–226.

[57] Aaron Weiss, Arjun Guha, and Yuriy Brun. 2017. Tortoise: Interactive system configuration repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 625–636.

[58] Chengyuan Wen, Yaxuan Zhang, Xiao He, and Na Meng. 2020. Inferring and Applying Def-Use Like Configuration Couplings in Deployment Descriptors. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

[59] Christoph Zengler and Wolfgang Küchlin. 2010. 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. 51–56.