Kmax: Finding All Configurations of Kbuild Makefiles Statically

Paul Gazzillo
Yale University, USA
paul.gazzillo@yale.edu

ABSTRACT

Feature-oriented software design is a useful paradigm for building and reasoning about highly-configurable software. By making variability explicit, feature-oriented tools and languages make program analysis tasks easier, such as bug-finding, maintenance, and more. But critical software, such as Linux, coreboot, and BusyBox rely instead on brittle tools, such as Makefiles, to encode variability, impeding variability-aware tool development. Summarizing Makefile behavior for all configurations is difficult, because Makefiles have unusual semantics, and exhaustive enumeration of all configurations is intractable in practice. Existing approaches use ad-hoc heuristics, missing much of the encoded variability in Makefiles. We present Kmax, a new static analysis algorithm and tool for Kbuild Makefiles. It is a family-based variability analysis algorithm, where paths are Boolean expressions of configuration options, called *reaching configurations*, and its abstract state enumerates string values for all configurations. Kmax localizes configuration explosion to the statement level, making precise analysis tractable. The implementation analyzes Makefiles from the Kbuild build system used by several low-level systems projects. Evaluation of Kmax on the Linux and BusyBox build systems shows it to be accurate, precise, and fast. It is the first tool to collect all source files and their configurations from Linux. Compared to previous approaches, Kmax is far more accurate and precise, performs with little overhead, and scales better.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; *Interpreters*; Software testing and debugging;

KEYWORDS

Kmax, Makefiles, Kbuild, Variability, Configuration, Static Analysis

ACM Reference Format:
Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 12 pages.
https://doi.org/10.1145/3106237.3106283

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106283

1 INTRODUCTION

Feature-oriented software design (FOSD) is a paradigm that provides a principled way to develop highly-variable software [3]. A line of software products can be produced from a single codebase by combining a selection of features. By making features explicit and carrying them through the build process, variability in FOSD programs is easier to reason about and more amenable to automated software engineering tools.

Some of the most critical software infrastructure, however, does not use FOSD-friendly tools, instead relying on brittle languages such as Makefiles to encode variability, impeding variability-aware analysis. Linux, BusyBox, Apache and other systems that use Makefiles have been shown to harbor variability bugs that are not exposed unless a particular configuration is tested [1]. Even bugs easy to find for a single-configuration bugfinder have been unwittingly committed into these codebases, from simple bugs such as linker errors to more pernicious buffer overflows, and testing all configurations is infeasible.

Beyond bug-finding, many other software engineering tasks depend on a complete picture of Makefile variability: measurement of the scale, evolution, and interactions of features in variable systems [12, 15, 26, 31, 38, 43]; program analyses such as dataflow analysis, code coverage, and feature model validation [13, 23–25, 39, 45, 46]; code maintenance such as evolution and dead code elimination [35, 36, 46]; and translating existing variability to new variability encodings [6, 21, 27]. All of these analyses use ad-hoc tools or ignore Makefile variability altogether, resulting in an incomplete feature-model that necessarily excludes configurations. For instance, several studies on the Linux 2.6.33.3 x86 kernel source explicitly report using 7,691 or fewer C source files [17, 20, 25] while our new results show that there are 9,044 C source files, 17% more that have been missed in previous experiments.

While Makefiles are widely used, they remain a pain point for automated reasoning about build system variability. Brute force enumeration is intractable, because configurations are exponential in the number of configuration options. Static analysis is hard, because Makefiles are written in an expressive programming language that has unusual semantics. For instance, variable names can be constructed at runtime via string operations.

Previous attempts to analyze build system Makefiles, such as fuzzy parsing and brute force, have been ad-hoc and incomplete [5, 12, 33, 46]. Their heuristics elide the complexities of the Makefile language. As a result, they miss much information, limiting the completeness and correctness of downstream analysis tools. To be fair, these tools were not designed to be a complete analysis of configurations, and provide enough precision for the experimental evaluation of their respective contributions. In contrast, our aim is an accurate and precise configuration analysis that can be used for variability analysis of build systems themselves and for downstream tools.

Our goal is to enable *family-based* analysis of build system variability [3]. A family-based analysis reasons about the space of possible configurations as a whole, typically in a single pass, as opposed to reasoning about configurations separately. Previous heuristics for Kbuild Makefile analysis have difficulty covering the configuration space tractably, so are less useful for reasoning about variability. An example of the utility of a variability-aware analysis is finding unreachable code. The Linux kernel contains source that can never be compiled, either due to bugs in the Makefiles or infeasible combinations of configurations. More subtly, many source files are only feasible when building for a specific architecture. This is useful information for software tools, e.g., a bug found in an infeasible configuration is a false positive. Reasoning about the entire space of configurations makes handling such issues feasible.

KBuildMiner [5] and GOLEM [12, 46] represent state of art approaches to analyzing Kbuild Makefile configurations. KBuildMiner works by parsing common Makefile usage patterns. But parsing alone invariably misses much by not accounting for Makefiles semantics, even when Makefiles are manually adjusted to fit the parser's grammar. GOLEM uses heuristic brute force enumeration. It enables one or more features at a time and executes the Makefiles, recording the resulting output. But this approach is very slow, and still only covers a fraction of all configurations, because of the sheer number of combinations of configuration options. In contrast, we approach variability in Makefiles as a static analysis problem. This is a natural fit: features are Makefile variables and variability is encoded with if-then-else statements and variable expansion.

We introduce a new, family-based, static analysis algorithm called Kmax for Kbuild Makefiles that yields a precise description of variability. Kmax provides abstractions for the configuration and string domains, and its data-flow results yield a precise summary of all possible configurations of a Makefile. The analysis is path-sensitive, where paths model the *reaching configurations*, i.e., the combinations of configuration settings that lead to a particular point in the Makefile program. By using Binary Decision Diagrams (BDDs) to concisely represent paths, we can model all configurations with good performance in practice, even on the highly-configurable build system for Linux.

Kmax's abstract state enumerates all configurations of string values, which localizes configuration explosion to the statement-level instead of the entire Makefile. This enables precise analysis of complex string operations in Makefiles such as runtime variable name construction. In a single pass, Kmax collects a BDD representation of the space of configurations from Kbuild Makefiles. With this representation, it can incorporate further constraints on the space due to developer policy, e.g., drivers that only apply to a specific platform.

Kmax's analysis approach is implemented in a new tool, which analyzes Makefiles written for the Kbuild build system used by many highly-configurable projects including the Linux kernel, BusyBox, and coreboot. The Kmax tool collects all C files comprising the codebase and Boolean expressions of configuration options that control them, i.e., the feature model. We evaluate Kmax for accuracy, precision, and performance on the build systems for the Linux kernel and BusyBox. Kmax takes only minutes on the Linux build system and is the first to correctly find all C files for the Linux kernel and their configurations. Kmax is compared against the state of art heuristic approaches, KBuildMiner and GOLEM, and is shown to be much more accurate and precise with better or little performance overhead and better scalability. Because of its principled approach to variability, Kmax is able to identify dead and orphaned code due to infeasible configurations.

The contributions of this paper are as follows:

- A new static analysis algorithm for Kbuild Makefiles (Section 3).
- An implementation of the algorithm in a new tool, Kmax, for the Makefile-based Kbuild build system that collects all C source file names and the configurations in which they are built (Section 4).
- An experimental evaluation of Kmax on two build systems that demonstrates its precision, accuracy, and speed, taking minutes for the Linux build system and seconds for BusyBox (Section 5).
- A comparison of Kmax to previous Kbuild analysis tools KBuildMiner and GOLEM on two versions of the Linux source code showing that Kmax is substantially more accurate and precise with little performance overhead and better scalability (Section 6).

Kmax is available online for download[1].

2 OVERVIEW

Figure 1 is a small, representative example drawn from the thousands of individual Makefiles comprising the Linux build system[2]. For brevity, this example involves two configuration options `A` and `B`, but Makefiles in practice may have dozens or hundreds of configuration options, making exhaustive enumeration of all configurations infeasible.

`A` and `B` are Boolean Linux configuration options, which in Linux means that the option is set either to `y` for yes to include the feature or is left unset to exclude the feature. Makefiles do not have any other variable type besides string, so Boolean is a convention describing this range of values for a configuration option. When executed, this Makefile determines, given inputs `A` and `B`, which source code files to include in the build. Lines 1–7 programmatically construct the set of filenames and store them in the variable `obj-y`, which is used in the rule on lines 8–9 to perform the actual build.

The first statement (line 1) is an assignment that initializes `obj-y` to `fork.o`, the object file corresponding to the `fork.c` source file. The following if-then-else block tests whether `A` equals `y` (line 2), i.e., whether feature `A` has been enabled. Variables are referenced (*expanded* in Makefile terminology) with the `$(A)` syntax. Depending on `A`'s value, `BITS` is either set to to `32` (line 3) or to `64` (line 5).

The next statement (line 7) is another type of assignment (`+=`) that appends to an existing variable value. Looking first at the *right-hand* side, the value of the assignment is computed by first expanding any variable references. In this case, `BITS` is expanded to one of its earlier, configuration-dependent definitions, `32` or `64`. By Makefile semantics, any adjacent strings are implicitly concatenated, resulting in `probe_32.o` or `probe_64.o`. Notice that because `BITS` is configuration-dependent, the filename itself is now configuration-dependent, depending indirectly on input `A`.

[1] http://github.com/paulgazz/kmax.
[2] All Linux examples are from v3.19.

```
1 obj-y := fork.o
2 ifeq ($(A),y)
3     BITS := 32
4 else
5     BITS := 64
6 endif
7 obj-$(B) += probe_$(BITS).o
8 built-in.o: $(obj-y)
9   # do compilation
```

Figure 1: An example illustrating the challenges of static analysis of Makefiles. Collected from Linux kernel source: kernel/Makefile, arch/x86/Makefile, arch/x86/kernel/apic/Makefile, and scripts/Makefile.build.

Table 1: The final state of all variables used in Figure 1 for all combinations of configuration options A and B.

A	B	obj-y	obj-
on	on	fork.o probe_32.o	
on	off	fork.o	probe_32.o
off	on	fork.o probe_64.o	
off	off	fork.o	probe_64.o

The *left-hand* side of this assignment exposes Makefile's unusual support for runtime variable name construction. The name of the variable to assign is computed in exactly the same way as the value on the right side. In this case, B is expanded and concatenated with the prefix obj-, so the variable name will either be obj-y or just obj- when B is unset. Because variable names can, and often are, constructed at runtime with string operations, the *variable name itself is configuration-dependent*. This complicates static analysis, because the abstract state needs to map variable names to their abstract values. Crucially, if our abstraction of string values is not precise enough, our summary of variability will be very imprecise, unable to collect the names of source files or how and if they are added to obj-y to be built. The usage patterns in Figure 1 are very common in Linux build system Makefiles.

Figure 1 happens to be small enough to execute all combinations of configuration options, though in practice Makefiles can have too many configuration options to do so. Table 1 shows the final program state for each configuration of Figure 1, one per row. The first two columns are the settings for A and B respectively. For simplicity, "on" means the configuration option is set to y, while "off" means it is left undefined. The third and fourth columns are the final values for obj-y and obj-, respectively, for each combination of inputs. For instance, when A is off and B is on, BITS is set to 64 by line 5, then line 7 appends probe_64.o to obj-y. Notice that B controls the name of the variable being assigned: whenever B is off, obj-, not obj-y, gets the assignment from line 7, preventing either of the probe_##.o files from being built.

As the number of configuration options grows, computing this complete table by brute force is infeasible. Kmax summarizes this table for larger numbers of configuration options than would be feasible with brute force by abstracting the representations of configurations and program state. Configuration options are treated symbolically. Program state uses abstracted string values whose abstract domain is precise enough for runtime string computation, even of variable names, while still being tractable.

Kmax represents paths with symbolic Boolean expressions of configuration options. To reflect the notion that a path captures variability, we call this expression a *reaching configuration*. The path condition is updated by if-then-else constructs and tracked for each program point by Kmax. The abstract state of a program value is a set of concrete string values tagged with their reaching configurations. This domain permits Kmax to summarize all paths, yet have the runtime string values available for computing precise variable names and values. Input variables are also given in this abstract domain to describe all possible input values. Linux and BusyBox have well-defined configuration options, and our implementation is able to automatically construct these input values for nearly all configuration options.

Let us see how Kmax works on the example in Figure 1. The path condition begins as ⊤ or true, which represents all configurations. The first assignment sets obj-y to fork.o under this path condition, i.e., all configurations (line 1). Then the algorithm computes an updated path condition for both the if and else branches of the if-then-else block (line 2), i.e., "A is on" and "A is off" respectively. The path condition is represented with Binary Decision Diagrams (BDDs), where BDD variables correspond to these input values. Updates to the path condition are backed by Boolean operations on the BDD. This representation enables a much more compact representation of the configuration than the exhaustive table of all configurations. Moreover, BDDs can be compared for equality efficiently, which Kmax exploits to deduplicate abstract state.

Each branch updates the value of BITS *under its own path condition* (lines 3 and 5). Both of its possible values, 32 and 64, are stored in the abstract symbol table and tagged with the path conditions in which they were defined, i.e., the reaching configurations. The assignment statement on line 7 illustrates Kmax's abstract semantics when expanding these values. Kmax expands all variables to all definitions and executes all resulting assignments according to their concrete semantics. For B on, i.e., set to y, the assignment to obj-y is executed for both definitions of BITS, 32 and 64. Likewise, for B off, i.e., left unset, the assignment to obj- is executed for both definitions of BITS as well. This results in the three unique definitions (after deduplication) for obj-y and two for obj- that are shown in Table 1. The next section details the intricacies of the complete analysis algorithm, such as how Kmax tracks the reaching configurations, ensures the abstract state covers all configurations, performs variable expansion, and more.

Each variable's configurations are handled separately, and the number of unique abstract configurations stored in the abstract state is smaller, three compared to four, than for brute force enumeration. The difference is even more pronounced for Makefiles with larger numbers of configuration options. Kmax's abstraction localizes the explosion caused by exhaustive enumeration of configurations to the statement-level, while preserving the precise string values needed to evaluate Makefile semantics like runtime variable name computation. While abstract values can themselves suffer intractable blowup in theory, they are efficient in practice because

of this localization. Moreover they are amenable to deduplication and infeasible path trimming. Our implementation takes minutes even on the highly-configurable build system of Linux.

3 ALGORITHM

The Kmax algorithm is a path-sensitive static analysis of Makefile constructs. Like the concrete Makefile evaluator, Kmax keeps track of a symbol table of variable assignments and a list of collected rules. There are, however, two key differences. First, the analysis maintains a *path condition* at each program point. The path condition reflects some combination of conditional branches from the beginning of the Makefile to the program point. It is represented with a Binary Decision Diagram (BDD), where BDD nodes are the input variables to the Makefile. Because input variables are configuration options, the path condition corresponds to reaching configurations, i.e., a Boolean expression representing the configuration settings leading to that point in the program.

Second, Kmax maintains abstract versions of the symbol table and rule list. The symbol table stores, for each variable in the program, an enumeration of each possible concrete string value the variable may take. These concrete values are tagged with the path condition in which each assignment was made, possible because Kmax tracks the path condition at each program point. Similarly, the rule list maintains an enumeration of concrete rules tagged their path conditions. The utility of this design is that it enables abstraction over all paths, even when there is runtime string manipulation.

The abstract state blends the precision of concrete evaluation with the generality of abstraction. Conveniently, the Makefile language does not have a loop construct, enabling the analysis to be very precise since finding a fixed-point is unnecessary. While enumerating configurations in the abstract state does mean superlinear running time for the analysis, the symbolic representation of paths via BDDs makes for an efficient abstraction of paths and enables aggressive trimming and deduplication of the abstract state.

3.1 Definitions

Path conditions. A single path condition is denoted with $\hat{p}$ and is a BDD with its usual Boolean operations. Kmax uses a stack P to track the current path condition in nested conditional blocks and restore the path condition after a conditional block. The stack provides the methods $\textsc{Push}(P, \hat{p})$, $\hat{p} \leftarrow \textsc{Pop}(P)$, and $\hat{p} \leftarrow \textsc{Peek}(P)$ with their usual definitions.

Abstract values. An *abstract value* is denoted by $\bar{\mathbf{v}}$ and is a set of pairs $\{(v, \hat{p})\}$, where v is a concrete value and $\hat{p}$ is a path condition. It is an abstraction of concrete strings. They are the bread-and-butter of the analysis and are used as the input and output to several functions, such as variable expansion and symbol table lookups.

We define a special cross-product operation $\times^\circ$ for abstract values. It lifts any concrete operator $\circ$, such as string concatenation, for use on abstract values. It computes the operation for all possible combinations of concrete values, producing the corresponding path conditions. For two abstract values $\bar{\mathbf{v}}$ and $\bar{\mathbf{w}}$ and some concrete operation $\circ$, it is defined to be

$$\bar{\mathbf{v}} \times^\circ \bar{\mathbf{w}} = \{(v \circ w, \hat{p} \wedge \hat{q}) \mid \forall (v, \hat{p}) \in \bar{\mathbf{v}}, \forall (w, \hat{q}) \in \bar{\mathbf{w}}\}$$

As an optimization, the implementation removes entries when $\hat{q} \wedge \hat{p} = \bot$, i.e., infeasible paths.

BDD construction. The MakeBDD function takes an abstract value from a conditional guard and generates a BDD. Conditional expressions can themselves contain variables and function calls that need to be expanded to an abstract value first. (The Expand algorithm is described in the next subsection.) After expanding the conditional guard, MakeBDD evaluates the resulting abstract value $\bar{\mathbf{v}}$, producing a new BDD.

$$\textsc{MakeBDD}(\bar{\mathbf{v}}) = \bigvee_{(v,\hat{p}) \in \bar{\mathbf{v}}} \text{BDD}(v) \wedge \hat{p}$$

MakeBDD works by converting each concrete part of the abstract value into a BDD and conjoining it with the corresponding path condition. Finally, the terms are unioned. The resulting BDD is used to compute new path conditions for conditional blocks.

The symbol table. S denotes the symbol table. It stores the definitions of each variable in the program during analysis. The symbol table is a set of triples $(v, d, \hat{p})$. v is the concrete name of the variable, d is its concrete definition, and $\hat{p}$ is the path condition in which the $v \rightarrow d$ mapping holds. An entry for v where d is `null` represents an undefined variable, because the empty string is still considered to be a defined variable in Makefile semantics.

The symbol tables maintains the following invariants:

(1) The definitions for each variable span all paths, i.e.,

$$\forall v \forall (v, d, \hat{p}) \in S, \bigvee \hat{p} = \top$$

This ensures that expansions of the variable cover all possible paths through the program.

(2) All definitions for a variable v are mutually exclusive, i.e.,

$$\forall v \; \forall (v, d_i, p_i), (v, d_j, p_j) \in S, d_i \neq d_j, p_i \wedge p_j = \bot$$

This invariant ensures that, at any point in the program, there is one and only one definition per path.

The first invariant is maintained by adding a `null` definition for any paths that do not assign the variable. The second invariant, however, requires careful handling of updates to the symbol table.

A special update operation $\uplus$ adds a new definition $(v, d, \hat{p})$ to S, preserving the invariant. Before adding the new definition, it updates the path conditions of any existing entries $(w = v)$ to exclude the new definition's path condition, i.e., $(\hat{q} \wedge \neg\hat{p})$. All other entries $(w \neq v)$ are left alone. Formally, this is

$$\begin{aligned} S \uplus (v, d, \hat{p}) = &\{(w, *, \hat{q} \wedge \neg\hat{p}) \quad \mid (w, *, \hat{q}) \in S, w = v\} \\ &\cup \{(w, *, \hat{q}) \quad \mid (w, *, \hat{q}) \in S, w \neq v\} \\ &\cup \{(v, d, \hat{p})\} \end{aligned}$$

As an optimization, the implementation removes entries when $\hat{q} \wedge \neg\hat{p} = \bot$, i.e., infeasible paths.

Define updates the symbol table S given *abstract* values for the variable name $\bar{\mathbf{v}}$ and the definition $\bar{\mathbf{d}}$ under a given path condition $\hat{p}$. It uses $\uplus$ to update the symbol table with every combination of concrete variable name and definition under the appropriate path condition.

$$\begin{aligned} \textsc{Define}(S, \bar{\mathbf{v}}, \bar{\mathbf{d}}, \hat{p}) = S \uplus (v, d, \hat{p} \wedge \hat{p}_v \wedge \hat{p}_d) \\ \forall (v, \hat{p}_v) \in \bar{\mathbf{v}}, \forall (d, \hat{p}_d) \in \bar{\mathbf{d}} \end{aligned}$$

Lookup retrieves the definition of a given abstract variable name. It produces a new abstract value with each definition of each possible variable name under the appropriate path conditions.

$$\textsc{Lookup}(S, \bar{\mathbf{v}}) = \{(d, \hat{p} \wedge \hat{q}) \mid (v, \hat{p}) \in \bar{\mathbf{v}}, (v, d, \hat{q}) \in S\}$$

$\textsc{IsDefined}_{\text{conc}}$ returns the path conditions under which a given concrete variable name v is defined. It forms the union of the path conditions for all definitions that are not `null`, representing all the paths in which the variable has been assigned.

$$\textsc{IsDefined}_{\text{conc}}(S, v) = \bigvee_{(v,d,\hat{p}) \in S} (\hat{p}), \text{ for } d \neq \texttt{null}$$

The abstract version of IsDefined combines all the situations in which the abstract variable name is defined. For each $(v, \hat{p})$ of the abstract variable name $\bar{\mathbf{v}}$, it uses $\textsc{IsDefined}_{\text{conc}}$ on v, restricting the resulting path condition via $\hat{p}$. These results are then unioned.

$$\textsc{IsDefined}(S, \bar{\mathbf{v}}) = \bigvee_{(v,\hat{p}) \in \bar{\mathbf{v}}} (\hat{p} \wedge \textsc{IsDefined}_{\text{conc}}(S, v))$$

The rules set. R is the set of Makefiles rules. A rule is a tuple (t, d, c, p), where t represent the rule targets, d its dependencies, c the list of commands to run when the rule is matched, and $\hat{p}$ is the path condition in which the rule appears. Rules can appear in conditionals and have variable expansion, making them configuration-dependent as well. Adding a new rule works by taking each combination of possibilities for the targets $\bar{\mathbf{t}}$, dependencies $\bar{\mathbf{d}}$, and commands $\bar{\mathbf{c}}$ and adding them to the rule set under the appropriate path condition, i.e.,

$$\textsc{AddRules}(R, \bar{\mathbf{t}}, \bar{\mathbf{d}}, \bar{\mathbf{c}}, \hat{p}) = R \cup \bigcup (t, d, c, \hat{p} \wedge \hat{p}_t \wedge \hat{p}_d \wedge \hat{p}_c)$$
$$\forall (t, \hat{p}_t) \in \bar{\mathbf{t}}, \forall (d, \hat{p}_d) \in \bar{\mathbf{d}}, \forall (c, \hat{p}_c) \in \bar{\mathbf{c}}$$

3.2 The Static Analysis Algorithm

Algorithm 1 is the Makefile static analysis algorithm Kmax. Kmax takes a Makefile m and an initial symbol table S_0 containing the abstract values of the input variables and returns the resulting symbol table and rule set. Global variable are first initialized (lines 2 and 3). P, a stack of path conditions from nested conditional blocks, is initialized to $\top$, i.e., all configuration. The symbol table S is initialized to the input values S_0, and the rule R set starts as the empty set.

Two core methods comprise the analysis: Eval (lines 4–30) handles *statements* and Expand (lines 31–46) handles *expressions*. Eval iterates over each statement (line 5). The path condition for each statement is computed from the stack of path conditions P (line 6). This represents the conjunction of conditions due to nested conditionals. Lines 8-17 show how Makefile conditional statements update these path conditions.

Updating path conditions. `ifeq` tests for equality between the value of two expressions x and y (line 8). In normal Makefile evaluation, any variables or function calls are first expanded, and the resulting string is compared for equality. In Kmax, variables are expanded to abstract values with Expand (line 9). The two abstract values $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ are crossed using the lifted equality operation $\times^{==}$, then MakeBDD converts the resulting abstract value into a BDD (line 10). This new BDD path condition is then pushed onto P. `ifdef`

Algorithm 1 Kmax(m, S_0) - Evaluate a Makefile

Input: A Makefile m and an initial symbol table S_0
Output: The final symbol table and rule set

```
1:  function Kmax(m, S_0)
2:    P = ∅; Push(P, ⊤)                                  ▷ Initialize to all paths
3:    S = S_0; R = ∅                                     ▷ Initialize abstract state
4:    procedure Eval(m)
5:      for each statement ∈ m do
6:        p̂ ← ⋀_{q̂∈P} q̂                                ▷ Compute current path condition
7:        switch statement do
8:          case ifeq (x, y)
9:            x̄ ← Expand(x); ȳ ← Expand(y)
10:           Push(P, MakeBDD(x̄ ×^== ȳ))
11:         case ifdef (x)
12:           x̄ ← Expand(x)
13:           Push(P, IsDefined(S, x̄))
14:         case else
15:           q̂ ← Pop(P); Push(P, ¬q̂)
16:         case endif
17:           Pop(P)
18:         case x:=y                                    ▷ Variable assignment
19:           x̄ ← Expand(x); ȳ ← Expand(y)
20:           S ← Define(S, x̄, ȳ, p̂)
21:         case t : d c                                 ▷ Rule definition
22:           t̄ ← Expand(t); d̄ ← Expand(d)
23:           c̄ ← Expand(c)
24:           R' = AddRules(R, t̄, d̄, c̄, p̂)
25:         case include x                               ▷ Makefile inclusion
26:           for each (m, q̂) ∈ Expand(x) do
27:             Push(P, p̂ ∧ q̂)
28:             Eval(m)
29:             Pop(P)
30:   end procedure
31:   function Expand(e)
32:     ē ← ∅
33:     for each subexp ∈ e do
34:       switch subexp do
35:         case x                                       ▷ Concrete string
36:           s̄ ← {(x, ⊤)}
37:         case xy                                      ▷ Concatenation
38:           s̄ ← Expand(x) ×^concat Expand(y)
39:         case $(x)                                    ▷ Variable expansion
40:           s̄ ← Lookup(S, Expand(x))
41:         case $(funname arglist)                      ▷ Function call
42:           ā ← Expand(arglist)
43:           s̄ ← {(funname(a), p̂) | ∀(a, p̂) ∈ ā}
44:       ē ← ē ×^join(' ') s̄                            ▷ Combine each subexpression
45:     return ē
46:   end function
47:   Eval(m)                                            ▷ Start evaluating the input Makefile
48:   return (S, R)                                      ▷ Return the final abstract state
49: end function
```

does much the same thing (line 11). x is expanded (line 12) and the symbol table is queried via IsDefined for the conditions under which it has definitions (line 13). Then this new path condition is pushed onto P.

`else` is handled differently. Its path represents the negation of its sibling if statement. The last path condition pushed onto P is that from the sibling `if`, because `else` can not appear alone. Line 15 pops this path condition and pushes the negation of it back onto P. `endif` terminates a conditional block and simply pops P (line 17) to return to the last path condition before entering the conditional block.

Collecting abstract state. Variable assignment maps a variable name x to a definition y (line 18). In Makefiles, both sides of the assignment are expanded (line 19), enabling runtime construction of variable names. This complexity is subsumed by abstract values and the Define operation. Define adds the new definitions to the symbol table, associating all possible concrete variable names with their definitions under mutually exclusive path conditions (line 20).

Makefiles have two *flavors*[3] of variable. They differ only by when their definitions are expanded. The `:=` operator used on line 18 creates a *simply-expanded* variable. Its definition is expanded at assignment time, as shown in the expansion of y on line 19. A *recursively-expanded* variable is defined with the `=` operator and is expanded at expansion time. This type of assignment is omitted from the algorithm for brevity. To evaluate it, remove the expansion of y on line 19 and instead perform it on the result of the Lookup on line 40. Recursively-expanded variables are the default flavor, and the symbol table tracks the flavor of each variable.

The append operator (`+=`), also omitted for brevity, concatenates a variable's previous definition with the new definition. It is supported by adding a call to Lookup after line 19 and concatenating the resulting definitions via the cross-product operator $\times^{\text{concat}}$. The flavor of the variable remains the same.

Rule definitions work much like variable assignment (line 21). The expressions for the rule targets, dependencies, and commands are expanded (line 22-23), since they may contain variable references and function calls. The helper function AddRules updates the rule set R using the resulting abstract values (line 24).

For `include`, Expand computes the abstract value of the expression (line 26), because the name of file can be computed at runtime. Eval is called on each file under its corresponding path condition $\hat{q}$ combined with the current path condition (lines 27–29).

Expanding expressions. Line 31 defines Expand. It takes a string expression e and performs any concatenations, variable expansions, and function calls it contains.

Makefile expressions are implicitly delimited by whitespace, forming subexpressions. Line 33 loops over each of these subexpressions of e. Concrete strings containing no expansions (line 35) are trivially converted to abstract values by constructing a single pair $(x, \top)$, i.e., this string has the same value in all paths (line 36).

Concatenations happen when variable expansions or function calls are adjacent to other subexpressions, i.e., xy (line 37). Each term is first expanded, yielding abstract values. The abstract values are concatenated by lifting the concrete string `concat` operation using the cross-product $\times^{\text{concat}}$ (line 38). This yields all possible concatenations under their corresponding path conditions.

Variable expansion (line 39) is handled using the symbol table Lookup function (line 40). Because variable names can be constructed at runtime, x is expanded before the lookup is performed and Lookup operates on this resulting abstract value. Note that this evaluation is for simply-expanded variables. For recursively-expanded variables, the resulting definition is expanded after the Lookup.

Function calls, e.g., `subst` or `filter`, take a comma-delimited list of arguments, `arglist` (line 41). In concrete Makefile evaluation, the arguments are first expanded. Similarly, `arglist` is passed to Expand (line 42). The function call is analyzed by evaluating the concrete implementation of the function on all combinations of concrete arguments under the appropriate path conditions (line 43). There is one function, `shell`, that makes a shell call. Because we are lifting the concrete operation, the shell call can still be abstracted, since each result is associated with its path condition. It is assumed, however, that the shell calls do not have side-effects between shell calls that affect the mutual exclusion of the path conditions.

Line 44 joins the abstract results of the expansion of each subexpression using the cross-product. The join(' ') operator is used to produce a whitespace-delimited expression from the expanded subexpressions.

4 IMPLEMENTATION

To evaluate the efficacy of Kmax, we implemented its static analysis algorithm in a tool that analyzes the variability of Makefiles for codebases that use the Kbuild build system. Kbuild, originally developed for the Linux kernel, is used in several low-level systems projects including BusyBox (a toolchain used in operating system installers, routers, etc), coreboot (open-source firmware supporting many hardware platforms and shipped in Chromebooks), uClibc and uClibc++ (libraries for embedded systems), and EmbToolkit (build toolchain for embedded systems).

Kbuild provides canned build rules for reserved variable names, such as `obj-y`. Programmers encode variability by populating these variables in a collection of Makefiles like the one shown in Figure 1. The names of C source files (or their corresponding `.o` object files) are added to a well-known variable, `obj-y` (lines 1 and 7). Any directories added to `obj-y` have their Makefiles recursively processed by Kbuild, paralleling the hierarchical directory structure of large codebases. Build rules (lines 7-8) are typically not needed, because they are already provided by Kbuild. Kbuild has many other special variables and features, such as `obj-m` for load-time modules. A full description can be found in its documentation[4].

Kbuild Makefiles provide an ideal testbed. There are thousands of lines of source code that encode variability for large codebases like Linux, they contain an enormous variety of real-world usage that spans the range of Makefile semantics, and they abstract away shell calls to the C compiler and other external tools that cannot be modeled easily by any analysis tool.

Several Kbuild clients also use Kconfig, a specification language for configuration options, of which Linux has over 14,000. Kconfig makes it possible to automatically generate the abstract values for

[3] https://www.gnu.org/software/make/manual/html_node/Flavors.html

[4] https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt

```
1 config USB
2         tristate "Support for Host-side USB"
3         depends on USB_ARCH_HAS_HCD
4         select NLS # for UTF-8 strings
```

Figure 2: An example Kconfig definition for a configuration option. From Linux: drivers/usb/Kconfig.

the input variables. Figure 2 is an example definition of a configuration option from the Linux kernel. Line 1 declares the option, in this case USB. Line 2 declares the type of the option. `tristate` options are like Boolean options, but they have a range of three values instead of two: `y` for enabled, `m` for enabled as a kernel module, and unset for disabled. Some configuration variables are of type `string`. The range of string values for such options is specified manually in our implementation for three variables as noted in next section.

Lines 3–4 specify constraints on this configuration option in terms of other options. The `depends on` declaration means that USB_ARCH_HAS_HCD must be enabled in order for USB to be enabled. `select` is the reverse: enabling USB forces NLS to be enabled as well. These constraints define the legal combinations of configuration options, a feature model. Our implementation uses these constraints to limit the range of input configuration options when processing Makefiles. Kconfig provides more features such as file inclusion and conditionals that can be found in its complete documentation[5].

Kmax is implemented in python. It uses the Makefile parser from pymake[6], a python port of make. For BDDs, it uses the CUDD library[7]. The static analysis algorithm and its data structures for abstract values have been implemented from scratch. Kbuild behavior is built into Kmax, so that it recognizes reserved variables such as `obj-y`. It permits file inclusion both via Makefile `include` as well as by specifying a subdirectory to `obj-y`. Makefile rules are currently ignored as they are rarely used in Kbuild Makefiles. Kmax takes a Kbuild Makefile as input and produces the final abstract state in the form of a symbol table, as well as the list of the source files and directories for all configurations for use in evaluating Kmax and comparing it with other tools.

5 EXPERIMENTAL EVALUATION

The implementation of Kmax is evaluated on the build systems of two systems projects: Linux (v3.19) and BusyBox (v1.25.0). The experiment runs Kmax on the Kbuild Makefiles of each and, from the resulting abstract state, collects the names of the C files that comprise the program along with their reaching configurations. All experiments in this and subsequent sections are performed on a workstation running a 4.2GHz 4-core processor with 16GB of RAM. The scripts to run the experiments are available online with the implementation of Kmax[8].

Not all C files in a source tree are program source files. There are helper programs, example applications, and other source files that can never be configured for inclusion in the final program. Because Kmax is the first tool to find the C files that comprise the program, we manually verify the C files that Kmax excludes against the source tree and by inspecting, when necessary, the original Makefiles to evaluate its accuracy and precision. Accuracy is measured by the number of source files correctly identified as being part of the program. Precision is measured by the number of files misidentified, where more misidentified files means less precision.

Table 2: Verification of the accuracy of Kmax on the Linux v3.19 and BusyBox v1.25.0 build systems.

	Type of C File	Linux	BusyBox
1	*Identified by Kmax*	19,651	560
2	Libraries	200	0
3	In non-Kbuild directories	524	23
4	Helper programs	215	5
5	`#included` C files	147	21
6	Examples	3	6
7	Orphaned	49	18
8	No configuration option	13	0
9	Not built with Kbuild	2	0
	TOTAL C FILES	20,804	633
	All C files in source tree	20,804	633
	Missed by Kmax	0	0

Table 2 summarizes the results of the verification. For each build system tested, it shows the number of C files identified by Kmax (row 1) followed an account of every other C file in the source tree (rows 2–9). These are then added up and compared to the actual number of C files in the source tree, found using `find` and `grep` to get all files ending with `.c`.

Linux. Linux has 1,985 Kbuild Makefiles with 29,525 lines of source. Its build process is complicated by its support for multiple architectures, x86, arm, powerpc, etc. The build system treats the different architectures like separate projects, each with its own top-level Kbuild and Kconfig entry points. To find source files for the entire Linux project, Kmax is run once for each of its 30 architectures separately.

Most architecture-specific code is abstracted away into the `arch/` directory while the rest of code is largely shared. Still, there are architecture-specific source files throughout the codebase that Kconfig associates with the a specific architecture via constraints. For instance, `ps3disk.c` is only built when `CONFIG_PS3_DISK` is enabled, an option only defined when building for powerpc. Kmax has no difficulty with this. When analyzing x86, for instance, it will not include `ps3disk.c`. It evaluates Makefiles according to the input variables, and `CONFIG_PS3_DISK` is always undefined for x86.

All input configuration options used to initialize Kmax's symbol table were automatically generated from Kconfig except for three. These three are manually-specified, because they are not of type `bool` or `tristate`: `CONFIG_WORD_SIZE` takes either 32 or 64; `BITS` also takes 32 or 64; and MMU is either `-nommu` or the empty string. All three are used in Makefiles to dynamically generate the names of C files via concatenation.

On average, Kmax takes 77.52 seconds for one architecture with a minimum of 63.28 seconds and a maximum of 357.65 seconds. The

[5] https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt
[6] https://github.com/mozilla/pymake
[7] http://vlsi.colorado.edu/~fabio/CUDD/
[8] http://github.com/paulgazz/kmax

total for all 30 architectures is 2,325.55 seconds. Table 2 details the verification of Kmax's output for Linux. Kmax finds that 19,651 C files from the source tree are referenced from the Kbuild Makefiles as part of the Linux kernel (row 1) including 200 built as libraries (row 2) compared to 20,804 total C files.

While most C files are in the kernel source directories, i.e., the Kbuild directories, there are some directories that exclusively contain tools not part of the kernel program. For example, the `scripts` directory contains the Kbuild and Kconfig tools themselves. There are 524 C files in these non-Kbuild directories (row 3).

Within the Kbuild directories, there are also many C files not meant for compilation into the kernel program. Helper programs (row 4) are stand-alone programs to assist the build process, such as `hex2hex.c`, a hex converter used for generating a header file. Some C files are included via the preprocessor `#include` in other C files (row 5) and should not be double counted. Still others are only example programs for driver developers (row 6).

Some C files are orphaned (row 7), i.e., they appear to be kernel source, but are not referenced by the Makefiles. These source files are dead code or are being staged for inclusion into the kernel. Similarly, several C files are referenced by Kbuild, but, lacking a definition of the their configuration options, cannot be compiled into the kernel by any configurations (row 8). Lastly, there were two kernel source files built by a custom Makefile rule instead of via Kbuild (row 9).

Adding the counts for all the C files identified by Kmax and those it correctly omitted, we find that every C file from the source tree is accounted for and conclude Kmax misses none.

BusyBox. We repeat the same experiment for BusyBox. BusyBox is a much simpler case compared to Linux. It has 30 Kbuild files with 840 SLoC. Kmax takes about 2.30 seconds to run on all BusyBox Makefiles. The last column of Table 2 shows the verification of Kmax's output. There are 633 total C files in the source tree, and Kmax identifies 560 from the Kbuild Makefiles (row 1).

As with the Linux kernel, BusyBox also has a `script` directory as well as others that do not contain Kbuild Makefiles (row 3). It uses stand-alone helper programs during the build (row 4), has `#included` C files (row 5), and has example user applications such as `networking/httpd_ssi.c` that are not referenced in any build files (row 6).

BusyBox's orphaned C files (row 7) are orphaned for different reasons, however. Some, like `networking/tc.c` are referenced in the Makefiles, but their controlling configuration options have been commented out of the Kconfig files, preventing their inclusion in any configuration. Several C files in the Kbuild directories lack references in the Makefiles and appear to be intentionally dead code, as all 13 have the prefix `unused_` in their filenames.

Adding up the rows we find that the total number of C files matches what is in the source tree and conclude that Kmax has missed none.

6 COMPARISON TO PREVIOUS WORK

Kmax is compared against the previous tools KBuildMiner[9] and GOLEM[10] for both correctness and performance. KBuildMiner parses Makefiles with an approximate grammar, looking for usage patterns [5]. GOLEM employs a brute force approach [12, 46]. It enables one or more features at a time and runs `make` to see which compilation units get activated.

[9]https://code.google.com/p/variability/
[10]https://www4.cs.fau.de/Research/VAMOS/

Table 3: Accuracy and precision for each tool as compared to Kmax. "All" lists the number of C files gathered for all architectures. "Missed" are those missed by each tool as compared to Kmax. "x86" is the number gathered from the run on x86 only. "Misidentified" is the number of C files from the "x86" column not configurable for x86.
†Kmax has by definition 0 missed and misidentified files.
‡The number of missed files is biased by the KBuildMiner's failure to run on 6 architectures for v3.19. v2.6.33.3 shows it is on par with GOLEM.

Tool	All	Missed	x86	Misidentified
v3.19				
Kmax	19,651	†	14,783	†
KBuildMiner	16,948	2,703‡	14,904	440
GOLEM	18,404	1,247	14,460	390
v2.6.33.3				
Kmax	12,325	†	9,044	†
KBuildMiner	10,353	1,972	8,981	134
GOLEM	10,553	1,772	8,962	129

In this experiment, each tool is run on the Linux source code for each architecture, and the names of all C files found from the Makefiles are collected from each. We compare their results using the Kmax output as a baseline, because the previous section verified the set of C files produced by Kmax.

Two version of the Linux source are tested: v3.19 and v2.6.33.3. v3.19 is used because it is the version on which the verification of Kmax was performed in the previous section. To eliminate bias due to a technical rather than fundamental limitation, we also run the experiment on an older version of Linux, v2.6.33.3. This version was chosen because a previous study testing both KBuildMiner and GOLEM shows both running successfully on that version [12].

Accuracy is measured by the difference in the total number of C files found compared to Kmax's output. To measure precision, we compare the outputs for just the x86 version of the Linux kernel source to measure the number of misidentified files, i.e., architecture-specific C files that cannot be enabled for the x86 architecture. Lastly, we compare running time.

Accuracy and Precision. Table 3 compares the C files found by Kmax with those found by KBuildMiner and GOLEM. The rows are grouped by the version of Linux tested. The first column is the tool name, and the second shows the number of C files gathered for all architectures. The Missed column shows how many C files were missing as compared to Kmax, which for both versions of Linux is well over a thousand. We note that KBuildMiner was unable to successfully process six architectures of Linux v3.19, so its results for that version are misleading. Its results on v2.6.33.3, a version on which it is known from the literature to work, shows that it is fundamentally limited, missing over 1,900 C files.

Table 4: Running time of each tool running on the x86 architecture of two Linux versions, v3.19 and v2.6.33.3. Each tool was run five times, excluding the warm-up run for KBuildMiner. The minimum, average computed by the mean, and maximum are listed in "sec" for seconds, "min" for minutes, and "hrs" for hours.

Tool	Min	Mean	Max
v3.19			
Kmax	84.03 sec	84.15 sec	84.25 sec
KBuildMiner	44.17 sec	45.00 sec	45.87 sec
GOLEM	3.41 hrs	3.42 hrs	3.43 hrs
v2.6.33.3			
Kmax	46.69 sec	46.75 sec	46.80 sec
KBuildMiner	11.82 sec	12.32 sec	12.87 sec
GOLEM	53.96 min	54.56 min	55.04 min

The last two columns show the number of C files found on the x86 run as well as the number misidentified files. To compute the misidentified files, we take the list of filenames produced by each tool and remove those also identified by Kmax. The remaining are those that can not be configured when building for x86.

Nearly all of these misidentified files are specific to another architecture. For instance, both KBuildMiner and GOLEM collect `ps3disk.c` when analyzing x86. As shown in Section 5, this C file can only be compiled for powerpc, as enforced by its configuration option. A small number of misidentified C files are dead code. For example, several C files in `drivers/acpi/acpica`, e.g., `hwtimer.c`, are controlled by a phony configuration option that is not defined in Kconfig, `ACPI_FUTURE_USAGE`. Kmax's static analysis approach enables the precision necessary for handling these situations.

KBuildMiner and GOLEM both miss thousands of C files and misidentify hundreds for the x86 architecture alone. This means that configuration-dependent bug-finders based on these tools will miss much source code compared to Kmax.

Running Time. The running time of all three tools was evaluated by running each five times for the x86 architecture of both Linux v2.6.33.3 and v3.19. Kmax and GOLEM are both python applications. KBuildMiner is written in Java and Scala, so a JVM warm-up run is used before collecting its running time.

Table 4 shows the results of the running time experiment. Each row is one tool, grouped by Linux version. The columns are the tool name followed by the minimum, mean, and maximum running times for the five runs. For v3.19, The fastest tool is KBuildMiner, taking 45 seconds. This is expected, since it only performs parsing without evaluation. Kmax takes about 85 seconds, about double the time. Note that Kmax's python implementation would likely improve in performance if ported to Java. GOLEM, on the other hand, is far slower than both tools, taking hours instead of seconds. GOLEM's long running time comes from repeatedly execute `make` for one configuration at a time. This process is time-consuming without yielding much better results than KBuildMiner.

The results are similar for v2.6.33.3. Both Kmax and KBuildMiner take under a minute, with KBuildMiner less than a quarter of the running time of Kmax. GOLEM is still prohibitively long in comparison, taking over 50 minutes. When compared to the larger v3.19 version of Linux, both KBuildMiner and GOLEM take four times times longer. Kmax, on the other hand, scales much better, taking only about twice as long.

In summary, KBuildMiner's fuzzy parsing is the fastest, while GOLEM is orders of magnitude more time-consuming than both of the other tools. Kmax is far more accurate and precise, performs very quickly on the highly-configurable Linux build system, and scales much better.

7 RELATED WORK

Analysis of Makefiles. KBuildMiner collects C files by parsing Makefiles [5]. It uses a custom grammar supporting specific usage patterns. Makex takes a similar parsing approach [33]. Dietrich et al. found it yielded only 75 percent coverage, underperforming both KBuildMiner and GOLEM [12]. After adding support for Makefile conditionals, its authors report a yield of 85%[11]. GOLEM uses a dynamic analysis approach, trying one or more configuration variables at a time to see which C files are enabled. Dietrich et al. compare GOLEM to other tools including KBuildMiner and Makex to evaluate their coverage [12].

Tamrawi et al. describe SYMake, a symbolic Makefile evaluator for use in tools such as Makefile refactorings [44]. SYMake builds a symbolic dependency graph from Makefiles for use in identifying code smells and in refactoring. It preserves string operations, variable assignments, and conditionals on the symbolic dependency graph. Like Kmax, SYMake is a static analysis of Makefile constructs. It does not, however, model configurations as paths through the Makefile, which limits its ability to perform variability-aware analyses and collect configurations. In contrast, Kmax provides abstractions for both configurations and the string domain and yields data-flow results that precisely summarize a Makefiles' variability, enabling it, for instance, to find dead code.

Adams et al. describe MAKAO, a visualization tool for Makefile dependencies. It extracts a concrete dependency graph for a single configuration [2]. While it is not variability-aware, Kmax has the potential to enhance MAKAO and such concrete tools by defining the space of legal configuration in which to sample. Such configuration space sampling approaches have been studied before [28].

Program analysis. Several program analyses use BDDs, such as BDD-based points-to analysis [7] and BDD-based software verification [8]. Xie and Aiken use BDDs to represent paths and use SAT solving to find bugs [50]. Notably it checks for bugs *after* preprocessing, and therefore in a single configuration. Kmax enables configuration-dependent bug-finders by using BDDs to represent reaching configurations.

Several existing program analysis techniques simulate multiple executions of a program with a single execution, using state splitting to minimize combinatorial explosion and avoid duplicate work. Kmax's technique for keeping multiple values of each variable while executing Makefile constructs in a single pass resembles these approaches. Tucek et al. provide an improvement to testing patched systems called *delta execution* [48]. Exploiting the fact

[11] http://www.sarahnadi.org/research/kbuildvariability

that patches are often relatively small, it uses a single execution rather than running unpatched and patched versions side-by-side, splitting only at differences. Austin and Flanagan describe an information flow analysis of JavaScript programs that tracks *faceted values*, i.e. both high and low values simultaneously, during a single execution [4]. This retains the benefits of multi-process execution, but collapses executions when raw values are the same, improving performance. Sen et al. describe MultiSE, an improvement over dynamic symbolic execution for JavaScript programs [41]. It deals with the path-explosion problem by representing variables with multiple symbolic values guarded by path constraints represented by BDDs, rather than auxiliary variables.

Christensen et al. describe a technique that statically analyzes the string values of Java programs [10]. In contrast, Kmax uses the simpler technique of exhaustively computing each possible concrete string given the range of values in the abstract variables.

Variability-aware analysis. Midtgaard et al. describe a new framework to systematically lift abstract-interpretation-based analyses to family-based analyses [30]. Abstract interpretation is a rigorous technique for constructing static analyses [11]. Kmax is a family-based static analysis approach. While not strictly abstract interpretation per se, Kmax's static approach interprets Makefile language constructs over an abstract state to perform variability-aware analysis of Kbuild Makefiles.

Kästner et al. describe a variability-aware module system that permits variability within modules, overcoming the limitations of modeling real-world variability with the traditional coarse-grained module-level variability [21]. An implementation extracts a variability model by parsing BusyBox C files and checks for type and linker errors, the latter akin to the application of configuration-dependent bug-finders to undefined function calls. There is no mention of Makefile analysis, and we imagine Kmax would be a useful tool for this and other variability-aware analyses.

Kmax's representation of variable values, by storing all possible values with their configuration, resembles previous work on variational programming. Walkingshaw et al. proposes *variational data structures*, which represents variation explicitly in the language itself, rather than requiring repeated execution of the entire program for each combination of variations [49]. Chen et al. go further and describe variational programming in general, including syntax, variational-preserving computation and other semantics, and a variational type system. [9]. Kmax's abstract variables can be thought of as variational data types that Kmax infers the values of during analysis, albeit not a language feature per se.

Thum et al. produce a comprehensive survey of software product line analysis strategies that classify them into product-based, feature-based, and family-based [47]. In this classification, Kmax is a family-based analysis, because it reasons about the space of possible configurations as a whole. It simulates multiple executions simultaneously by maintaining abstract variables that represent multiple concrete values with their configurations. Variability-aware C preprocessors use a similar approach to store and expand macros [16, 17, 20]. Kim et al. test all combinations of features in a software product line with *shared execution* [22]. Instructions are executed until variability requires a split in state, and states are merged when possible. Meinicke et al. describe variability-aware execution for Java, a dynamic analysis used to measure the configuration complexity of several programs [29]. Its *conditional values* store multiple concrete values and their configurations.

Configuration analysis. Reisner et al. use symbolic evaluation of PHP to measure configuration properties and code coverage [37]. Nguyen et al. describe variability-aware execution of PHP to test combinations of plugins efficiently [34]. Concolic execution combines symbolic and concrete execution for improved testing, specifically to create concrete test cases [18, 40].

There are several studies on Linux's build system as a software product line. Sincero et al. identified Kconfig as a feature model [43], and several publications demonstrate building feature models from Kconfig. Berger et al. compared Kconfig and another modeling language called CDL to illustrate real-world use of variability modeling [6]. She et al. built a formal hierarchy of features for Linux [42]. Dietrich et al. quantified the granularity of features in the Linux kernel [13]. Dintzner et al. tracked changes in Linux's feature model over time [14]. Tartler et al. calculated code coverage for a single configuration and maximized coverage with a minimal set of features [45]. Nadi and Holt analyzed Kbuild Makefiles to find anomalies such as unused C files [32]. Thum surveys software product line analysis techniques, categorizing methods for modeling features as well as techniques for software tools to deal with variability in software [47].

Kmax provides a precise characterization of the configurations encoded by Kbuild Makefiles, essential to family-based analysis, such as bug-finding, testing, and refactoring tools, for the Linux kernel and other tools that use Kbuild.

Partial evaluation. Partial evaluation specializes a program, e.g., for optimization, for a given set of inputs, precomputing values and unfolding all possible results [19]. The way Kmax evaluates operations on abstract values is akin to this unfolding.

8 CONCLUSION

We have introduced Kmax, a new static analysis algorithm for Kbuild Makefiles that summarizes their behavior for all configurations. It is a path-sensitive analysis, where paths are abstracted with Binary Decision Diagrams and represent the reaching configurations at each program point. The abstract state enumerates unique string values tagged with their reaching configurations. The algorithm is implemented in a tool that collects the C source files and their configurations from the Makefile-based Kbuild build system. This tool is evaluated on the Linux and BusyBox build systems and can analyze even the highly-configurable Linux build system in minutes. Compared to ad-hoc solutions, Kmax is more accurate and precise, enabling more precise downstream variability-aware tools. Additionally, Kmax performs better or with little overhead, is more scalable, and can find unreachable code. Future work includes using Kmax on more build systems and porting it to other build languages as well as incorporating its results into other variability-aware analysis tools.

ACKNOWLEDGMENTS

I would like to thank the all the reviewers for helping make this a better paper and Shiyi Wei for his indispensable advice.

REFERENCES

[1] 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 (ASE '14)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2642937.2642990

[2] B. Adams, H. Tromp, K. de Schutter, and W. de Meuter. 2007. MAKAO. In *2007 IEEE International Conference on Software Maintenance*. 517–518. https://doi.org/10.1109/ICSM.2007.4362678

[3] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated.

[4] Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 165–178. https://doi.org/10.1145/2103656.2103677

[5] Thorsten Berger, Steven She, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. *Feature-to-Code Mapping in Two Large Product Lines*. Technical Report. University of Leipzig (Germany), University of Waterloo (Canada), IT University of Copenhagen (Denmark). http://informatik.uni-leipzig.de/~berger/tr/2010-berger.pdf

[6] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2010. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/1858996.1859010

[7] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. 2003. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Inplementation*. ACM Press, 103–114. https://doi.org/10.1145/781131.781144

[8] Dirk Beyer and Andreas Stahlbauer. 2014. BDD-based software verification. *International Journal on Software Tools for Technology Transfer* 16, 5 (2014), 507–518. https://doi.org/10.1007/s10009-014-0334-1

[9] Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2016. A Calculus for Variational Programming. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. 6:1–6:28. https://doi.org/10.4230/LIPIcs.ECOOP.2016.6

[10] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Proceedings of the 10th International Conference on Static Analysis (SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 1–18. http://dl.acm.org/citation.cfm?id=1760267.1760269

[11] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. ACM, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[12] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikshat, 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*. 21–30. http://doi.acm.org/10.1145/2362536.2362544

[13] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikshat, and Daniel Lohmann. 2012. Understanding linux feature distribution. In *Proceedings of the 2012 Workshop on Modularity in Systems Software*. 15–20. http://doi.acm.org/10.1145/2162024.2162030

[14] Nicolas Dintzner, Arie Van Deursen, and Martin Pinzger. 2013. 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)*. ACM, New York, NY, USA, Article 22, 8 pages. https://doi.org/10.1145/2556624.2556631

[15] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2017. Analysing the Linux kernel feature model changes using FMDiff. *Software & Systems Modeling* 16, 1 (2017), 55–76. https://doi.org/10.1007/s10270-015-0472-2

[16] Alejandra Garrido and Ralph Johnson. 2005. Analyzing Multiple Configurations of a C Program. In *Proceedings of the 21st ICSM*. 379–388. http://dx.doi.org/10.1109/ICSM.2005.23

[17] Paul Gazzillo and Robert Grimm. 2012. SuperC: parsing all of C by taming the preprocessor. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 323–334. http://doi.acm.org/10.1145/2254064.2254103

[18] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[19] Neil D. Jones. 1996. An Introduction to Partial Evaluation. *ACM Comput. Surv.* 28, 3 (Sept. 1996), 480–503. https://doi.org/10.1145/243439.243447

[20] Christian Kästner et al. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the 26th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. 805–824. http://dx.doi.org/10.1145/2048066.2048128

[21] Christian Kästner et al. 2012. A variability-aware module system. In *Proceedings of the 27th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. 773–792. http://doi.acm.org/10.1145/2384616.2384673

[22] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don S. Batory. 2012. Shared Execution for Efficiently Testing Product Lines. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*. 221–230. https://doi.org/10.1109/ISSRE.2012.23

[23] Jörg Liebig et al. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32th International Conference on Software Engineering*. 105–114. http://doi.acm.org/10.1145/1806799.1806819

[24] Jörg Liebig et al. 2011. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development*. 191–202. http://doi.acm.org/10.1145/1960275.1960299

[25] 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 (ESEC/FSE 2013)*. ACM, New York, NY, USA, 81–91. https://doi.org/10.1145/2491411.2491437

[26] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. *Evolution of the Linux Kernel Variability Model*. Springer Berlin Heidelberg, Berlin, Heidelberg, 136–150. https://doi.org/10.1007/978-3-642-15579-6_10

[27] Bill McCloskey and Eric Brewer. 2005. ASTEC: A New Approach to Refactoring C. In *Proceedings of the 10th European Software Engineering Conference*. 21–30. http://dx.doi.org/10.1145/1081706.1081712

[28] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM Press, New York, NY, 643–654. https://doi.org/10.1145/2884781.2884793

[29] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-configurable Systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/2970276.2970322

[30] Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2015. Systematic Derivation of Correct Variability-aware Program Analyses. *Sci. Comput. Program.* 105, C (July 2015), 145–170. https://doi.org/10.1016/j.scico.2015.04.005

[31] 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. https://doi.org/10.1109/TSE.2015.2415793

[32] Sarah Nadi and Ric Holt. 2011. Make It or Break It: Mining Anomalies from Linux Kbuild. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering (WCRE '11)*. IEEE Computer Society, Washington, DC, USA, 315–324. https://doi.org/10.1109/WCRE.2011.46

[33] Sarah Nadi and Ric Holt. 2012. Mining Kbuild to Detect Variability Anomalies in Linux. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*. IEEE Computer Society, Washington, DC, USA, 107–116. https://doi.org/10.1109/CSMR.2012.21

[34] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-aware Execution for Testing Plugin-based Web Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 907–918. https://doi.org/10.1145/2568225.2568300

[35] Yoann Padioleau et al. 2010. Documenting and automating collateral evolutions in linux device drivers. 247–260. http://doi.acm.org/10.1145/1352592.1352618

[36] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. 2006. Understanding Collateral Evolution in Linux Device Drivers. In *Proceedings of the 1st European Conference on Computer Systems*. 59–71. http://dx.doi.org/10.1145/1217935.1217942

[37] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. 2010. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 445–454. https://doi.org/10.1145/1806799.1806864

[38] Valentin Rothberg, Nicolas Dintzner, Andreas Ziegler, and Daniel Lohmann. 2016. Feature Models in Linux: From Symbols to Semantics. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '16)*. ACM, New York, NY, USA, 65–72. https://doi.org/10.1145/2866614.2866624

[39] Sandro Schulze et al. 2011. Analyzing the Effect of Preprocessor Annotations on Code Clones. In *Proceedings of the 11th IEEE International Workshop on Source Code Analysis and Manipulation*. 115–124. http://dx.doi.org/10.1109/SCAM.2011.12

[40] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[41] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 842–853. https://doi.org/10.1145/2786805.2786830

[42] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 461–470. https://doi.org/10.1145/1985793.1985856

[43] 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 (SPLC-OSSPL)*. 134–140.

[44] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen. 2012. Build code analysis with symbolic evaluation. In *Software Engineering (ICSE), 2012 34th International Conference on*. 650–660. https://doi.org/10.1109/ICSE.2012.6227152

[45] Reinhard Tartler et al. 2011. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review* 45, 3 (Dec. 2011), 10–14. http://dx.doi.org/10.1145/2094091.2094095

[46] Reinhard Tartler et al. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proceedings of the 6th European Conference on Computer Systems*. 47–60. http://dx.doi.org/10.1145/1966445.1966451

[47] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1, Article 6 (June 2014), 45 pages. https://doi.org/10.1145/2580950

[48] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. 2009. Efficient Online Validation with Delta Execution. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 193–204. https://doi.org/10.1145/1508244.1508267

[49] Eric Walkingshaw, Christian KÃďstner, Martin Erwig, Sven Apel, and Eric Bodden. 2014. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *In Proceedings of the ACM Symposium on New Ideas in Programming and Reflections on Software (Onward*. ACM, 213–226.

[50] Yichen Xie and Alex Aiken. 2005. Scalable Error Detection Using Boolean Satisfiability. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 351–363. https://doi.org/10.1145/1040305.1040334