# Kmax: Analyzing the Linux Build System

NYU CS Technical Report TR2015-976

Paul Gazzillo
New York University
gazzillo@cs.nyu.edu

## ABSTRACT

Large-scale C software like Linux needs software engineering tools. But such codebases are software product families, with complex build systems that tailor the software with myriad features. This variability management is a challenge for tools, because they need awareness of variability to process all software product lines within the family. With over 14,000 features, processing all of Linux's product lines is infeasible by brute force, and current solutions employ incomplete heuristics. But having the complete set of compilation units with precise variability information is key to static tools such a bug-finders, which could miss critical bugs, and refactoring tools, since behavior-preservation requires a complete view of the software project. Kmax is a new tool for the Linux build system that extracts all compilation units with precise variability information. It processes build system files with a variability-aware `make` evaluator that stores variables in a conditional symbol table and hoists conditionals around complete statements, while tracking variability information as presence conditions. Kmax is evaluated empirically for correctness and completeness on the Linux kernel. Kmax is compared to previous work for correctness and running time, demonstrating that a complete solution's added complexity incurs only minor latency compared to the incomplete heuristic solutions.

## Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software configuration management*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Patterns, Modules, packages*

## General Terms

Systems, Language, Software Engineering

## Keywords

Kmax, Linux, C, Kbuild, Kconfig, Makefiles, Build Systems

## 1. INTRODUCTION

As software systems become larger, automated software engineering tools such as source code browsers, bug finders, and automated refactorings, become more important. Larger systems are more vulnerable to bugs, and modifications to the codebase are more difficult to verify by hand due to the larger number of interactions between features of the system. C is the language of choice for many common large-scale software systems, including the Linux kernel, the Apache web server, and the GNU compiler collection, all of which are used in critical computing systems. One facet of large-scale software development is variability management, with which software systems are tailored to a specific use by enabling features at build-time. With such variability, a codebase encompasses a family of customized software product lines, which share portions of the source code and features. Variability amplifies the difficulty of creating and using automated software tools, because such tools need to be aware of the variability in order to operate on all product lines in the software family. Worse still, variability introduces new classes of bugs. Abal et al found bugs resulting from the interactions and dependencies between features of the Linux kernel, but lacking automated tools, found them by manually examining patches send to the Linux kernel mailing list [3].

New languages and formalisms for describing variability promise safety and the easier application of software engineering tools [19, 16, 25], but until such variability tools are widespread, an abundance of critical C software remains that uses ad-hoc techniques for variability. In our previous work, SuperC, we preprocess and parse all variations of C source files in the Linux kernel, which uses the preprocessor to implement variability within source files [14]. While this provides the foundation for variability-aware tool implementation for individual source files, large C programs are comprised of potentially thousands of compilation units, i.e., C files compiled separately and linked to form the final program. The Linux kernel v3.19, for instance, contains over 20,000 compilation units, but only a subset of these compilation units are used for a single software product line, depending on the selected features.

Being able to extract all compilation units and their variability information is crucial for software engineering tools. For instance, C function calls can cross compilation unit boundaries, only being referenced by an `extern` declaration. Without knowing the complete set of compilation units that may be linked together, static analyses cannot find all callees. Bug-checking in particular is limited without variability-awareness. Chou et al shows that static checkers find bugs in the Linux kernel [9], but they only operate on one software product line. Families of product lines harbor untestable bugs, since it is not feasible to check every possible combination of features separately. Variability-awareness enables tools to operate across sofware product lines. For instance, know-

ing which compilation units are linked under which combinations of features can help root out linker errors without having to build and link every possible variation of the software. Additionally, previous work on translating Linux's extensive C preprocessor use to a safer alternative, such as aspects [4] or to the ASTEC preprocessor [19], depends on a complete view of the kernel source.

This paper focuses on the Linux build system due to its size, complexity, and prevalence. Several software projects also use Linux's build system tools to manage variability, including the BusyBox toolkit [8] and the uClibc library [26], and Kmax's approach applies generally to any build system language that uses conditionals to implement variability. The Linux build system is a relevant target for Kmax, because Linux is a frequent object of study for researchers, yet it is difficult to extract variability information from its build system. For instance, Liebig et al computes statistics on variability metrics in the Linux kernel [18]. But that study along with others, including including this author's previous work, use an incomplete set of compilation units to experiment on the 2.6.33.3 version Linux [15, 14]. All three report using 7,665 or fewer units while there are 9,344, which is off by more than 15%. At best, this leads to incomplete data. At worst, static analysis tools get a incomplete view of the kernel, missing critical problems in the source code. These incomplete studies can be traced to back to a single tool, KBuildMiner.

Using a fuzzy parser for Linux Makefiles, KBuildMiner collects compilation units by looking for usage patterns [6]. This approach misses the mark, because some compilation unit names are defined by concatenation and function calls, which requires evaluating the `make` language. Furthermore, some Kbuild files need to be hand-modified to fit the syntax recognized by the parser. Aware of the limitations of KBuildMiner, Dietrich et al sought to improve the state of build system analysis with GOLEM. GOLEM enables one or more features at a time and runs `make` to see which compilation units get activated [10]. While this semi-brute-force approach successfully avoids having to try all combinations of features, its heuristic approach falls short.

This paper introduces Kmax. Kmax extracts all compilation units and their variability information without using heuristics. At its core is its variability-preserving `make` evaluator, that records all possible compilation units that comprise any software product line, and the feature selections that lead to these compilation units. In addition to evaluating most of the `make` language, it employs three key techniques: (1) it maintains a conditional symbol table with all possible variable definitions, (2) it evaluates all branches of conditionals blocks, and (3) it *hoists* conditionals around statements that contain conditionals. During processing, Kmax's `make` evaluator tracks the combinations of features as a boolean expression, called a *presence condition*. By tracking the presence condition during evaluation, it can discover the combination of features that enables each compilation unit.

The contributions of this paper are as follows:

1. Algorithms to find selectable features and evaluate a subset of the `make` language across software product lines,
2. A new tool, Kmax, that implements the algorithms to extract compilation units and their presence conditions from the Linux build system, and
3. Empirical evaluation of Kmax's correctness and performance with a comparison to previous work.

Kmax is available at `http://cs.nyu.edu/~gazzillo/kmax.html`.

# 2. PROBLEM AND SOLUTION APPROACH

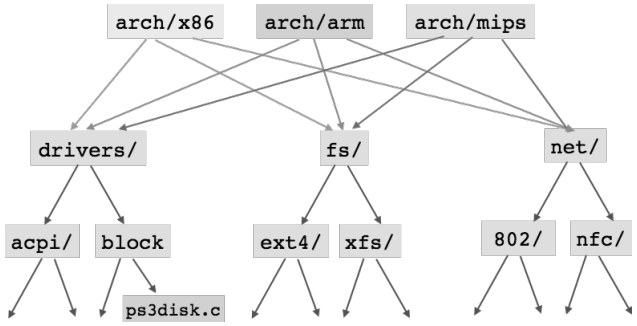


Figure 1: Hierarchy of source code in the Linux kernel codebase. Each architecture directory is a separate root of the source tree and includes the rest of the common codebase. Some compilation units appear in the common codebase, but can only be enabled when building for one architecture, e.g., `ps3disk.c` can only be enabled in `arch/arm`

Kmax's challenges stem from both the difficulty of evaluating the `make` language in the presence of variability and the peculiarities of the Linux source tree's organization. Its build system uses two specification languages, Kconfig to define features and their constraints and Kbuild, a `make`-based language that describes how features control the build process. To build one product line from the Linux codebase, the user first selects the architecture. Then the user selects the features to include in the kernel, such as drivers and file systems. Once configured, the build process is typical of C programs, using `make` to run the Kbuild files, compiling and linking the compilation units according to the selected features. The process Kmax uses to extract the compilation units and their variability information from the build system mirrors the build process. Given an architecture, Kmax first processes the Kconfig files to find the domain of features. Then, using its variability-preserving `make` evaluator on the Kbuild Makefiles, it extracts the compilation units while recording their presence conditions. The challenges to this process include handling architecture-specific source code, finding selectable features, the particulars of Kbuild, and evaluating `make` across software product lines. These challenges and Kmax's solution approach are detailed below.

## 2.1 Architecture-Specific Source Code

The Linux kernel source code is hierarchical. Top-level directories define major subsystems, such as `net/` for networking and `drivers/`, and nest related code in subdirectories, for example, `net/ethernet` and `drivers/video`. While the codebase contains source code that is mostly shared by all software product lines, each architecture serves as the root of its own hierarchy. Figure 1 illustrates this with a forest. At the roots of the trees are the architecture-specific source code directories. These directories roughly form the hardware abstract layer (HAL), defining macros, functions, types, and include paths that the rest of the codebase uses. Beneath the HAL are the top-level directories, to which each architecture points to form the rest of its hiearchy. There are two consequences to this structure. Firstly, static analyses only make sense only after a HAL is selected and should operate on one architecture at at time. Secondly, not all compilation units are accessible to each architecture's product lines. As Figure 1 also shows, finding these compilation units is not straightforward. The `drivers/block/ps3disk.c` compilation unit is part of every architecture's hierarchy. But because of the constraints on features,

| Metric | Count |
|---|---|
| Total compilation units | 21,158 |
| Shared compilation units | 13,881 |
| Architecture-specific units in `arch/` directories | 5,973 |
| Architecture-specific units in common directories | 1,304 |
| Total features | 14,636 |
| Shared | 9,658 |
| Architecture-specific | 4,978 |
| Per-architecture compilation units | |
| Minimum | 13,906 |
| Maximum | 15,976 |
| Per-architecture features | |
| Minimum | 9,684 |
| Maximum | 11,232 |

Table 1: Linux v3.19 build system metrics broken out by architecture-sharing.

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

(a) A feature definition. From drivers/usb/Kconfig.

```
1 if USB
2 source "drivers/usb/storage/Kconfig"
3 endif
```

(b) Kconfig's if and source commands. From drivers/usb/Kconfig. Edited to show one out of nine includes.

```
1 config BLK_DEV_IDE_ICSIDE
2         tristate "ICS IDE interface support"
3         depends on ARM && ARCH_ACORN
```

(c) A feature unselectable in most architectures. From drivers/ide/Kconfig.

Figure 2: Examples of Kconfig from Linux v3.19.

defined in Kconfig files, only software product lines built for the `arm` architecture can ever enable this compilation unit. Kmax first employs its Selectable algorithm to find architecture-specific features. Then Kmax only allows selectable features to be enabled when extracting compilation units from Kbuild. Table 1, generated by Kmax, illustrates how much architectures share with each other. While most architecture-specific compilation units live in the `arch/` directory, more than a thousand are in the common source code directories. As for features, nearly a third are architecture specific.

## 2.2 Finding Selectable Features

Kconfig files use a domain-specific specification language to define features and their constraints. Figure 2 shows several representative examples from Linux. (All examples in this paper come from Linux v3.19). Example (a) defines the `USB` feature that enables Universal Serial Bus (USB) support. Line 1 is the variable declaration, while line 2 gives `USB` its type, `tristate`. Tristate variables can be set to one of three values, `y` for inclusion the compiled kernel,

```
1 obj-$(CONFIG_USB_UAS) += uas.o
2 obj-$(CONFIG_USB_STORAGE) += usb-storage.o
3 usb-storage-y := scsiglue.o protocol.o transport.o usb.o
4
5 obj-$(CONFIG_USB_STORAGE) += storage/
```

Figure 3: Snippets of Kbuild from Linux v3.19. Lines 1–3 are from drivers/usb/storage/Makefile, line 5 from drivers/usb/Makefile.

`m` for inclusion as a loadable kernel module, and an empty string for exclusion from the kernel. Other types include boolean, which is tristate without the `m`, string, and number. The latter two take constants of their respective types, and they can be used in boolean expressions with relational operators. The text after `tristate` on line 2 is displayed to the user during interactive feature selection.

Kconfig provides three ways to specify constraints between features. The `depends on` keyword on line 3 of example (a) creates a *direct dependency* on `USB_ARCH_HAS_HCD`. USB support can only be enabled if `USB_ARCH_HAS_HCD` is also enabled. The dependency can be any boolean expression of features. Another way to make a dependency is with the `select` keyword, as shown on line 4. This *reverse dependency* forces `NLS` to be enabled when `USB` is enabled, regardless of `NLS`'s other dependencies. Example (b) shows the last way to create a dependency, with an `if`/`endif` block. Every feature defined in the block on lines 1–3 gets a direct dependency on `USB`. The `source` statement on line 2 includes another Kconfig file, which is used to form the hierarchy of Kconfig files.

Example (c) shows an architecture-specific variable defined in the shared part of the Kconfig hierarchy. `BLK_DEV_IDE_ICSIDE` is defined for all architectures, but can only be enabled for ARM because of its direct dependence on the `ARM` feature. For example, x86's Kconfig files never define `ARM`, making it an *unreachable* feature. In contrast, `BLK_DEV_IDE_ICSIDE` is reachable, but its dependencies prevent it from ever being enabled when building for x86, making it *unselectable*. A feature is selectable only if two conditions hold: (1) it is reachable and (2) any dependencies are also selectable. Kmax uses Linux's own parser for the Kconfig files, which yields a in-memory representation of the features and their constraints as boolean expressions. The Selectable algorithm finds the selectable features for an architecture. As Table 1 shows, out of 14,636 features in Linux v3.19, only between 9,684 and 11,232 are selectable for any given architecture.

## 2.3 The Particulars of Kbuild

Compilation units are defined in Kbuild using Makefile syntax. Their names are added to Kbuild's reserved variables `obj-y` for built-ins and `obj-m` for dynamically-loadable modules. The build system later uses these lists to compile and link the kernel binaries. Since enabled tristate features are set to `y` or `m`, Kbuild files make use of a common pattern where the `obj-` prefix is concatenated with the value of the feature. Figure 3 is an example of this pattern. On line 1, `uas.c` is only compiled if the `USB_UAS` feature is enabled with `y` or `m`. In Makefile syntax, `$(CONFIG_USB_UAS)` expands to the value of the feature, which is given the `CONFIG_` prefix as a de facto namespace. Adjacent strings get concatenated, requiring no special operator. When `USB_UAS` is set to `y`, expansion and concatenation yield the string `obj-y`, while the `+=` operator appends `uas.o` to the existing definition of the `obj-y` variable, adding it to the list of built-in compilation units. When disabled, `USB_UAS` expands to the empty string, adding the compilation unit to the variable `obj-`, which is ignored by Kbuild. This pattern makes clear which feature controls a compilation unit, only compiling it when the feature is

```
1 ifdef CONFIG_NO_BOOTMEM
2        obj-y += nobootmem.o
3 else
4        obj-y += bootmem.o
5 endif
```

(a) Makefile conditionals create mutually-exclusive compilation units. From mm/Makefile.

```
1 obj-$(CONFIG_SMP) += smp.o
2
3 # after hoisting
4 ifeq (CONFIG_SMP, y)
5 obj-y += smp.o
6 endif
7 ifndef CONFIG_SMP
8 obj- += smp.o
9 endif
```

(b) Using Kbuild's reserved obj-y variable with feature variable expansion. From kernel/Makefile.

```
1 cacheops-$(CONFIG_CPU_SH2) := cache-sh2.o
2 cacheops-$(CONFIG_CPU_SH2A) := cache-sh2a.o
3 cacheops-$(CONFIG_CPU_SH3) := cache-sh3.o
4 # three more reassignments
5 obj-y += $(cacheops-y)
```

(c) Variable assignment creates mutually exclusive compilation units. From arch/sh/mm/Makefile.

```
1 # From arch/x86/Makefile
2 ifeq ($(CONFIG_X86_32),y)
3        BITS := 32
4 else
5        BITS := 64
6 endif
7
8 obj-$(CONFIG_X86_LOCAL_APIC) += probe_$(BITS).o
```

(d) Compilation unit names can be generated from Makefile variables. From arch/x86/kernel/apic/Makefile.

Figure 4: Examples from Linux v3.19 of the challenges of evaluating Kbuild.

enabled.

Line 2 is an example of a *composite* compilation unit. If a compilation unit, such as `usb-storage.o`, has no corresponding C file, the Kbuild evaluator looks for a variable with the name of the compilation unit plus a `-y` or `-objs` suffix. In this case, `usb-storage-y` on line 3 defines the constituent compilation units, which can themselves be composite. As with `obj-`, a composite's variable name may be concatenated with a feature to conditionally include compilation units.

Line 5 adds a subdirectory name to `obj-y` or `obj-m` instead of a compilation unit. The Kbuild evaluator enters these subdirectories to find more compilation units, which is how the Kbuild hierarchy is formed. Each subdirectory's compilation units are linked into `builtin.o` or `.ko` files for modules. Once finished with the subdirectory, Kbuild replaces `storage/` with `storage/builtin.o` for linking into the parent directory's own `builtin.o`. The `subdir-y` variable may also be used to explicitly add subdirectory names for Kbuild to traverse.

```
1 # From net/ipv6/Makefile.
2 obj-$(subst m,y,$(CONFIG_IPV6)) +=
      inet6_hashtables.o
3
4 # From arch/s390/Makefile.
5 head-y += arch/s390/kernel/$(if $(CONFIG_64BIT)
      ,head64.o,head31.o)
6
7 # From arch/arm/Makefile.
8 machdirs := $(patsubst %,arch/arm/mach-%/,$(
      machine-y))
```

(e) Makefiles can use functions when expanding features.

```
1 obj-$(CONFIG_BLK_DEV_IDE_ICSIDE) += icside.o
```

(f) Some compilation units depend on architecture-specific features. From drivers/ide/Makefile.

Figure 4: More examples from Linux v3.19 of the challenges of evaluating Kbuild.

## 2.4 Challenges to Evaluating make

Even though they frequently use common patterns, Kbuild files have the full power of the `make` language features available to use. Figure 4 contains examples that illustrate Kbuild usage. The first two examples show how Kbuild files make certains combinations of features mutually exclusive, the next two show variable expansion and functions used while defining compilation units, and the last is an example of an architecture-specific compilation unit.

Example (a) is a tests for the feature named `CONFIG_NO_BOOTMEM` and compiles one of `nobootmem.o` or `bootmem.o`, but never both. Kmax first evaluates the conditional expression on line 1 to find the conditions needed to enter the if-branch. It then evaluates the statements in both branches on lines 2 and 4, storing both definitions of `obj-y` in its conditional symbol table. Example (b) shows a feature `SMP` concatenated with the `obj-` to conditionally compile `smp.o` on line 1. When features or other Makefile variables that have multiple definitions are expanded, it is an implicit conditional, since each definition has a condition, called a presence condition, under which it is expanded. Kmax handles multiply-defined variables by expanding all their definitions to a conditional and hoisting it around the statement. Lines 3–9 show the conceptual result of hoisting, although Kmax does not explicitly create a conditional block.

Example (c) shows how variable reassignment can implicitly create mutually exclusive feature combinations. Because `cacheops-y` is reassigned repeatedly on lines 1–3, only one of the named compilation units can appear in any single software product line. Kmax creates an entry for each possible definition of `cacheops-y` along with a boolean expression representing the conditions under which the definition is possible.

Example (d) shows a case where a compilation unit's name is constructed by concatenation with the value of a variable. `BITS` is a global variable, defined in a top-level Makefile, that expands to either 32 or 64 depending on a feature as shown on lines 2–6. On line 8, Kmax expands both definitions a conditional, hoists the implicit conditional around the entire assignment statement, and as with the the conditional in example (a) stores both compilation unit names.

Example (e) shows function calls used while defining compilation units. Line 2 uses the substitute function to force the compilation unit to be built-in, instead of a module. Line 5 uses the conditional function to decide between compilation units. And Line 8

**Algorithm 1** Find selectable features.

```
 1: procedure Selectable(v, C)
 2:     procedure Evaluate(e)
 3:         if e = l ∧ r then
 4:             return Evaluate(l) ∧ Evaluate(r)
 5:         else if e = l ∨ r then
 6:             return Evaluate(l) ∨ Evaluate(r)
 7:         else if e = w, for config variable w then
 8:             return Selectable(w, C)
 9:         else if e = ¬w, for config variable w then
10:             return true
11:         end if
12:     end procedure
13:     if v ∉ C then
14:         ▷ Unreachable variables never selectable
15:         return false
16:     else if v ∈ C and v.direct = v.reverse = ∅ then
17:         ▷ Non-dependent variables always selectable
18:         return true
19:     else ▷ v is reachable and has dependencies.
20:         ▷ Check v's dependencies.
21:         return Evaluate(v.direct ∨ v.reverse)
22:     end if
23: end procedure
```

uses pattern substitution to generate a list of directories from arm machine names. These cases in particular make it difficult to collect compilation units without doing some evaluation. As with variable expansion, any feautres used in function arguments are expanded to a conditional and hoisted around the function calls. After hoisting, the functions can be evaluated normally under each resulting presence condition.

Example (f) shows the architecture-specific feature from Figure 2c being used to control a compilation unit. Because this feature is only available when compiling for ARM, Kmax takes a set of selectable configuration variables for each architecture before evaluating Kbuild Makefiles.

# 3. ALGORITHMS

The core of Kmax's solution comprises the Selectable algorithm that finds features available to a given architecture and a `make` language evaluator the collects all compilation units, recording their enabling features as boolean expressions. This section describes these algorithms in detail.

## 3.1 Selectable Features

Kmax finds selectable features by excluding those that depend only on other unreachable or unselectable features. Kmax employs Linux's own Kconfig parser, which produces a in-memory list of features and symbolic boolean expressions for their dependencies. It then uses the Selectable algorithm on each one, which returns true if selectable.

Algorithm 1 defines Selectable, which takes a feature name $v$ and the list of features $C$ produced by the Kconfig parser. Lines 13–15 check whether $v$ is reachable by checking it against the list of parsed features. Unreachable features are never selectable. Lines 16–18 look at features with no dependencies. Such variables are always selectable, since there are no other features constraining them. Lines 19–21 check all other features, i.e., those that are reachable, but have dependencies. Evaluate determines whether such a variable is selectable by examining its dependencies. Since either a direct or reverse dependency can allow a variable to be enabled, their expressions are first ORed. Lines 2–12 define Evaluate, which computes the given boolean expression $e$, recursively evaluating its subexpressions and any other features used. Lines 3–4 handle an AND operator by ANDing the results from checking the subexpressions for selectability. Only when both subexpressions allow selection will the expression be selectable. Lines 5–6 handle an OR operator by ORing the subexpression. Only one subexpression needs to be true for selectability. Lines 7–8 recursively check features used in the expression by recursively calling Selectable. This call is optimized by memoizing the return value for features previously evaluated.

Evaluating selectability mirrors evaluating boolean expressions, except for negation. To see why this is, take a feature `VAR` that depends on ¬`DEP`. If `DEP` is unselectable, then `VAR` is selectable, because of the negation. If instead `DEP` is selectable, using boolean negation would force the `VAR` to be unselectable. This would be incorrect, because `VAR` can still be enabled when `DEP` is disabled. Thus negation gives no information about selectability, so lines 9–10 always return true so as not to incorrectly limit selectability. The Selectable algorithm has a complementary Unselectable algorithm that returns true when a feature cannot be selected. This algorithm differs only in the grayed sections of Algorithm 1, namely the boolean operators and the true and false constants. Swapping AND with OR and true with false yields the complementary algorithm. The resulting sets of selectable and unselectable features are also complementary.

## 3.2 Evaluating the make Language

The selectable features are fed to Kmax's `make` evaluator, which evaluates Kbuild files across all software product lines. To acheive this, Kmax uses a conditional symbol table that holds all definitions of the `make` variables it encounters, enters and evaluates all branches of conditionals, and hoists conditionals that appear within statements to evaluate all possible complete statements. Because the Linux build system keeps compilation units in the `obj-y` and `obj-m`, Kmax finds all compilation units by inspecting the contents of the conditional symbol table. To record the features that control each compilation unit, Kmax's evaluator tracks the presence condition, a boolean expression of features, at each point in the Kbuild file, saving the presence condition when encountering a new compilation unit. The following describes the conditional symbol table, hoisting, and the evaluation algorithm.

A conditional symbol table maps a variable name to a list of $(d, c)$ tuples, where $d$ is a definition and $c$ is a presence condition representing a boolean expression of features. The conditional symbol table is initialized to contain all selectable tristate and boolean features. A tristate feature is represented as a Makefile variable $v$ and is initialized to

$$T(v) \leftarrow \{\,(\text{"y"}, v = \text{"y"}), (\text{"m"}, v = \text{"m"}), (\text{""}, \neg\,\text{defined}(v))\,\}$$

These initial conditions are tautologies, i.e., $v$ expands to y when $v$ = "y" is true and it expands to nothing when $v$ is undefined. Once expanded, however, these initial conditions ensure variability information is carried along in presence conditions in subsequent evaluation. For instance, the following variable definition involves the expansion of a second variable in order to determine the name of variable being assigned:

```
obj-$(CONFIG_USB_UAS) += uas.o
```

Because `CONFIG_USB_UAS` has multiple definitions, the evaluator expands all possible definitions, using a conditional block to preserve the presence conditions of the expanded variable definitions:

```
obj-
ifeq (CONFIG_USB_UAS, y)
  y
else
  # empty
endif
+= uas.o
```

But such a statement with an embedded conditional is not readily able to be evaluated. To handle conditionals within statements, Kmax hoists the conditional around the entired statements, yielding two complete variable assignments. Hoisting takes every possible combination of conditionals that appears within a statement and makes each complete statement explicit:

```
ifeq (CONFIG_USB_UAS, y)
  obj-y += uas.o
else
  obj- += uas.o
endif
```

Hoisting leaves a conditional block surrounding two assignment statements. Kmax's evaluator handles conditionals by entering and evaluating the contents of each branch, while tracking the presence condition that controls each statement. After evaluating the above example, the symbol table gets four new entries, two for `obj-y` and two for `obj-`, since both variable names are possible and each has two possible definitions:

$$T(\text{“obj-y”}) \leftarrow \{(\texttt{"uas.o"}, \text{CONFIG\_USB\_UAS=y}),\ (\texttt{""}, \neg\,\text{defined(CONFIG\_USB\_UAS)}\}$$
$$T(\text{“obj-”}) \leftarrow \{(\texttt{""}, \text{CONFIG\_USB\_UAS=y}),\ (\texttt{"uas.o"}, \neg\,\text{defined(CONFIG\_USB\_UAS)}\}$$

Note that the symbol table's entries record not only contents of `obj-y`, but also the features that lead to it.

Algorithm 2 is the pseudo-code for Kmax's evaluator. Statements takes a list of parsed statements $S$, a presence condition $p$, and a conditional symbol table $T$. It supports three kinds of statements: conditionals, variable assignment, and includes. Lines 3–7 handle conditionals by first expanding any variables or function calls in its conditional expression with a call to the Expands procedure on line 5. This returns a list of $(e, c)$ tuples where $e$ is an expansion of the expression and $c$ is the presence condition of the expansion. Each conditional expression is conjoined with its presence condition, and the resulting conjunctions are unioned to produce $c_{\text{if}}$, which represents all the ways the conditional block's expression can be made true and the if-branch entered. Line 6 enters the if-branch with an updated presence condition, recursively calling Statements on the branch's statements. The else-branch is similarly evaluated on line 7, but with the negation of the condition that enters the if-branch as the presence condition.

Variable assignment is handled on lines 8–16. Line 9 first expands the variable name, because the variable name itself can contain variable expansions and function calls. Likewise, the definition is expanded. The nested for loops on lines 11–12 try each combination of variable name and definition that resulted from expanding them. So when a feature is used to add a new compilation unit as in `obj-$(CONFIG_USB_UAS) += uas.o`, both `obj-y` and `obj-` get assigned. The conditions of the assignment are stored with the definition in the conditional symbol table. Line 14 updates each variable name's entry in $T$ with a new definition under the combined presence condition that yielded the name and definition combination.

This variable assignment is a simplification of what the Kmax tool actually does, because Makefiles have several assignment operators, each with a different meaning. Variables come in two *flavors* [1]. "=" creates a *recursively-expanded* variable. Its definition gets expanded at the time of the assignment. In contrast, ":=" creates a *simply-expanded* variable whose definition is not expanded until call-time. "?=" only updates the definition if the variable isn't already defined. Variable definitions can be appended to with the

**Algorithm 2** Evaluate the statements of a Makefile.

```
1: procedure Statements(S, p, T)
2:     for s ∈ S do
3:         if s is a conditional (e, S_if, S_else) then
4:             ▷ Compute all possible ways to enter the if-branch.
5:             c_if ← ⋁ e' ∧ c for (e', c) ∈ Expand(e, p, T)
6:             Statements(S_if, p ∧ c_if, T)
7:             Statements(S_else, p ∧ ¬c_if, T)
8:         else if s is a variable assignment (e_v, e_d) then
9:             V ←Expand(e_v, p, T)
10:            D ←Expand(e_d, p, T)
11:            for (v, c_v) ∈ V do
12:                for (d, c_d) ∈ D do
13:                    ▷ Add each possible definition to T.
14:                    T(v) ← T(v) ∪ {(d, p ∧ c_v ∧ c_d)}
15:                end for
16:            end for
17:        else if s is an include of e then
18:            I ←Expand(e, p, T)
19:            for (i, c) ∈ I do
20:                S_i ← parsed statements from file i
21:                Statements(S_i, p ∧ c, T)
22:            end for
23:        end if
24:    end for
25: end procedure
```

**Algorithm 3** Expand Makefile expressions, hoisting conditionals.

```
1: procedure Expand(E, p, T)
2:     ▷ Initialize result with empty string for all presence conditions.
3:     R ← {("", true)}
4:     for subexpression e ∈ E do
5:         if e is variable expansion of v then
6:             ▷ Get all definitions, recursively expanding them.
7:             R ← R × {Expand(d_i, c_i ∧ p, T) | (d_i, c_i) ∈ T(v)}
8:         else if e is function f with args (a_1, a_2, ...) then
9:             ▷ Expand all function arguments.
10:            A_n = Expand(a_n, p, T) for all a_n ∈ (a_1, a_2, ...)
11:            ▷ Execute function for all argument combinations.
12:            R ← R × {f(a_1i, a_2j, ...) | a_1i ∈ A_1, a_2i ∈ A_2, ...}
13:        else ▷ e is a string
14:            ▷ Append e to every expanded subexpression.
15:            R ← {(re, c) | (r, c) ∈ R}
16:        end if
17:    end for
18:    return R
19: end procedure
```

"+=" operator. For the latter two operators, a previously undefined variable becomes recursively-expanded by default.

Lines 17–22 evaluate the include statement. The file named in an include statement can also come from variable expansion and function calls, so its name is expanded on line 18. Lines 19-21 parse the statements from the file and evaluate them under the presence conditions of the expanded filenames.

Algorithm 3 defines the Expand procedure that finds all possible expansions of an expression. It takes an expression $E$, a presence condition $p$, and a conditional symbol table $T$. Expand returns a list of all possible expansions of the expression as $(e, c)$ tuples, where $e$ is an expanded expression and $c$ is its presence condition that leads to the expansion. Line 3 initializes the result $R$ with an empty string and the true condition, since it is the only possible expansion so far. A Makefile expression can contain several subexpressions that are either variable expansions, function calls, or string constants. Once expanded, the resulting subexpressions get concatenated. Lines 4–17 loops through each subexpression and hoists its expansions with $R$ to find all possible feature expressions of the expanded subexpressions. The operator $\times$ represents hoisting, which is formally defined as

$$R \times E = [\, (e_1 e_2, c_1 \wedge c_2) \mid (e_1, c_1) \in R \text{ and } (e_2, c_2) \in E \,]$$

where $e_1 e_2$ is concatenation.

Lines 5–7 handle the expansion of a variable $v$. Line 7 first gets all definitions from the conditional symbol table $T$. Each definition is recursively expanded, since it may contain more variables or function calls. The resulting expansions are then hoisted with $R$. Lines 8–12 handle function calls where $f$ is a function name and $a_1, a_2, \ldots$ are its arguments. Its arguments first get expanded on line 10, potentially yielding multiple expansions for each one. Line 12 evaluates $f$ for all possible combinations of arguments and hoists the results with $R$. Finally, string constants are appended to all subexpressions expanded so far in $R$ on lines 13–15. Line 18 returns the final list of expansions $R$.

# 4. EMPIRICAL EVALUATION

Kmax is evaluated for correctness, by comparing to previous work, and for performance. Section 4.1 uses the Linux source code to evaluate Kmax's completeness and correctness. First, all C files in the source tree are reconciled with Kmax's compilation units or confirmed to be non-kernel compilation units. Second, Kmax's found compilation units are each mapped to its corresponding source file. Section 4.2 compares the compilation units found by Kmax, KBuildMiner, and GOLEM, and running time performance is compared for all three tools.

## 4.1 Kmax Correctness

Kmax's correctness is evaluated with a two-pronged approach. On the one hand, if kmax gets all possible compilation units from variables, then we can be sure that there are none missing. On the other, we start with all C files in the kernel source code, and ensure that no possible compilation units are missed. If both hold true, then Kmax correctly identifies all compilation units. That Kmax collects all units from Makefile variables is matter of correctness of implementation. Since Kmax evaluates all possible variable definitions of `obj-y` and `obj-m` even in all conditional branches, Kmax collects all compilation units defined in Kbuild files by assignment to these variables. Reconciling all C files in the Linux kernel is more tricky. We need to ensure that any C files not identified by Kmax are truly not kernel compilation units. To do so, we start with all C files contained in codebase. Then we remove all C files Kmax identifies as compilation units. If Kmax is complete, the remaining C files should not be kernel compilation units. This is verified this by hand and with tool support where stated. There are many different C files in the codebase that are not kernel compilation units, and the following is a description of the process of eliminating those C files.

| **Type of C File** | **Count** |
|---|---|
| *Found by Kmax* | |
| Compilation units | 19,651 |
| Library compilation units | 200 |
| Unconfigurable units | 13 |
| Host programs | 9 |
| Extra targets | 12 |
| *Found by hand or additional scripts* | |
| From non-kbuild directories | 524 |
| Architecture-specific tools | 150 |
| ASM offsets files | 31 |
| Included C files | 147 |
| Helper programs | 13 |
| Skeleton files | 3 |
| Staging compilation units | 4 |
| Orphaned compilation units | 27 |
| Other non-Kbuild | 18 |
| Make targets | 2 |
| TOTAL C FILES | 20,804 |
| All C files in source tree | 20,804 |

Table 2: Reconciling C files Linux v3.19 source tree with Kmax's compilation units.

There are two main ways we show a C file is not a kernel compilation unit. The first way is to take the C file name and check by hand that it is not in any Kbuild file for the kernel or that it is in directory that is not part of the kernel. For instance, the `scripts` directory contains the Kbuild and Kconfig tools themselves, including a C program that parses and evaluates Kconfig constraints. This program is not part of kernel program; it is used only used during the build proess. The second way to rule out a C file uses Kmax's ability to collect variable definitions. The Kbuild files are used to compile helper programs such as hex-to-binary converters used during the build process. These non-kernel C files are identified in other reserved Kbuild variables such as `hostprogs-y`. As with kernel compilation units, we collect these compilation unit names with Kmax and rule them out as kernel compilation units.

Table 2 lists the results of accounting for all C files in the kernel source tree. The first column lists the type of C file identified, and the second column lists its count. The C file types are divided into those found from Makefile variables using Kmax and those verified by hand. At the bottom of the table are the number of C files verified followed by the number of C files contained in the entire kernel source tree, computed by running the unix `find` command from the root of the source tree: `find linux/ -name "*.c"`. The vast majority of C files are kernel compilation units, with 19,651 files corresponding to compilation units identified in `obj-y` or `obj-m`. Library compilation units account for another 200 C files identified in `lib-y` and `lib-m`, special Kbuild variables used to build libraries.

There are three types of non-kernel compilation units that Kmax identifies, unconfigurable units, host programs, and extra targets. An unconfigurable unit cannot be activated because of the feature that controls it. Several compilation units in `drivers/acpi/acpica/` are controlled by the Makefile variable `ACPI_FUTURE_USAGE`, which

| Type of Unit | Count |
|---|---|
| C files | 19,651 |
| ASM files | 687 |
| Library files | 604 |
| Generated | 48 |
| Other non-C files | 156 |
| No corresponding source | 10 |
| TOTAL UNITS | 21,158 |

Table 3: The total number of compilation units found in Linux v3.19 by Kmax with a breakdown by types of unit.

is not a feature. A unit can also become unconfigurable if controlled by an unreachable or unselectable feature. For example, `arch/cris/arch-v32/kernel/smp.o` is controlled by the feature `SMP` that is not defined in the cris architecture's Kconfig files. Even though Kmax excludes these from the list of compilation units, it still finds them in Kbuild Makefiles.

Host programs are tools compiled and run during the build process but not compiled into the kernel. `sound/oss/hex2hex.c`, for example, is a stand-alone program that converts hexadecimal codes to a C array. Kmax finds these in several special Kbuild variables such as `hostprogs-y`. Other programs compiled by Kbuild that are not part of the kernel are put in the special variable 'extra-y', which is used during `make clean` to remove the compiled programs.

Four directories do not contain kernel source, as confirmed both by their omission from Kbuild files and by manually inspecting the directories and Linux documentation. These are `Documentation/`, `samples/`, `scripts/`, and `tools/` and account for 524 C files. Similarly, there are architecture-specific `tools/` directories and bootloader code that is also not part of the kernel proper. ASM offsets files are those used to generate the `asm-offsets.h` header file for the given architecture by compiling the C file to assembly. 147 files have the `.c` extension, but are included via the `#include` directive like headers in other C files. Helper programs are test code or template files that were manually confirmed not to be referenced by Kbuild files and usually contain comments that describe their purpose. Similarly, skeleton files have the word "skeleton" in their name and are templates for driver writers.

Some C files look like kernel compilation units, but are not referenced by Kbuild. Four of these appear in the `staging/` directory. Drivers in this directory are pending inclusion into the mainline kernel, and may not be completely integrated with Kbuild. 27 unreferenced files are orphaned, perhaps representing dead code or bugs in the Kbuild files. All orphans were investigated manually to confirm their omission from Kbuild. The other non-Kbuild files come from real-mode and user-mode Linux directories and were also manually confirmed not to be used by Kbuild. Lastly, some compilation units do not have the same name as their C counterpart, because the Makefiles use `make` rules to build the unit instead of Kbuild's special `obj-y` variables. These represent a true limitation of Kmax, since it does not evaluate `make` targets.

The limitations of this approach are that compilation units without a source file behind it are not accounted for. Also, gathering all compilation units depends on the correctness of the implementation, which can have bugs, in spite of a correct algorithm that collects all variable definitions.

The second evaluation of Kmax correctness starts instead with the compilation units and associates them with their corresponding source files. Table 3 shows a breakdown by type of units and their counts. Most compilation units are C files, but there are also hundreds of assembly files. The library compilation units are mostly assembly, as Table 2 shows only 200 are C files. 48 of the compilation units do not exist in the source because they are generated while building the kernel, as confirmed by investigating their Kbuild files. The other types of non-C files are firmware binaries and device tree blobs which are loaded by the bootloader along with the kernel. 10 compilation units are defined in Kbuild, but have missing source code, representing errors or regressions.

| Tool | Units | C File Units | Archs Failed |
|---|---|---|---|
| Kmax | 21,158 | 19,651 | 0 |
| KbuildMiner | 17,812 | 16,948 | 6 |
| GOLEM | 19,601 | 18,404 | 0 |

(a) The total number of compilation units found in Linux v3.19 by each tool, the number of C file units, the number architectures the tool failed to process.

| Tool | Extracted | Misidentified | Found |
|---|---|---|---|
| Kmax | 15,124 | – | 15,124 |
| KBuildMiner | 15,056 | 450 | 14,606 |
| GOLEM | 15,031 | 404 | 14,627 |

(b) A summary of previous work's precision in extracting compilation units from the x86 version of Linux v3.19, with the number compilation units misidentified for x86.

| Tool | Units | Archs Failed | x86 | Misidentified |
|---|---|---|---|---|
| Kmax | 13,510 | 0 | 9,344 | – |
| KbuildMiner | 11,220 | 0 | 9,136 | 195 |
| GOLEM | 11,325 | 0 | 9,145 | 185 |

(c) Comparison with Linux 2.6.33.3.

Table 4: A comparison of tools running on Linux v3.19.

## 4.2 Comparison

Kmax is compared to the previous tools KBuildMiner and GOLEM for both correctness and performance. For correctness, each tool was run on the Linux v3.19 source code for each architecture, and their resulting compilation units collected. Since previous work shows both tools running successfully on Linux v2.6.33.3 [2, 10], the same experiment was repeated for that version. For performance, each tool was repeatedly run on the x86 architecture alone to collect its latency.

Table 4 compares the compilation units found by Kmax with those found by the other tools. Table 4a list the total number of compilation units extracted by each tool across all architectures, how many correspond to C files, and the number of architectures, if any, the tool failed to process. Kmax extracts more compilation units when compared to both KBuildMiner and GOLEM by about 3,000 and 1,500 respectively. KBuildMiner, however, fails on six out of the 30 architectures, which is partly responsible its low numbers.

To control for these failures, the tools are also compared on the x86 architecture alone in Table 4b. This table lists the number of x86 compilation units extracted by each tool, how many are misidentified as being part of the x86 architecture, and the actual number of correct compilation units found. While the number of units found is similar for each tool, this number does not reflect tool precision. When compared to Kmax's results, both KBuildMiner and GOLEM misidentify more than 400 compilation units each as being part of the x86 source code. These units were spot-checked to confirm the misidentification.

Nearly all of these misidentified units appear in Kmax's compi-

| Tool | Language | Min | Mean | Max |
|---|---|---|---|---|
| Kmax | python | 46.69 sec | 46.75 sec | 46.80 sec |
| KBuildMiner | java/scala | 11.82 sec | 12.32 sec | 12.87 sec |
| GOLEM | python | 53.96 min | 54.56 min | 55.04 min |

(a) Latency for Linux v2.6.33.3 x86.

| Tool | Language | Min | Mean | Max |
|---|---|---|---|---|
| Kmax | python | 84.03 sec | 84.15 sec | 84.25 sec |
| KBuildMiner | java/scala | 44.17 sec | 45.00 sec | 45.87 sec |
| GOLEM | python | 3.41 hrs | 3.42 hrs | 3.43 hrs |

(b) Latency for Linux v3.19 x86.

Table 5: Latency of each tool to compute the compilation units for the x86 architecture of two Linux versions, v3.19 and v2.6.33.3. Each tool was run five times, plus a 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.

lation units for other architectures. For instance, both KBuildMiner and GOLEM identify `drivers/block/ps3disk.c`. This compilation unit is controlled by the `PS3_DISK` feature defined only in `arch/powerpc/platforms/ps3/Kconfig`, making it only available when building for the PowerPC architecture. Another example is `drivers/ide/icside.c`, illustrated in Figure 4f as being only available for the arm architecture.

Some misidentified units can never be compiled into the kernel. For example, compilation units from `drivers/acpi/acpica` such as `hwtimer.o` are controlled by `ACPI_FUTURE_USAGE`, apparently a non-feature used a a placeholder for future use. There are seven such compilation units for KBuildMiner and three for GOLEM. The remaining misidentified compilation units were all found in other architectures by Kmax.

To account for tool regressions on newer versions of Linux, the same experiments were conducted on a version used in each tool's own previous work. Table 4c shows the number of units, failures, x86 units, and misidentifications for the 2.6.33.3 version of Linux. KBuildMiner does not fail on any architecture, and finds about as many compilation units as GOLEM. However, Kmax's relative performance is comparable to v3.19, finding more than 2,000 more compilation units. The number of misidentifications by both KBuildMiner and GOLEM is also comparable. While there are about half the misidentifications compared to v3.19, there are also about 40% fewer compilation units overall.

The latency of all three tools was tested by running each five times for the x86 architecture of both Linux v2.6.33.3 and v3.19. These experiments were run on a development machine with an Intel Core i5 3.30GHz processor and 8GB of RAM. Table 5 lists the tool, the language its written in, and the latency. Since KBuildMiner uses the Java Virtual Machine (JVM), a warm-up run was performed before collecting the five tests to avoid the additional latency incurred by the first run.

Table 5a shows the results for Linux v2.6.33.3, listing the minimum, average computed by mean, and maximum of the fives runs. KBuildMiner is the fastest, since it parses the Kbuild Makefiles without having to evaluate them and is written in Java. It takes on average 12.32 seconds for the x86 architecture, while Kmax takes 46.75 seconds on average, nearly four times slower. But both take less than a minute, while GOLEM takes nearly an hour. While also written in python, GOLEM repeatedly executes `make` on each Kbuild Makefile for one or more features at a time. Given the large number of Makefiles and features in each, this process is time-consuming without much better results than the faster KBuildMiner parsing approach.

Table 5b shows the results for the same experiment on Linux v3.19. Linux v2.6.33.3 has 9,344 compilation units for x86, while v3.19 has 15,124, more than 60% more. As expected, all tools take longer. KBuildMiner is still the fastest at 45 seconds, but takes almost four times longer than on v2.6.33.3. GOLEM takes 3.42 hours on average, about 3.5 times longer. Kmax scales somewhat better, taking about twice as long with 84.15 seconds on average.

KBuildMiner's fuzzy parsing is the fastest, while GOLEM is orders of magnitude more time-consuming than both of the other tools. Given the added complexity of Makefile evaluation across software product lines, Kmax incurs a relatively small latency compared to parsing alone and scaled better for a larger version of the Linux kernel. The trade-off is an accurate and precise set of compilation units.

# 5. LIMITATIONS

Kmax does not evaluate the complete Make language. It supports variable assignment and expansion, most function calls, and the include statement. Missing are Makefile rules. Rules build a target file by running shell commands and user-defined functions. Rules are used to run helper programs in Kbuild files, but do not limit Kmax's ability to find compilation unit names, which are specified in special Kbuild variables like `obj-y`. This limitation did prevent Kmax from finding two C files that correspond to compilation units with a different name, because rules are used to compile them instead of Kbuild. The `shell` function, like rules, can also be used to call external programs. An external program could potential take features and perform build steps outside the Kbuild specification. This would be problematic for any attempt to find all compilation units from the build system. External programs are used to generate header files like `asm-offsets`, creating an issue for software tools like bug finders that try to process all possible source code, including headers.

Some non-boolean variables are globally defined in non-Kbuild Makefiles, e.g., the `BITS` variable from Figure 4b is used to generate some compilation unit names. Other non-booleans are features. `CONFIG_WORD_SIZE` is used to construct compilation unit names in the PowerPC architecture. Not defined in any Makefile, one way get the range of values for this non-boolean is to look at the `default` construct in its Kconfig definition, although not all non-boolean features have explicit defaults. Kmax requires the non-booleans to be preloaded in the conditional symbol table. There are only three such multiply-defined variables needed by Kmax for Linux v3.19 including the two described above. The last one is `MMU` which, as defined under the microblaze architecture, is either set to `-nommu` or the empty string and is used to choose between two compilation units depending on support for a memory management unit.

# 6. RELATED WORK

Both GOLEM and KBuildMiner are part of greater efforts not just to find compilation units but also to map features to compilation units. Dietrich et al compares GOLEM to other tools including KBuildMiner to evaluate their coverage of compilation units [10]. KBuildMiner is a standalone tool described by Berger [6]. Both GOLEM and KBuildMiner are part of greater efforts not just to find compilation units also map features to compilation units. Tartler et al describes using Undertaker to remove dead code from compilation units, i.e., code that can never be enabled in any software product line [24]. Andersen et al used KBuildMiner to create a fea-

ture model for Linux [5]. KBuildMiner has also been used solely as a source for the set of compilation units. Liebig et al uses them for analyzing Linux's variability [17, 18]. Gazzillo and Grimm [14] and Kastner et al [15] tests their parsers on this set of compilation units as well. But since KBuildMiner yields incomplete results, these analyses are not of the complete kernel.

There are many studies on Linux's feature model, its build system, and its variability mechanisms. Sincero et al identified Kconfig as a feature model [22], 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 [7]. She et al built a formal hierarchy of features for Linux [21]. Dietrich et al quantified the granularity of features in the Linux kernel [11]. Dintzner et al tracked changes in Linux's feature model over time [12]. Tartler et al calculated code coverage for single software product line and maximized coverage with a minimal set of features [23]. Nadi and Holt analyzed Kbuild Makefiles to find anomalies such as unused compilation units [20]. 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 [25].

Kmax uses techniques from other tools that process source code containing variability. Garrido et al discuss refactoring C code containing preprocessor directives and macros and introduces conditional symbol tables for storing multiple definitions across combinations of features [13]. Gazzillo and Grimm formalize and use hoisting to evaluate language constructs across feature combinations in their configuration-preserving C parser [14].

# 7. CONCLUSION

Kmax is a building block for variability-aware software engineering tools that extracts Linux compilation units and their variability information accurately, making heuristic approaches unnecessary. This building block is key to project-wide static analysis tools, such as bug-finders, code browsers, and refactoring tools. The core of Kmax is its algorithm to evaluate `make` language across all combinations of features simultaneously. It collects all possible variable definitions, evaluates all conditional branches, and maintains a presence condition that determines the features controlling the compilation units. Linux's complex build process adds extras challenges, because each architecture forms the root of its own source code hierarchy. Kmax uses the Selectable algorithm to deduce which features belong to which architectures. Kmax is empirically evaluated on the Linux 3.19, demonstrating the completeness and correctness of Kmax's results. Morever, it is compared to two previous solutions that approximate the complete set of compilation units, revealing the limitations of heuristic solutions. A comparison of running time shows that the added complexity needed by Kmax only incurs a small trade-off in running time compared to previous work.

## Acknowledgements

I would like to thank Professor Thomas Wies for his advice and support.

# 8. REFERENCES

[1] GNU make Manual. `https://www.gnu.org/software/make/manual`.

[2] Kbuildminer. `https://code.google.com/p/variability/`.

[3] I. Abal, C. Brabrand, and A. Wasowski. 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, pp. 421–432, New York, NY, USA, 2014. ACM.

[4] B. Adams et al. Can we refactor conditional compilation into aspects? In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development*, pp. 243–254, Mar. 2009.

[5] N. Andersen, K. Czarnecki, S. She, and A. Wąsowski. Efficient synthesis of feature models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pp. 106–115, New York, NY, USA, 2012. ACM.

[6] T. Berger, S. She, K. Czarnecki, and A. Wasowski. Feature-to-code mapping in two large product lines. Tech. report, University of Leipzig (Germany), University of Waterloo (Canada), IT University of Copenhagen (Denmark), 2010.

[7] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. 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, pp. 73–82, New York, NY, USA, 2010. ACM.

[8] BusyBox. `http://www.busybox.net/`.

[9] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pp. 73–88, Oct. 2001.

[10] C. Dietrich, R. Tartler, W. Schröder-Preikshat, and D. Lohmann. A robust approach for variability extraction from the linux build system. In *Proceedings of the 16th International Software Product Line Conference*, pp. 21–30, Sept. 2012.

[11] C. Dietrich, R. Tartler, W. Schröder-Preikshat, and D. Lohmann. Understanding linux feature distribution. In *Proceedings of the 2012 Workshop on Modularity in Systems Software*, pp. 15–20, Mar. 2012.

[12] N. Dintzner, A. Van Deursen, and M. Pinzger. Extracting feature model changes from the linux kernel using fmdiff. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, pp. 22:1–22:8, New York, NY, USA, 2013. ACM.

[13] A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[14] P. Gazzillo and R. Grimm. SuperC: parsing all of C by taming the preprocessor. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 323–334, June 2012.

[15] C. Kästner et al. 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*, pp. 805–824, Oct. 2011.

[16] C. Kästner et al. A variability-aware module system. In *Proceedings of the 27th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 773–792, Oct. 2012.

[17] J. Liebig et al. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of*

*the 32th International Conference on Software Engineering*, pp. 105–114, May 2010.

[18] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pp. 81–91, New York, NY, USA, 2013. ACM.

[19] B. McCloskey and E. Brewer. ASTEC: A new approach to refactoring C. In *Proceedings of the 10th European Software Engineering Conference*, pp. 21–30, Sept. 2005.

[20] S. Nadi and R. Holt. Make it or break it: Mining anomalies from linux kbuild. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pp. 315–324, Washington, DC, USA, 2011. IEEE Computer Society.

[21] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pp. 461–470, New York, NY, USA, 2011. ACM.

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

[23] R. Tartler et al. Configuration coverage in the analysis of large-scale system software. *ACM SIGOPS Operating Systems Review*, 45(3):10–14, Dec. 2011.

[24] R. Tartler et al. 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*, pp. 47–60, Apr. 2011.

[25] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1):6:1–6:45, June 2014.

[26] uClibc. `http://www.uclibc.org/`.