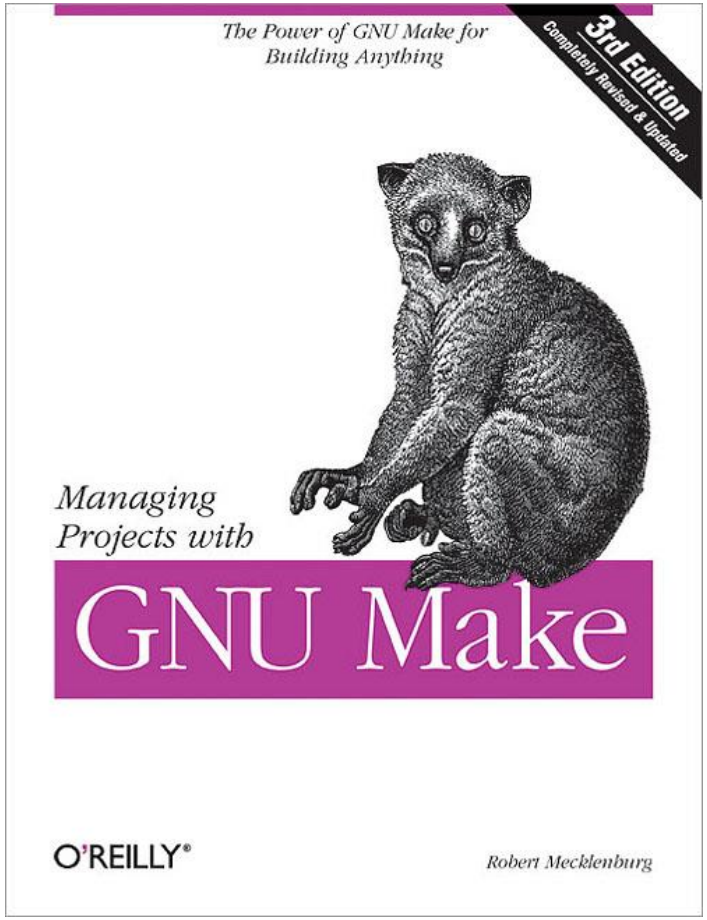
Kmax: Finding All Configurations of Kbuild Makefiles Statically

Paul Gazzillo

Stevens Institute

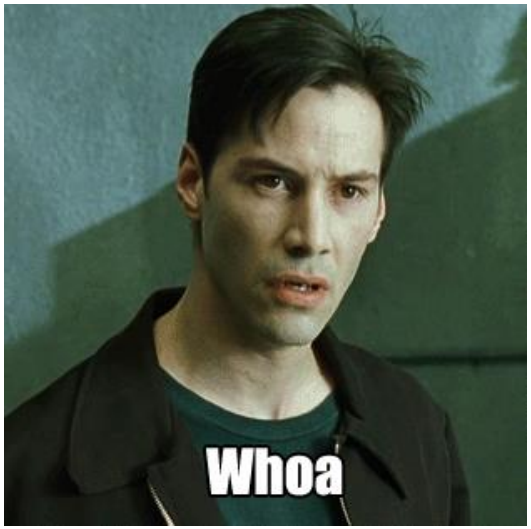
ESEC/FSE 2017 Paderborn, Germany

Let’s Talk About Makefiles



Variability in Linux Kbuild

- Kbuild is Linux's Makefile-based build system
- Linux has 14,000+ configuration options
  - 2^14,000 configurations in the worst case
- 1,985 Kbuild Makefiles
- 29,525 SLoC
- Controlling 19,651 C files

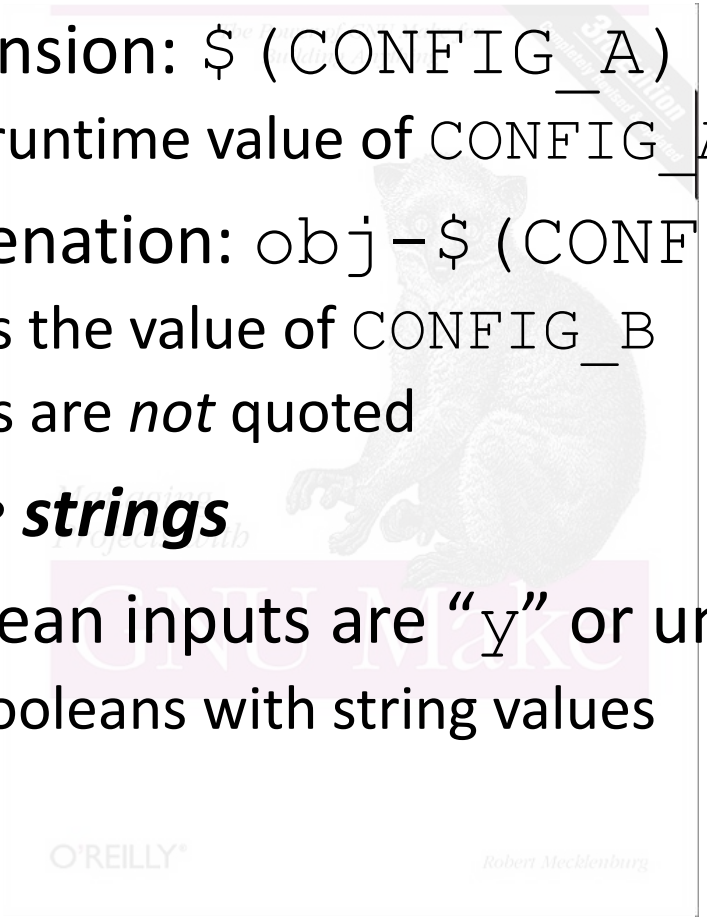


What Kmax Offers

- Lack tools to reason about Makefile variability
- Simple questions are hard
  - What C files comprise the Linux kernel?
- Kmax is a static analysis of Kbuild Makefiles
- Finds all C files and their configurations
  - 1-2k more C files compared to previous heuristics
- Takes minutes
- Finds dead code

Makefile Syntax

- Variable expansion: `$(CONFIG_A)`
  - Expands to runtime value of `CONFIG_A`
- String concatenation: `obj-$(CONFIG_B)`
  - "`obj-`" plus the value of `CONFIG_B`
  - String values are *not* quoted
- ***All values are strings***
- In Linux, boolean inputs are "`y`" or undefined
  - Simulates booleans with string values



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

- Takes `CONFIG_A` and `CONFIG_B` as boolean inputs
  - "y" or undefined
- Sets `obj-y` to set of object files, conditioned on inputs
- Compiles and links C files in `obj-y`

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

- Assignment: `obj-y` gets `fork.o` to compile

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

Kbuild-speak for Boolean ”true”

- Conditional: value of `BITS` depends on `CONFIG_A`

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

- Concatenation: right-hand side computed from `BITS`, implicitly depends on `CONFIG_A`

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

Also a string!

Kbuild won't build these files

- Runtime variable name construction:
  - Variable to assign depends on value of CONFIG_B
  - Appends `probe_*.o` to either `obj-y` or `obj-`
  - Challenge for static approaches

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

What C files does this build and when?

Compute All Configurations

CONFIG_A	CONFIG_B	obj-y	obj-
on	on	fork.o probe_32.o	(undefined)
on	off	fork.o	probe_32.o
off	on	fork.o probe_64.o	(undefined)
off	off	fork.o	probe_64.o

- Take all combinations of CONFIG_A and CONFIG_B
- Exponential in number of configuration options
- Has duplicate information

Solution Approaches?

- Brute force
  - Too many possible configurations
- Dynamic analysis
  - GOLEM heuristically chooses configurations to run
  - Still too many configurations
- `grep`
  - Runtime string manipulation limits effectiveness
- Parsing
  - Syntax is not enough, need semantics
  - KBuildMiner is an example of the parsing approach

Key Insight

Paths are configurations. A static analysis can collect configuration information if it is path-sensitive and has a precise string abstraction.

Kmax's Static Analysis

- Static analysis analyzes all paths
  - Paths are configurations
- Path abstraction treats configurations symbolically
- String abstraction enumerates concrete values
- Scalability and precision
  - Efficient symbolic representation
  - Aggressively trim infeasible paths

Path Abstraction

- Boolean expressions of configuration options
  - Symbolic, e.g, `CONFIG_B` $\wedge \neg$ `CONFIG_A`
- Implemented with binary decision diagrams (BDDs)
  - Easy to join and deduplicate paths
  - Easy to trim infeasible paths

String Abstraction

- Enumerate all possible concrete strings
- Relies on path abstraction to be efficient
- For example, one string may be

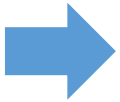
[“probe_32.o” if BITS==32 ∧ CONFIG_B ,
“probe_64.o” if BITS==64 ∧ CONFIG_B]

- Akin to conditional symbol tables
  - Previous variability-aware approaches
    [Garrido & Johnson ’05, Kaestner et al ‘11, Gazzillo &Grimm ’12, Walkingshaw et al ‘14, Nguyen et al ‘14, Meinicke et al ‘16]

```
1  obj-y := fork.o
2  ifeq ($(CONFIG_A),y)
3    BITS := 32
4  else
5    BITS := 64
6  endif
7  obj-$(CONFIG_B) += probe_$(BITS).o
```

Current path	True (all configurations)

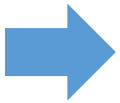
Symbol table	(empty)



```
1   obj-y := fork.o
2   ifeq ($(CONFIG_A),y)
3     BITS := 32
4   else
5     BITS := 64
6   endif
7   obj-$(CONFIG_B) += probe_$(BITS).o
```

Current path	True (all configurations)

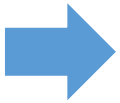
Symbol table	obj-y = [“`fork.o`” if True]



```
1   obj-y := fork.o
2   ifeq ($(CONFIG_A),y)
3     BITS := 32
4   else
5     BITS := 64
6   endif
7   obj-$(CONFIG_B) += probe_$(BITS).o
```

Current path `CONFIG_A`

Symbol table obj-y = [“`fork.o`” if True]

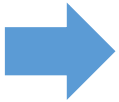


```
1   obj-y := fork.o
2   ifeq ($(CONFIG_A),y)
3     BITS := 32
4   else
5     BITS := 64
6   endif
7   obj-$(CONFIG_B) += probe_$(BITS).o
```

Current path `CONFIG_A`

Symbol table obj-y = [“`fork.o`” if True]

BITS = [“`32`” if `CONFIG_A`]

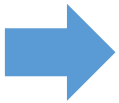


```
1   obj-y := fork.o
2   ifeq ($(CONFIG_A),y)
3     BITS := 32
4   else
5     BITS := 64
6   endif
7   obj-$(CONFIG_B) += probe_$(BITS).o
```

Current path ¬CONFIG_A

Symbol table obj-y = [“fork.o” if True]

BITS = [“32” if CONFIG_A]



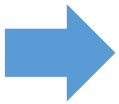
```
1   obj-y := fork.o
2   ifeq ($(CONFIG_A),y)
3     BITS := 32
4   else
5     BITS := 64
6   endif
7   obj-$(CONFIG_B) += probe_$(BITS).o
```

Current path CONFIG_B ∧ ¬CONFIG_A

Symbol table obj-y = ["fork.o" if True]

BITS = ["32" if CONFIG_A,

"64" if ¬CONFIG_A]



```
1   obj-y := fork.o
2   ifeq ($(CONFIG_A),y)
3     BITS := 32
4   else
5     BITS := 64
6   endif
7   obj-$(CONFIG_B) += probe_$(BITS).o
```

Current path

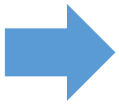
True (all configurations)

Symbol table

obj-y = [“fork.o” if True]

BITS = [“32” if CONFIG_A,

“64” if ¬CONFIG_A]



```
1   obj-y := fork.o
2   ifeq ($(CONFIG_A),y)
3     BITS := 32
4   else
5     BITS := 64
6   endif
7   obj-$(CONFIG_B) += probe_$(BITS).o
```

Current path — True (all configurations)

Symbol table

obj-y = [“fork.o” if True]

BITS = [“32” if CONFIG_A,

“64” if ¬CONFIG_A]

?

Runtime Variable Names

Expand to all assignments

```
obj-$(CONFIG_B) += probe_$(BITS).o
```

Evaluate under resulting new paths

```
ifeq ($(CONFIG_B),y)
  obj-y += probe_$(BITS).o
else
  obj- += probe_$(BITS).o
endif
```

```
1  obj-y := fork.o
2  ifeq ($(CONFIG_A),y)
3    BITS := 32
4  else
5    BITS := 64
6  endif
7  obj-$(CONFIG_B) += probe_$(BITS).o
```

- obj-y's final value tells us that
  - "`fork.o`" is in all configurations
  - "`probe_32.o`" when `CONFIG_B` ∧ `CONFIG_A`
  - "`probe_64.o`" when `CONFIG_B` ∧ ¬ `CONFIG_A`

More Details in the Paper

- Complete analysis algorithm
- Handling runtime variable name construction
- Updating the symbol table with assignments
  - Disjoint and complete configuration coverage
  - Undefined variable configurations
- Trimming infeasible configurations
- Converting conditionals to BDDs
- Gathering configuration options from Kconfig

Evaluation

Experimental Setup

- Kmax evaluated on two Kbuild clients
  - Linux v3.19
  - BusyBox v1.25.0
- Experiment #1: correctness
  - Checks for missing C files in Kmax output
- Experiment #2: comparison to previous work
  - Check C files against two previous heuristics
- Experiment #3: running time
  - Compare running time with previous tools

Experiment #1: Correctness

- Compare .c files in source tree with Kmax output
- Not all .c files destined for kernel binary
- Verify Kmax excluded only non-kernel .c files

Experiment #1: Correctness

These are not compilation units

Dead code!

Kmax misses none

	pe of C File	Linux	BusyBox
1	*ntified by Kmax*	19,651	560
	aries	200	0
	n-Kbuild directories	524	23
	elper programs	215	5
5	ncluded C files	147	21
6	Examples	3	6
7	Orphaned	49	18
	uration option	13	0
	with Kbuild	2	0
	TOTAL C FILES	20,804	633
	All C files in source tree	20,804	633
	Missed by Kmax	0	0

Experiment #2: Comparison

- Compared to two previous tools' heuristics
  - KBuildMiner parses Kbuild Makefiles
  - GOLEM runs Makefiles one configuration a time
  - *These were not advertised as complete solutions*
- Missing: should be included but weren't
- Misidentified: shouldn't be included, eg, dead code

Tool	x86 C Files	Missed	Misidentified
Kmax	14,783	—	—
KBuildMiner	14,904	319	440
GOLEM	14,460	713	390

Experiment #3: Running Time

- x86 version of kernel source
- 5 running time collections per tool
- KBuildMiner's parsing approach is the fastest
- GOLEM far slower than both, taking hours
- Kmax is more precise with little additional overhead

Tool	Mean Running Time
Kmax	84.15 sec
KBuildMiner	45.00 sec
GOLEM	3.42 hrs

Future Work

- Integration into variability-aware analyses, e.g., bugfinders
- Variability-aware dependence graphs
- Application to other Makefiles

Conclusion

- Kmax algorithm
  - Path-sensitive static analysis
  - Enumerates concrete strings
  - Symbolic configuration expressions
- Evaluation on Linux and BusyBox
  - Finds all C files and their configurations
  - More precise than heuristics with little overhead
  - Finds dead code

Thank You! Questions?

Kmax Repository

https://github.com/paulgazz/kmax

https://paulgazzillo.com @paul_gazzillo