Enabling Variability-Aware Software Tools

Paul Gazzillo
New York University

We Need Better C Tools

- Linux and other critical systems written in C
  - Need source code browsers
    - 20,000+ compilation units, 10+ million lines
  - Need bug finders
    - 1,000 found by static checkers [Chou et al., SOSP ’01]
  - Need refactoring tools
    - 150+ errors due to interface changes
      [Padioleau et al., EuroSys ’08]

We Need Better C Tools

- Linux and other critical systems written in C
  - Need source code browsers
    - 20,000+ ion lines
  - Need bug finders
    - 1,000 found by static checkers [Chou et al., SOSP ’01]
  - Need refactoring tools
    - 150+ errors due to interface changes
      [Padioleau et al., EuroSys ’08]

Variability gets in the way!

Undefined Symbol Error

```
#ifdef CONFIG_CRYPTO_BLKCIPHER
void *crypto_alloc_ablkcipher()
{
  return (void*)0;
}
#endif
#ifdef CONFIG_CRYPTO_TEST
static void test_cipher()
{
  crypto_alloc_ablkcipher();
}
#endif
```

Undefined Symbol Error

Defined with CRYPTO_BLKCIPHER

```
#ifdef CONFIG_CRYPTO_BLKCIPHER
void *crypto_alloc_ablkcipher()
{
  return (void*)0;
}
#endif
#ifdef CONFIG_CRYPTO_TEST
static void test_cipher()
{
  crypto_alloc_ablkcipher();
}
#endif
```

Undefined Symbol Error

Defined with CRYPTO_BLKCIPHER

Used with CRYPTO_TEST

```
#ifdef CONFIG_CRYPTO_BLKCIPHER
void *crypto_alloc_ablkcipher()
{
   return (void*)0;
}
#endif

#if ef CONFIG_CRYPTO_TEST
static void test_cipher()
{
   crypto_alloc_ablkcipher();
}
#endif
```

Undefined Symbol Error

```
#ifdef CONFIG_CRYPTO_BLKCIPHER
void *crypto_alloc_ablkcipher()
{
  return (void*)0;
}
#endif
#ifdef CONFIG_CRYPTO_TEST
static void test_cipher()
{
  crypto_alloc_ablkcipher();
}
#endif
```

Undefined Symbol Error

```
#ifdef CONFIG_CRYPTO_BLKCIPHER
void *crypto_alloc_ablkcipher()
{
  return (void*)0;
}
#endif
#ifdef CONFIG_CRYPTO_TEST
static void test_cipher()
{
  crypto_alloc_ablkcipher();
}
#endif
```

Undefined Symbol Error

```
#ifdef CONFIG_CRYPTO_BLKCIPHER
void *crypto_alloc_ablkcipher()
{
  return (void*)0;
}
#endif
#ifdef CONFIG_CRYPTO_TEST
static void test_cipher()
{
  crypto_alloc_ablkcipher();
}
#endif
```

Undefined Symbol Error

```
#ifdef CONFIG_CRYPTO_BLKCIPHER
void *crypto_alloc_ablkcipher()
{
  return (void*)0;
}
#endif
#ifdef CONFIG_CRYPTO_TEST
static void test_cipher()
{
  crypto_alloc_ablkcipher();
}
#endif
```

Undefined Symbol Error

```
#ifdef CONFIG_CRYPTO_BLKCIPHER
void *crypto_alloc_ablkcipher()
{
   return (void*)0;
}
#endif

#ifdef CONFIG_CRYPTO_TEST
static void test_cipher()
{
   crypto_alloc_ablkcipher();
}
#endif
```

Undefined symbol error

Variability

- Build-time configuration tailors software
- Linux has over 14,000 configuration variables
  - Architecture, file systems, drivers, and more
  - Runs on refrigerators and server farms alike
- C software uses
  - The preprocessor within C files
  - Makefiles (or equivalent) for the build process

Two Subproblems

- Parsing: difficult because of the preprocessor
  - SuperC

Two Subproblems

- Parsing: difficult because of the preprocessor
  - SuperC
- Build system: programs comprise many C files
  - Kmax

Two Subproblems

- Parsing: difficult because of the preprocessor
  - SuperC
- Build system: programs comprise many C files
  - Kmax
- Together they enable variability-aware tools
  - Simple bug-finding example

Overview

- SuperC
- Kmax
- Bug-finding

C Source Code Written in Two Languages

- C proper and the preprocessor
- Preprocessor is a simple, text processing language
  - Static conditionals: configure source code
  - Macros: abbreviate C constructs
  - Headers: break source code into separate files
- Preprocessor makes parsing source code tricky
  - Hides C source code in macros and headers
  - Breaks C syntax

Examples of the Challenges

- Macros expand to arbitrary C fragments

```
#define for_each_class(c) \
  for (c = highest_class; c; c = c->next)
```

Examples of the Challenges

- Macros expand to arbitrary C fragments

```
#define for_each_class(c) \
  for (c = highest_class; c; c = c->next)
```

- Directives appear between arbitrary C fragments

```
#ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
  if (imajor(inode) == 10)
    i = 31;
  else
#endif
    i = iminor(inode) – 32;
```

SuperC to the Rescue!

- Processes source in two steps, like a compiler
  - Preprocessor
    - Expands macros and includes headers
    - But preserves conditionals
  - Parser creates an AST for *all* configurations

Parsing All Configurations

- *Forks* subparsers at conditionals
- *Merges* subparsers in the same state after conditionals
  - Joins AST subtrees with *static choice nodes*
  - Preserves mutually exclusive configurations

Fork-Merge Parsing in Action

```
#ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
  if (imajor(inode) == 10)
    i = 31;
  else
#endif
    i = iminor(inode) – 32;
if (i >= MOUSEDEV_MINORS) ...
```

TypeChef Parsing in Action

(1) Fork subparsers on conditional

```
#ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
    if (imajor(inode) == 10)
        i = 31;
    else
#endif
        i = iminor(inode) – 32;
if (i >= MOUSEDEV_MINORS) ...
```

Conditional Merge Parsing

(1) Fork subparsers on conditional

(2) Parse the *entire* if-then-else

```
#ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
    if (imajor(inode) == 10)
        i = 31;
    else
#endif
        i = iminor(inode) – 32;
if (i >= MOUSEDEV_MINORS) ...
```

If-Then-Else

TypeChef Parsing

```
#ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
    if (imajor(inode) == 10)
        i = 31;
#endif
    i = iminor(inode) - 32;
if (i >= MOUSEDEV_MINORS) ...
```

(1) Fork subparsers on conditional

(2) Parse the *entire* if-then-else

(3) Parse *just* the assignment

If-Then-Else

Assignment

Fork-Merge Parsing

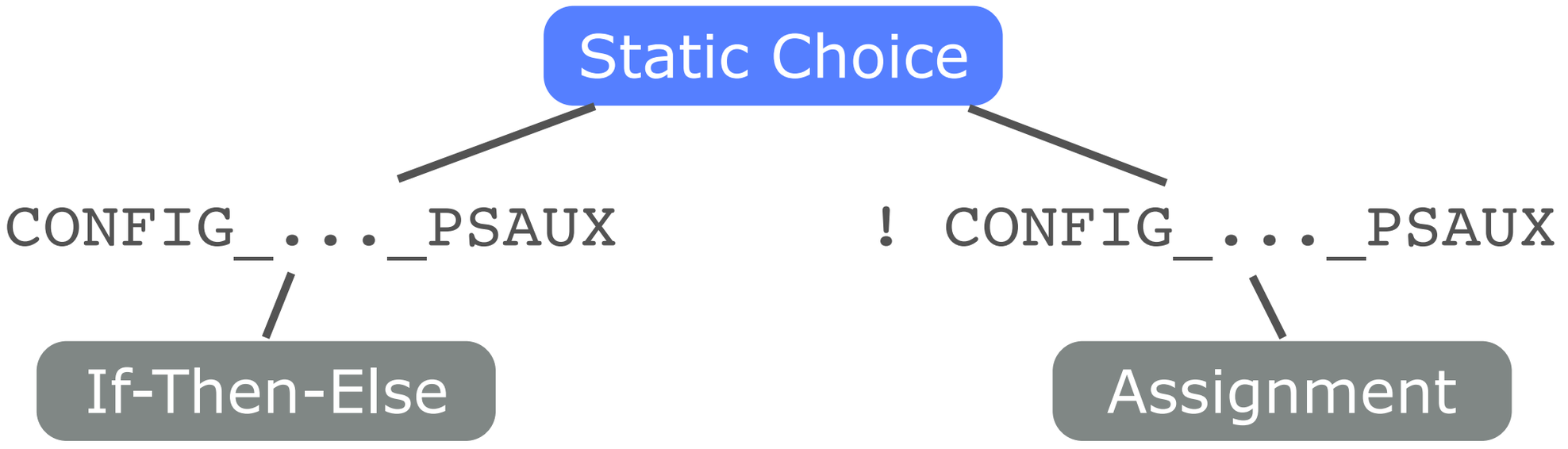
(1) Fork subparsers on conditional

(2) Parse the *entire* if-then-else

(3) Parse *just* the assignment

(4) Merge and create the static choice node

```
#ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
    if (imajor(inode) == 10)
        i = 31;
#endif
    i = iminor(inode) - 32;
if (i >= MOUSEDEV_MINORS) ...
```



When to Fork Subparsers?

- Naive strategy: fork on every conditional branch
  - Blows up on Linux x86
    - Conditionals are 40 levels deep, 10 in a row
- Our forking strategy: *token follow-set*
  - All tokens reachable from current position
  - Across all configurations

Summary

- SuperC is the first solution to parsing *all* of C
  - Preprocessor preserves all conditionals
    - Hoisting enables token-level operations
  - Parser forks and merges subparsers
    - Reuses existing parser generator and grammar
  - Token follow-set makes parsing feasible
    - Further optimized for fewer subparsers
- SuperC scales well across Linux x86
- “Parsing all of C by Taming the Preprocessor” PLDI12 with Grimm

Overview

- SuperC
- Kmax
- Bug-finding

Variability in Build System

- C programs built with separate compilation
  - Functions defined in global namespace
    - Interface checks only at link-time
- Linux has over 20,000 compilation units
  - Conditionally compiled and linked
- Software tools need complete source

Build System Example

- From `crypto/ablkcipher.c`

```
void *crypto_alloc_ablkcipher() {
  return (void*)0;
}
```

Build System Example

- From `crypto/ablkcipher.c`

```
void *crypto_alloc_ablkcipher() {
  return (void*)0;
}
```

- From `crypto/tcrypt.c`

```
static void test_cipher() {
  crypto_alloc_ablkcipher();
}
```

Build System Example

- From `crypto/ablkcipher.c`

```
void *crypto_alloc_ablkcipher() {
  return (void*)0;
}
```

- From `crypto/tcrypt.c`

```
static void test_cipher() {
  crypto_alloc_ablkcipher();
}
```

- From `crypto/Makefile`

```
obj-$(CONFIG_CRYPTO_BLKCIPHER) += ablkcipher.o
obj-$(CONFIG_CRYPTO_TEST) += tcrypt.o
```

Build System Example

- From `crypto/ablkcipher.c`

```
void *crypto_alloc_ablkcipher() {
  return (void*)0;
}
```

- From `crypto/tcrypt.c`

```
static void test_cipher() {
  crypto_alloc_ablkcipher();
}
```

- From `crypto/Makefile`

```
obj-$(CONFIG_CRYPTO_BLKCIPHER) += ablkcipher.o
obj-$(CONFIG_CRYPTO_TEST) += tcrypt.o
```

- Need all compilation units and when they are compiled

Kmax’s Makefile Evaluator

- Enters all branches of conditionals
- Hoists conditionals around complete statements
- Keeps variables in a conditional symbol table
  - `obj-y` contains all compilation units in Linux

Kmax in Operation

```
obj-$(CONFIG_CRYPTO_TEST) += tcrypt.o
```

Variable expands to conditional

...ax in Operation

```
obj-$(CONFIG_CRYPTO_TEST) += tcrypt.o
```

```
obj-
ifeq
(CONFIG_CRYPTO_TEST, y)
y
endif
+= tcrypt.o
```

...ax in Operation

Variable expands to conditional

```
obj-$(CONFIG_CRYPTO_TEST) += tcrypt.o
```

Hoist conditional around assignment

```
obj-
ifeq
(CONFIG_CRYPTO_TEST, y)
y
endif
+= tcrypt.o
```

```
ifeq
(CONFIG_CRYPTO_TEST, y)
obj-y += tcrypt.o
endif
ifndef
CONFIG_CRYPTO_TEST
obj- += tcrypt.o
endif
```

Kbuild Syntax in Operation

Variable expands to conditional

Hoist conditional around assignment

Record all definitions and their configurations

```
obj-$(CONFIG_CRYPTO_TEST) += tcrypt.o
```

```
obj-
ifeq
(CONFIG_CRYPTO_TEST, y)
y
endif
+= tcrypt.o
```

```
ifeq
(CONFIG_CRYPTO_TEST, y)
obj-y += tcrypt.o
endif
ifndef
CONFIG_CRYPTO_TEST
obj- += tcrypt.o
endif
```

Overview

- SuperC
- Kmax
- Bug-finding

Linker Error Bug Finder

1. Use Kmax to find all compilation units
2. Use SuperC to find all function calls, definitions
3. Match function calls with definitions
4. Construct boolean expression for potential bug
5. Use SAT solver to detect the bug

crypto/Makefile

```
obj-$(CONFIG_CRYPTO_BLKCIPHER) += ablkcipher.o
obj-$(CONFIG_CRYPTO_TEST) += tcrypt.o
```

crypto/ablkcipher.c

```
void *crypto_alloc_ablkcipher() {
  return (void*)0;
}
```

crypto/tcrypt.c

```
static void test_cipher() {
  crypto_alloc_ablkcipher();
}
```

1. Kmax

Unit	Condition
tcrypt.o	CRYPTO_TEST
ablkcipher.o	CRYPTO_BLKCIPHER

crypto/ablkcipher.c

```
void *crypto_alloc_ablkcipher() {
  return (void*)0;
}
```

crypto/tcrypt.c

```
static void test_cipher() {
  crypto_alloc_ablkcipher();
}
```

1. Kmax

Unit	Condition
`tcrypt.o`	`CRYPTO_TEST`
`ablkcipher.o`	`CRYPTO_BLKCIPHER`

2. SuperC

Unit	Functions	Condition
`ablkcipher.o`	defines `crypto_alloc_ablkcipher`	TRUE
`tcrypt.o`	calls `crypto_alloc_ablkcipher`	TRUE

1. Kmax

Unit	Condition
`tcrypt.o`	`CRYPTO_TEST`
`ablkcipher.o`	`CRYPTO_BLKCIPHER`

2. SuperC

Unit	Functions	Condition
`ablkcipher.o`	defines `crypto_alloc_ablkcipher`	TRUE
`tcrypt.o`	calls `crypto_alloc_ablkcipher`	TRUE

3. Match

Unit	Call	Defining Unit
`tcrypt.o`	`crypto_alloc_ablkcipher`	`ablkcipher.o`

1. Kmax

Unit	Condition
`tcrypt.o`	`CRYPTO_TEST`
`ablkcipher.o`	`CRYPTO_BLKCIPHER`

2. SuperC

Unit	Functions	Condition
`ablkcipher.o`	defines `crypto_alloc_ablkcipher`	TRUE
`tcrypt.o`	calls `crypto_alloc_ablkcipher`	TRUE

3. Match

Unit	Call	Defining Unit
`tcrypt.o`	`crypto_alloc_ablkcipher`	`ablkcipher.o`

4. Model

`(CRYPTO_TEST && !CRYPTO_BLKCIPHER)`

Kconfig Constraints

- Definition of Linux feature model
  - Describes legal configurations
- Discrepancies possible [Tartler et al., ECCS ’11]

Kconfig Constraints

- Definition of Linux feature model
  - Describes legal configurations
  - Discrepancies possible [Tartler et al., ECCS ’11]
- `tcrypt.c` bug fixed by adding Kconfig constraint
  - if `CRYPTO_TEST` then `CRYPTO_BLKCIPHER`
  - Can automate suggesting with abduction

Modeling the Bug

$$C_{\text{kconfig}} \wedge C_{\texttt{callunit}} \wedge C_{\texttt{call}} \wedge \neg(C_{\texttt{defunit}} \wedge C_{\texttt{def}})$$

Modeling the Bug

Function call's condition

$$C_{\texttt{kconfig}} \wedge C_{\texttt{callunit}} \wedge C_{\texttt{call}} \wedge \neg(C_{\texttt{defunit}} \wedge C_{\texttt{def}})$$

Calling unit's condition

Modeling the Bug

Function call's condition

Definition's condition

$$C_{\text{kconfig}} \wedge C_{\text{callunit}} \wedge C_{\text{call}} \wedge \neg(C_{\text{defunit}} \wedge C_{\text{def}})$$

Calling unit's condition

Defining unit's condition

Modeling the Bug

Kconfig constraints

Function call's condition

Definition's condition

$$C_{\text{kconfig}} \wedge C_{\text{callunit}} \wedge C_{\text{call}} \wedge \neg(C_{\text{defunit}} \wedge C_{\text{def}})$$

Calling unit's condition

Defining unit's condition

Preliminary Results

- 3,915,862 checked function calls
  - 1,007,606 labeled potential bugs
- SAT solver can tackle the boolean expressions
  - 12,673 Kconfig expressions for each function call
- Known bugs work
- There are false positives
  - Macros guarding function definitions
  - Same function defined in multiple units

Future Work

- Further develop of bug finding
- Use for real program analysis
  - Data-flow
  - Abstract interpretation
  - SMT
- Use in software engineering tools

Summary

- SuperC parses all configurations of C files
  - Supports semantic actions and parsing context
- Kmax extracts Linux build system information
  - Extended to emit Kconfig constraints
- Simple bug finder combines SuperC and Kmax
  - Scales to size of Linux kernel source