THE LINUX FOUNDATION OPEN SOURCE SUMMIT EUROPE | Embedded Linux Conference Europe

Oct. 27th, 2020

When You Come to a Fork in the Road, Take It

Finding Configuration Constraints from Kconfig, Kbuild, and the C Preprocessor

Paul Gazzillo
University of Central Florida

UNIVERSITY OF CENTRAL FLORIDA

#ossummit
@paul_gazzillo
https://paulgazzillo.com

THE LINUX FOUNDATION

the kernel has tons of configuration options

```
.config - Linux/x86 5.4.0 Kernel Configuration
                      Linux/x86 5.4.0 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus
  ----).  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for
  Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

          *** Compiler: gcc (Ubuntu 9.2.1-9ubuntu2) 9.2.1 20191008 ***
          General setup  --->
      [*] 64-bit kernel
          Processor type and features  --->
          Power management and ACPI options  --->
          Bus options (PCI etc.)  --->
          Binary Emulations  --->
          Firmware Drivers  --->
      [*] Virtualization  --->
          General architecture-dependent options  --->
      [*] Enable loadable module support  --->
      [*] Enable the block layer  --->
          IO Schedulers  --->
          Executable file formats  --->
          Memory Management options  --->
      [*] Networking support  --->
      v(+)

             <Select>    < Exit >    < Help >    < Save >    < Load >
```

this configurability brings maintenance challenges

over 17,000 configuration options
about 20 million source lines of code
over 20,000 C files

and growing!

there are serious cases of unexpected interactions between configuration options

Variability Bugs in Highly-Configurable Systems: A Qualitative Analysis

IAGO ABAL, IT University of Copenhagen, Denmark
JEAN MELO, IT University of Copenhagen, Denmark
ȘTEFAN STĂNCIULESCU, IT University of Copenhagen, Denmark
CLAUS BRABRAND, IT University of Copenhagen, Denmark
MÁRCIO RIBEIRO, Federal University of Alagoas, Brazil
ANDRZEJ WĄSOWSKI, IT University of Copenhagen, Denmark

L	bug type	CWE
7	**declaration errors:**	
4	undefined function	–
2	undeclared identifier	–
1	multiple function definitions	–
	undefined label	–
10	**resource mgmt. errors:**	
5	uninitialized variable	457
1	memory leak	401
1	use after free	416
2	duplicate operation	675
1	double lock	764
	file descriptor leak	403
11	**memory errors:**	
4	null pointer dereference	476
3	buffer overflow	120
3	read out of bounds	125
1	write on read only	–
8	**logic errors:**	
5	fatal assertion violation	617
2	non-fatal assertion violation	617
1	behavioral violation	440
4	**type errors:**	
2	incompatible types	843
1	wrong number of func. args.	685
1	void pointer dereference	–
2	**dead code:**	
1	unused variable	563
1	unused function	561
1	**arithmetic errors:**	
1	numeric truncation	197
	integer overflow	190
	validation errors:	
	OS command injection	078
43	**TOTAL**	–

there are even more pernicious examples of bad combinations of configuration options

Wednesday, September 26, 2018

A cache invalidation bug in Linux memory management

Posted by Jann Horn, Google Project Zero

"This exploit shows how much impact the kernel configuration can have on how easy it is to write an exploit for a kernel bug. While simply turning on every security-related kernel configuration option is probably a bad idea, some of them - like the `kernel.dmesg_restrict` sysctl - seem to provide a reasonable tradeoff when enabled." - Jann Horn, Google Project Zero

first steps towards tackling the maintenance challenges of configurability

to test a patch, kernel maintainers need .config files. can we automatically generate the relevant .config files?

Julia Lawall
Inria/LIP6

given a patch, what configurations does it affect? (jmake, lawall et al)

given a bug, what configurations does it appear in? (config-bisect)

what's a minimal configuration that includes specific source? (config-bisect)

what code is no longer configurable in the kernel? (undertaker, tarlet et al)

a common problem: mapping code back to the configurations that control that code

configuration localization:
given some program behavior, what are all the configurations which include that behavior?

if we can automate configuration localization, then we can enable automated tools for many problems

how does Kbuild work and how can we do configuration localization?

what does linux's build system do?

let’s look at the phases of build process

let’s look at the phases of build process

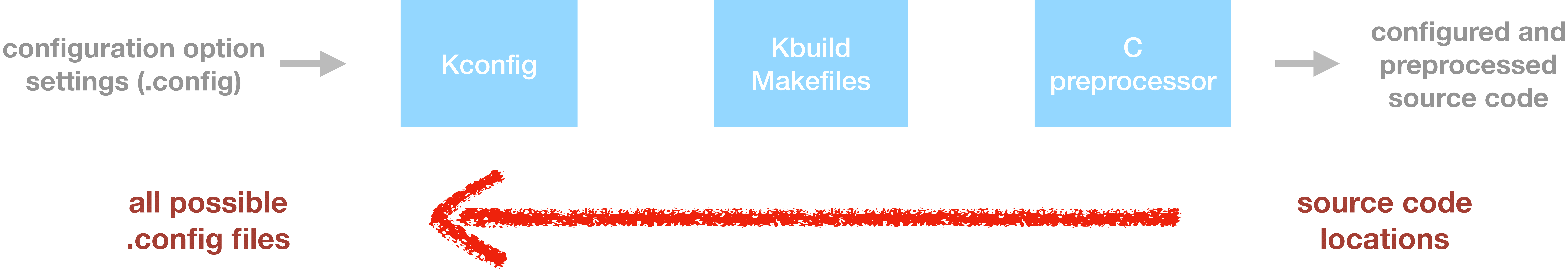
let’s look at the phases of build process

let's look at the phases of build process

let's look at the phases of build process

the build system as code generation using metaprogramming

configuration localization is finding the *inverse* of the build process



each phase of the build encodes rules to control the inclusion and exclusion of source code

```
fs/ufs/super.c:

#ifdef CONFIG_UFS_DEBUG
/*
 * Print contents of ufs_super_block, useful for debugging
 */
static void ufs_print_super_stuff(struct super_block *sb,
                                  struct ufs_super_block_first *usb1,
                                  struct ufs_super_block_second *usb2,
                                  struct ufs_super_block_third *usb3)
{
        u32 magic = fs32_to_cpu(sb, usb3->fs_magic);
// ...
#endif
```

each phase of the build encodes rules to control the inclusion and exclusion of source code

```
fs/ufs/Makefile:

obj-$(CONFIG_UFS_FS) += ufs.o
ufs-objs := balloc.o cylinder.o dir.o file.o ialloc.o inode.o \
            namei.o super.o util.o
```

each phase of the build encodes rules to control the inclusion and exclusion of source code

```
fs/ufs/Kconfig:

config UFS_DEBUG
        bool "UFS debugging"
        depends on UFS_FS

config UFS_FS
        tristate "UFS file system support (read only)"
        depends on BLOCK
```

we can use boolean logic to represent the "buildability" of code at each step

we can use boolean logic to represent the "buildability" of code at each step

configuration localization then becomes the boolean satisfiability problem

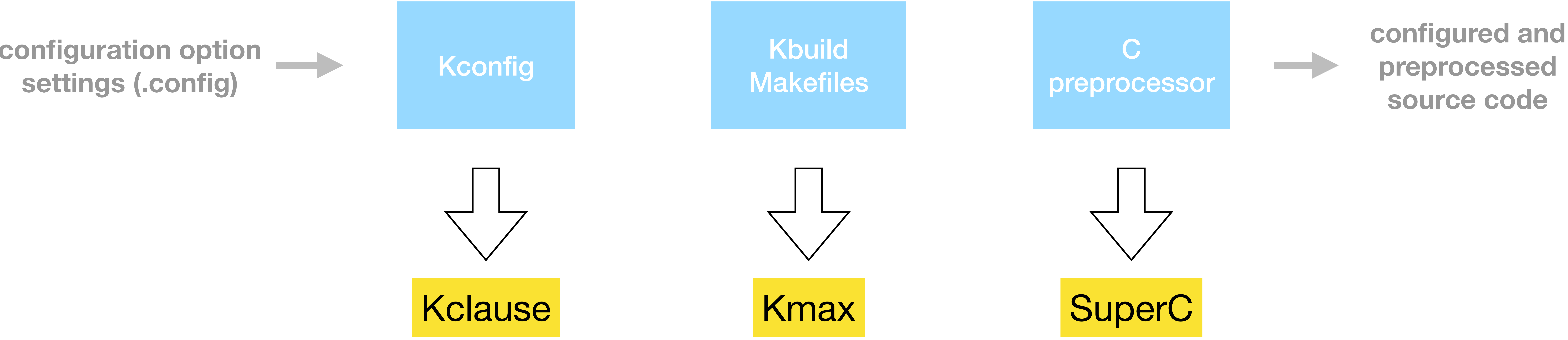
using static analysis to extract constraints from the build system

when you come to a fork in the road, take it

static analysis works by following *both* sides of all conditional branches

all these tools record branch path conditions from their target build system component

tooling



SuperC does configuration-preserving C preprocessing (and parsing)

does macro expansion and header inclusion

but leaves preprocessor conditionals in place

stores preprocessor conditions as symbolic boolean formulas

“SuperC: Parsing All of C by Taming the Preprocessor” by
Paul Gazzillo and Robert Grimm (PLDI 2012)

https://github.com/paulgazz/superc

Kmax collects Kbuild Makefile conditions for each source file's constraints

does static analysis of the Kbuild-style Makefiles

preserves path conditions as boolean formulas

finds a formula for each source file (modulo bugs)

"Kmax: Finding All Configurations of Kbuild Makefiles Statically" by Paul Gazzillo (ESEC/FSE 2017)

https://github.com/paulgazz/kmax

Kclause converts Kconfig files into logical fomulas

Kconfig combines Boolean formulas with “depends on”, “select”, etc

Kclause turns these into Boolean logic

“A depends on B” => “A implies B”

lots of subtley around reverse dependencies, choices, and more

Paper still being written with students
Jeho Oh and Necip Yildiran

tool available as part of Kmax:
https://github.com/paulgazz/kmax

klocalizer prototype and demo

klocalizer finds .config files for given source files

combines Kconfig and Kbuild Makefile constraints from Kclause and Kmax

(supporting preprocessor constraints is work in progress)

takes one or more source file names

constructs a boolean formula representing the file’s build conditions

produces one or more .config files that build a kernel including the file(s)

demo

conclusion

conclusion

the kernel’s extreme configurability brings challenges

automatic configuration localization can help automate several maintenance tasks

static analysis of the build system finds configuration constraints (Kclause, Kmax, and SuperC)

the Klocalizer prototype localizes configurations for given C files

future work: continue developing the tooling, finding configuration bugs

https://configtools.org