FOSD Online Meeting

April 13-15, 2021

April 15th, 2021

Helping Linux Maintainers Localize Configurations

Progress Towards a Comprehensive Solution

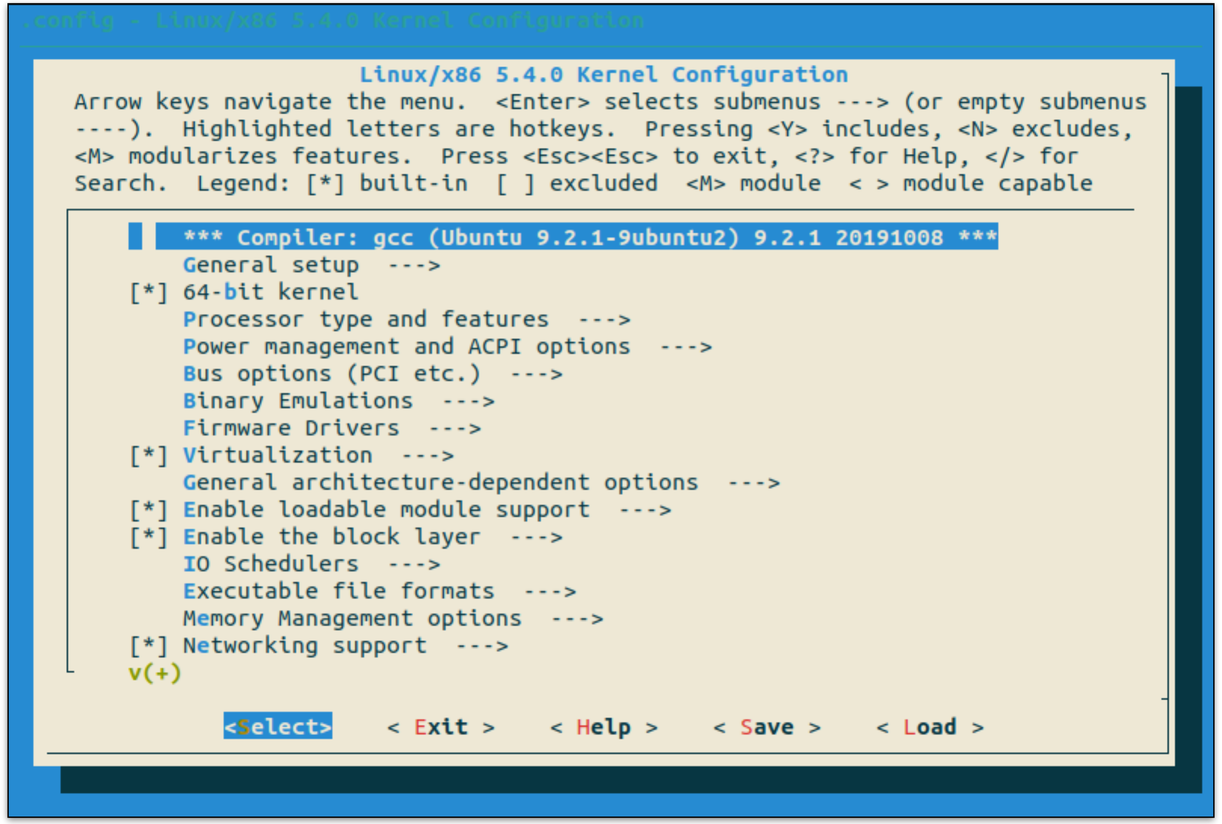
Paul Gazzillo
University of Central Florida

UNIVERSITY OF
CENTRAL FLORIDA

https://paulgazzillo.com

@paul_gazzillo

the linux kernel has tons of configuration options



this configurability brings maintenance challenges

over 15,000 configuration options
about 20 million source lines of code
over 20,000 C files

and growing!

maintainers need a configuration file to test a patch

can we automatically figure out the right .config files to use given a patch?

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, tarler et al)

a common problem: mapping code back to the configuration specifications 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

SPLC 2018 challenge case

Localizing Configurations in Highly-Configurable Systems

Paul Gazzillo
University of Central Florida
paul@pgazz.com

Ugur Koc
University of Maryland, College Park
ukoc@cs.umd.edu

ThanhVu Nguyen
University of Nebraska-Lincoln
tnguyen@cse.unl.edu

Shiyi Wei
University of Texas at Dallas
swei@utdallas.edu

ABSTRACT

The complexity of configurable systems has grown immensely, and it is only getting more complex. Such systems are a challenge for software testing and maintenance, because bugs and other defects can and do appear in any configuration. One common requirement for many development tasks is to identify the configurations that lead to a given defect or some other program behavior. We distill this requirement down to a challenge question: given a program location in a source file, what are valid configurations that include the location? The key obstacle is scalability. When there are thousands of configuration options, enumerating all combinations is exponential and infeasible. We provide a set of target programs of increasing difficulty and variations on the challenge question so that submitters of all experience levels can try out solutions. Our hope is to engage the community and stimulate new and interesting approaches to the problem of analyzing configurations.

software, such as Linux, BusyBox, Firefox, and Apache, have millions or billions of configurations. While bugs can and do appear in any configuration [1], there are simply too many configurations to test them all separately. With the proliferation of Internet-of-things devices, maintenance and testing highly-configurable systems are even more essential, given the variety of devices using different configurations of the same software.

Many aspects of software maintenance are impeded by configurability, including testing, localizing and repairing bugs, security auditing, and finding code smells and dead code. All must apply to every configuration of the system. One simple distillation of these tasks is to identify interesting configurations: *Given some point of interest in a program, what are the configurations that reach that point of interest?* A point of interest can be a particular line, file, program slice, bug, security violation, or some other subset of program behavior. Ideally, we would like to discover the complete space of configurations that reach the given point.

PCLocator: A Tool Suite to Automatically Identify Configurations for Code Locations

Elias Kuiter
Otto-von-Guericke-University
kuiter@ovgu.de

Sebastian Krieter
Harz University of Applied Sciences
Otto-von-Guericke-University
skrieter@hs-harz.de

Jacob Krüger
Otto-von-Guericke-University
jkrueger@ovgu.de

Kai Ludwig
Harz University of Applied Sciences
kludwig@hs-harz.de

Thomas Leich
Harz University of Applied Sciences
METOP GmbH
tleich@hs-harz.de

Gunter Saake
Otto-von-Guericke-University
saake@ovgu.de

ABSTRACT

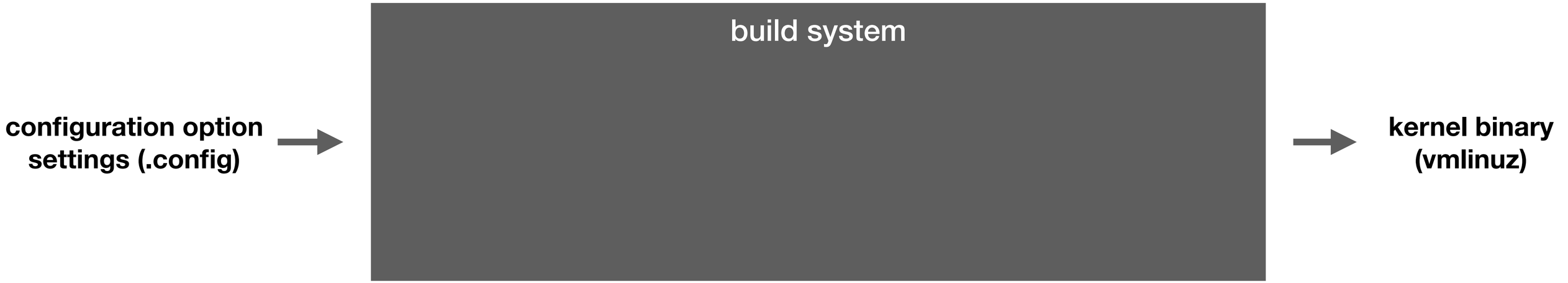
The source code of highly-configurable software is challenging to comprehend, analyze, and test. In particular, it is hard to identify all configurations that comprise a certain code location. We contribute PCLocator, a tool suite that solves this problem by utilizing static analysis tools for compile-time variability. Using BusyBox and the Variability Bugs Database (VBDb), we evaluate the correctness and performance of PCLocator. The results show that we are able to analyze files in a matter of seconds and derive correct configurations in 95% of all cases.

that specifies options for the presence or absence of each feature. If a configuration satisfies all feature dependencies (e.g., requires, alternatives), it is *valid* and a product can be derived.

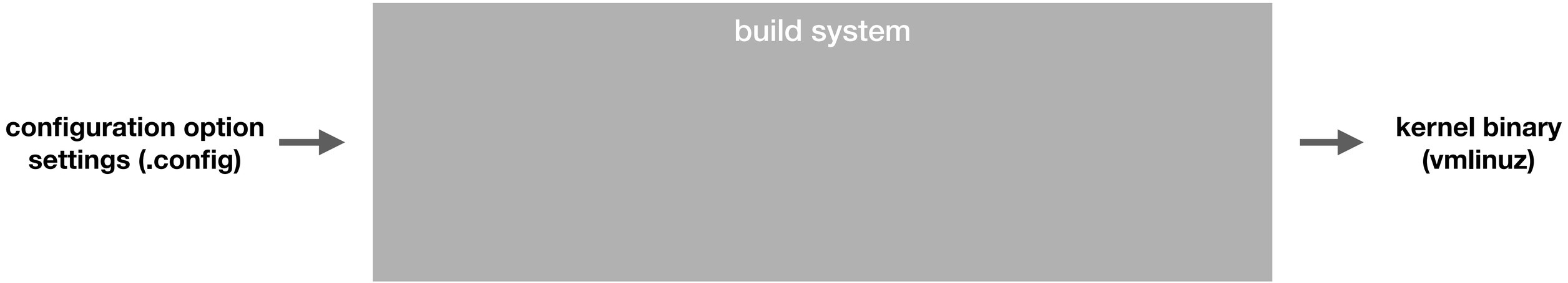
A high number of configuration options, which may be scattered across different variability mechanisms, hampers the comprehension of source code, its analysis, and especially testing it. For example, Linux comprises over 10,000 configuration options that allow for millions of products [17]. Also, Linux' configuration options and their dependencies are implemented in a combination of the C preprocessor and Kconfig files, which alone comprise more than 110,000 lines of code. In particular, it is important in such a context

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

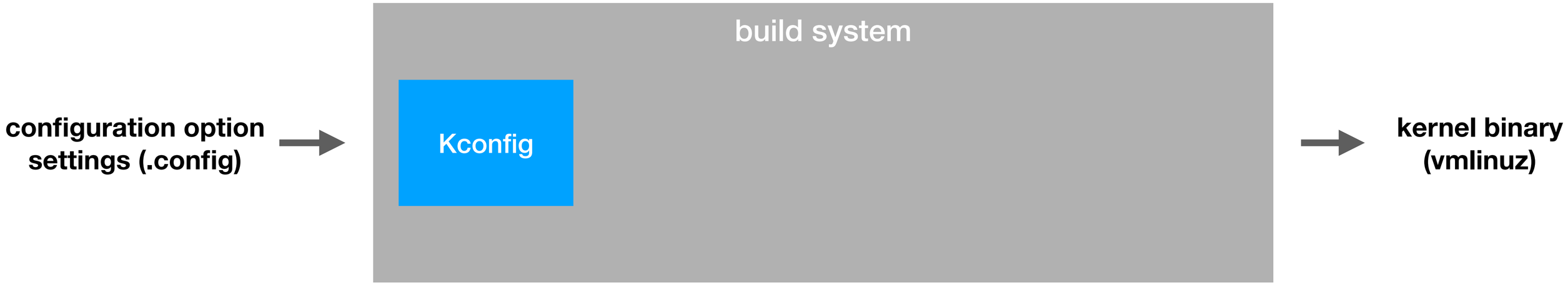
what does linux’s build system do?



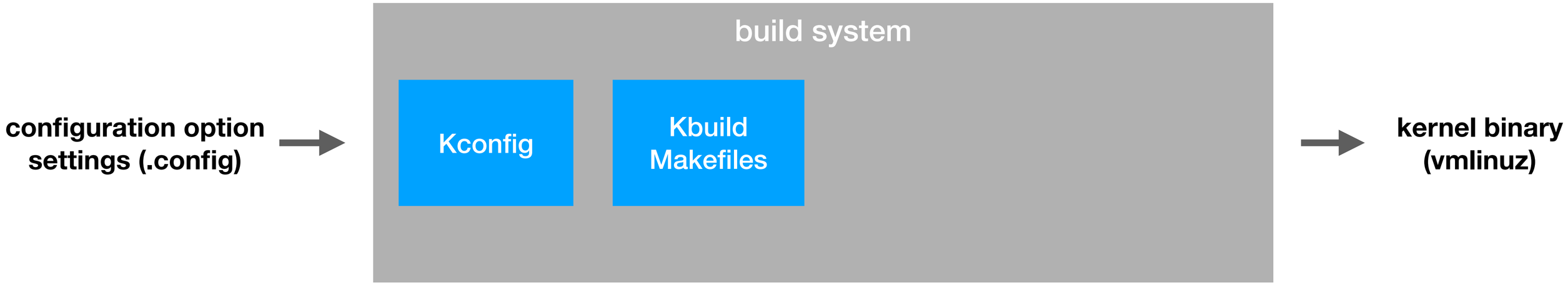
let’s look at the phases of build process



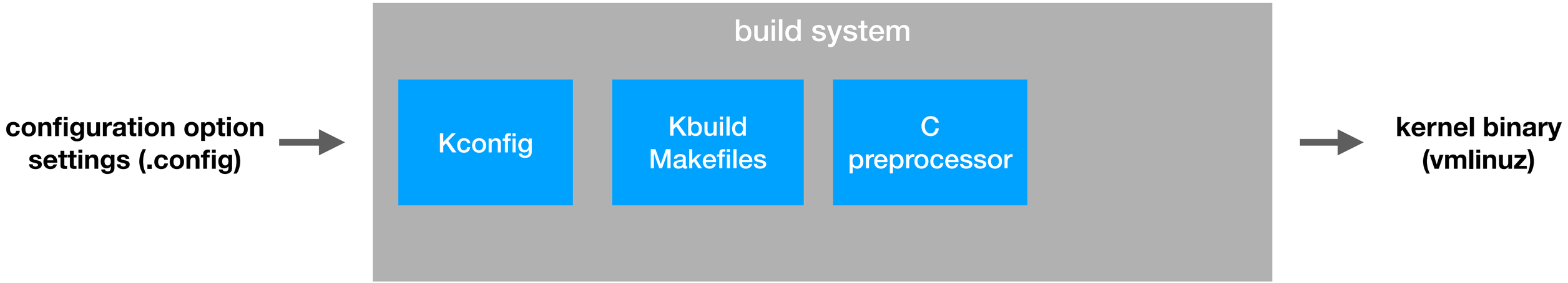
let’s look at the phases of build process



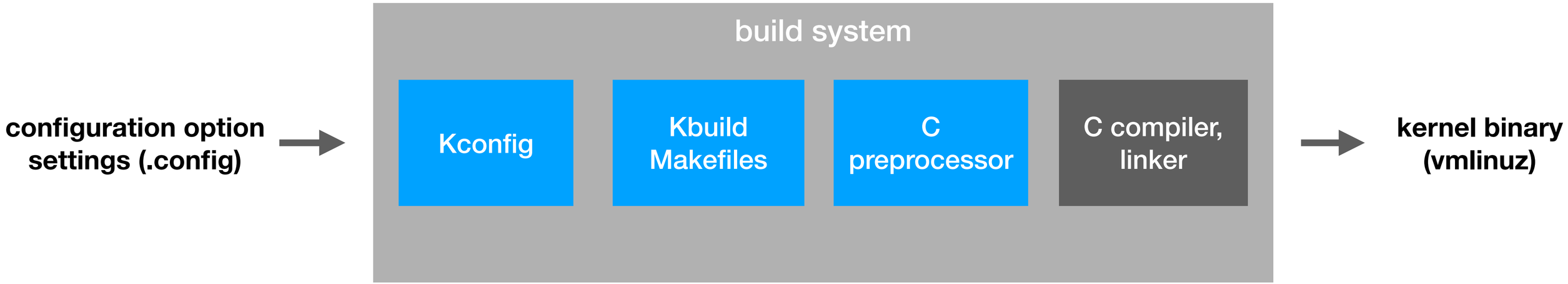
let’s look at the phases of build process



let’s look at the phases of build process

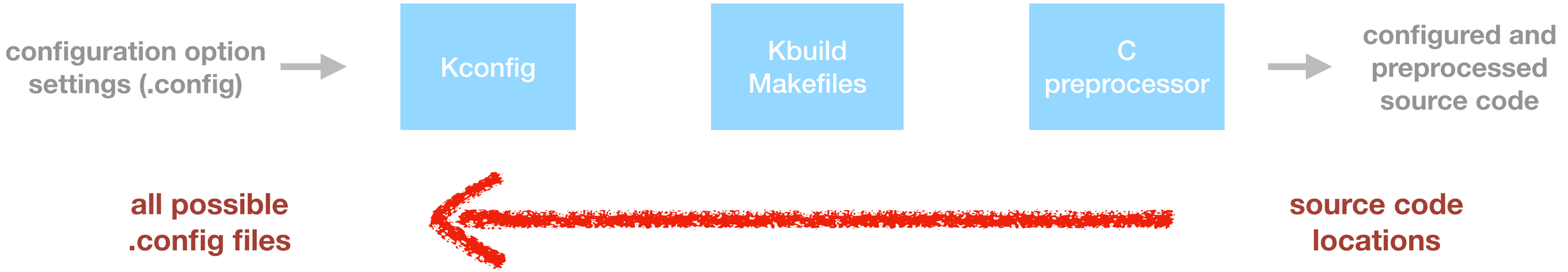


let’s look at the phases of build process



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

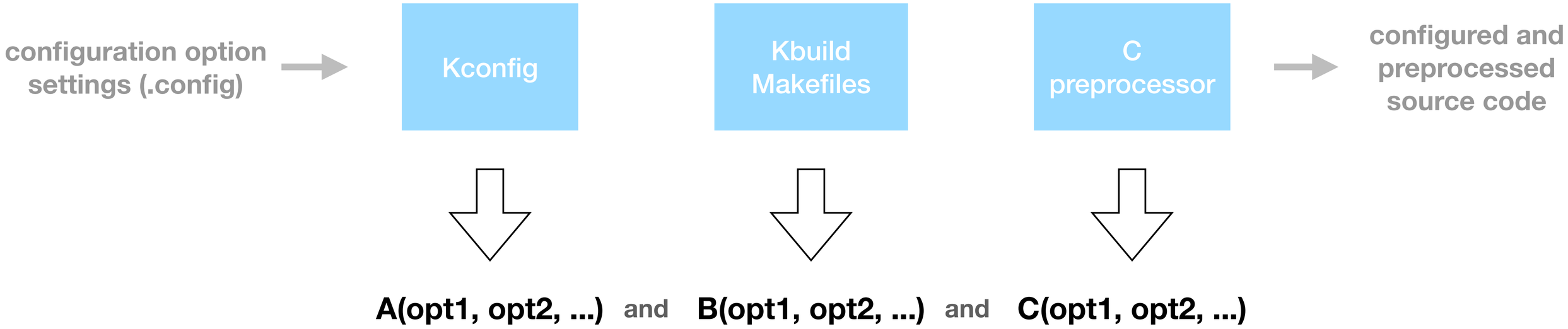
fs/ufs/Kconfig:

```
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

step 1: generate constraints for given source code

build_constraints(opt1, opt2, ...)

step 2: find solutions with a SAT/SMT solver to get .config files

there are many tools that extract linux feature models

lvat

kconfigreader

dumpconf

kclause

kbuildminer

kmax

typechef

superc

...

(many more)

1. Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 421--432.

2. Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. 2007. Design recovery and maintenance of build systems. In *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*. 114--123.

3. Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development (FOSD '13)*. ACM, New York, NY, USA, 1--8.

4. Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Feature-to-code Mapping in Two Large Product Lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10)*. Springer-Verlag, Berlin, Heidelberg, 498--499. http://dl.acm.org/citation.cfm?id=1885639.1885698

5. Renée C. Bryce and Charles J. Colbourn. 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology* 48, 10 (Oct. 2006), 960--970.

6. D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. 1996. The combinatorial design approach to automatic test generation. *IEEE Software* 13, 5 (Sept. 1996), 83--88.

7. M.B. Cohen, Amanda Swearngin, Brady Garvin, Jacob Swanson, Justyna Petke, Kaylei Burke, Katie Macias, Ronald Decker, Wayne Motycka, and Zhen and Wang. {n. d.}. Combinatorial Interaction Testing Portal. ({n. d.}). http://cse.unl.edu/cit-portal, Accessed on 2018-01-16.

8. M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. 2003. Constructing test suites for interaction testing. In *25th International Conference on Software Engineering, 2003. Proceedings.* 38--48.

9. Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying Feature-based Model Templates Against Well-formedness OCL Constraints. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06)*. ACM, New York, NY, USA, 211--220.

10. Gulsen Demiroz and Cemal Yilmaz. 2012. Cost-aware combinatorial interaction testing. In *Proceedings of the Internatinoal Conference on Advances in System Testing and Validation Lifecycles.* 9--16.

11. Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. A Robust Approach for Variability Extraction from the Linux Build System. In *Proceedings of the 16th International Software Product Line Conference - Volume 1 (SPLC '12)*. ACM, New York, NY, USA, 21--30.

12. Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikshat, and Daniel Lohmann. 2012. A robust approach for variability extraction from the Linux build system. 21--30.

13. Emine Dumlu, Cemal Yilmaz, Myra B. Cohen, and Adam Porter. 2011. Feedback Driven Adaptive Combinatorial Testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, New York, NY, USA, 243--253.

14. Alejandra Garrido and Ralph Johnson. 2005. Analyzing Multiple Configurations of a C Program. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*. IEEE Computer Society, Washington, DC, USA, 379--388.

but extracting models is not the whole story

unifying output

scaling to linux

high compatibility with linux configuration languages

high fidelity to build system behavior

producing drop-in .config files

quality-of-life features for users

plocalizer: creates .conf files for patches

currently localizes entire .c files (kconfig and kbuild)

currently integrating preprocessor conditions

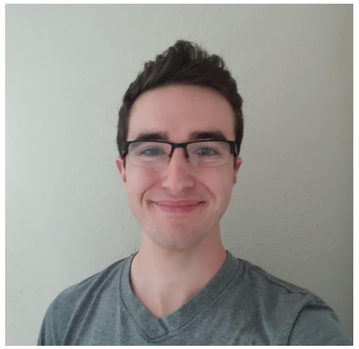
still investigating runtime conditions, e.g., IS_ENABLED

evaluating efficacy on real-world patches

upcoming challenge: patches involving configuration specifications themselves

graduate students currently working on this

Necip Yildiran



Julian Braha

tool demo video today at 18:30 CEST

conclusion

the kernel’s extreme configurability brings maintenace challenges

automatic configuration localization can help automate several maintenance tasks

build system analysis configuration constraints

the plocalizer tool will localize configurations for given patches

prototype is working for a subset of the problem

https://github.com/paulgazz/kmax