Toward Detection and Characterization of Variability Bugs in Configurable C Software: An Empirical Study

Austin Mordahl
Department of Computer Science
University of Texas at Dallas
Richardson, TX, USA
Email: austin.mordahl@utdallas.edu

***Abstract*—Variability in C software is a useful tool, but critical bugs that only exist in certain configurations are easily missed by conventional debugging techniques. Even with a small number of features, the configuration space of configurable software is too large to analyze exhaustively. Variability-aware static analysis for bug detection is being developed, but remains at too early a stage to be fully usable in real-world C programs. In this work, we present a methodology of finding variability bugs by combining variability-oblivious bug detectors, static analysis of build processes, and dynamic feature interaction inference. We further present an empirical study in which we test our methodology on two highly configurable C programs. We found our methodology to be effective, finding 88 true bugs between the two programs, of which 64 were variability bugs.**

***Index Terms*—static analysis, configurable C software, variability bugs**

I. Research Problem and Motivation

Compile-time variability allows large C programs to be tailored to a wide variety of use cases. This variability is achieved through the use of *features*, which are used to determine which parts of the codebase will be included in the final product [1]. While this variability proves useful in this regard, it can mask bugs in the codebase that only manifest in configurations with certain feature combinations [2] [3].

These bugs, called *variability bugs*, have been shown to exist in significant numbers in commonly used C programs. Abal et al. [4], for instance, demonstrated the existence of variability bugs in the Linux kernel and other C programs through manual inspection of bug-fixing patches. Rhein et al. [5] attempted to find bugs preemptively, developing seven variability-aware static analyses. These analyses produced impressive results on real-world C programs; however, they are still limited, not supporting all GNU C extensions and necessitating the exclusion of some files in order to work. In short, the work of finding variability bugs is still nascent, requiring either extensive manual inspection or the use of new, specialized tools.

To this end, we aim to develop a strategy that uses existing static analysis tools to find previously unknown variability bugs in highly configurable C programs. This strategy would allow developers to take advantage of the static analysis tools they are already familiar with to find variability bugs as part of the quality assurance process. Furthermore, studying new variability bugs would inform the design of variability-aware tools in the future. Two primary challenges arise as part of this goal. The first is *how we can find variability bugs with variability-oblivious static analyzers*. Then, given a variability bug, the second challenge is *determining what feature or feature interaction cause that bug to manifest*.

We address these challenges with the following contributions:

1) We develop a methodology that combines existing static bug detectors, static analysis of build systems, dynamic interaction inference, and configuration sampling to semiautomatically detect new variability bugs in real-world C software.
2) We conduct an empirical study involving two highly configurable C programs that shines light on the nature of variability bugs in the wild.

II. Detecting & Characterizing Variability Bugs

In this section, we describe our methodology in detail.

Sample Generation The number of configurations for programs even with a relatively small feature space is far too big to exhaustively analyze. As we aim to use variability-oblivious static analysis, we sample configurations from the configuration space with the feature information exposed in systems that use KConfig. Using the KMax [6] tool to obtain configuration information, and the methodology described by Oh et al. [1], we generate a configuration sample that 1) provides high feature coverage and 2) only generates valid configurations (i.e.,configurations that can be compiled successfully). We generate a sample of 1,000 valid configurations for each target program.

Postprocessing and Classifying Warnings We next run our detector suite on each configuration in the sample and obtain warnings (i.e., bug reports emitted by a bug detector). With 1000 configurations, the collection of warnings generated by just a single bug detector on a single target program quickly becomes too large to evaluate manually; however, since most of the codebase is the same between different configurations

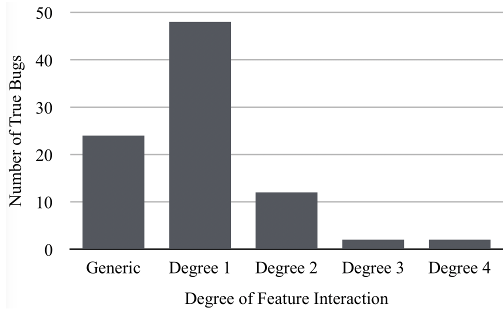

Fig. 1. Number of bugs plotted against degree of feature interaction. The latter refers to the number of features are associated with the bug. *Generic* refers to non-variability bugs.

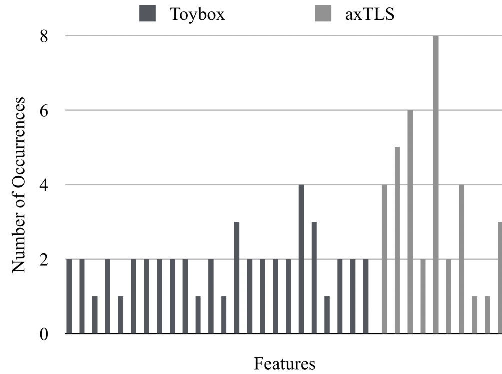

Fig. 2. Number of times a feature is involved in a bug-causing interaction plotted against feature. Feature names are excluded for readability, but each vertical bar represents one feature.

of the same program, there will be many duplicated warnings in the collection. By generating a hash value for each bug report, and comparing those hashes to weed out duplicates, we reduce the number of warnings to a manageable level. We next manually classify the unique warnings as true or false, referring to the reported location and description. We repeat this process for each bug detector.

Determining Feature Interactions We use a semiautomatic approach to determine feature interactions for true positive bugs, combining dynamic feature inference using iGEN [7] and manual code inspection. iGEN uses the list of configurations in which a bug occurred to infer the feature constraints that give those configurations. The combined approach allows us to effectively determine the feature interactions responsible for all found bugs with high precision.

Tools and Target Programs When choosing target programs for the empirical study, our goal was to find software that is 1) highly configurable, 2) under active maintenance, and 3) exposes feature constraints through KCONFIG. We thus chose two programs: axTLS 2.1.4 [8], which provides 84 features and 2.0×10^{12} possible configurations; and Toybox 0.7.5 [9], which provides 316 features and 1.4×10^{81} possible configurations. Similarly, when choosing static analysis tools to use, we wanted tools that 1) work on C code, 2) emit bug warnings instead of other code quality metrics, and 3) are free to use. The third criterion we enforced to ease reproducibility. The three static analysis tools we settled on are cppcheck 1.72 [10], Facebook Infer 0.15.0 [11], and clang 4.0's built-in static analyzer [12].

III. RESULTS AND FUTURE WORK

A. Results

In total we have found 88 bugs. 42 are from Toybox, and 46 are from axTLS. Of these bugs, 64 are variability bugs, 16 of which are caused by the conjunction of two or more features. Over half of the variability bugs are caused by enabling or disabling a single feature, with bug count decreasing as the number of features increases (see Figure 1).

Out of all found variability bugs, seven of them are associated with one or more disabled features: three are associated with only disabled features, and the other four are associated with some combination of enabled and disabled features. All seven of these bugs are in axTLS (all Toybox bugs were only associated with enabled features). This means while one could find all of the bugs we found on Toybox by running our detector suite on Toybox's *allyesconfig*, neither the *allyesconfig* nor the *allnoconfig* provided with these programs are sufficient to find bugs in axTLS. Whether the configuration information of a program could be used to choose whether to analyze the *allyesconfig* or sample the configuration space is an interesting question for future studies.

We also observe from Figure 2 that axTLS' features tend to be involved in more complex bug-causing interactions than those in Toybox. This is likely because of how features are used in these programs: Toybox's features are more independent of each other than those in axTLS. These data suggest that the implementation of variability can affect the nature of variability bugs in a program; how exactly this happens is an interesting research question, which we plan to address by running our experiments on more programs.

B. Evaluation and Future Work

Overall, our results suggest that variability-oblivious static analysis can be used to find variability bugs in C software. We are continuing to expand the scope of the empirical study by adding more target programs and bug detectors. Namely, we have added BusyBox 1.28.0 [13] and the Linux kernel 4.17.6 [14] as additional target programs, and CBMC 5.3 [15] and IKOS 1.3.r1.dd5a747 [16] as additional bug checkers. This should give us a fuller look at the nature of variability bugs across a variety of programs, and enable us to make further inferences about how the implementation of variability affects the nature of variability bugs. We also plan to make our bug database available in the near future, to provide a benchmark for future works to compare against. We will add to this database as we obtain results from more programs and tools.

ACKNOWLEDGMENT

This research is supported by NSF-1816951.

References

[1] J. Oh, P. Gazzillo, and D. Batory, "Multi-objective optimization in large software product lines," The University of Texas at Austin, Department of Computer Science, Tech. Rep. TR-18-02, 2018.

[2] H. Post and C. Sinz, "Configuration lifting: Verification meets software configuration," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 347–350.

[3] M. Attariyan and J. Flinn, "Using causality to diagnose configuration bugs." in *USENIX Annual Technical Conference*, 2008, pp. 281–286.

[4] I. Abal, J. Melo, x. Stănciulescu, C. Brabrand, M. Ribeiro, and A. Wasowski, "Variability bugs in highly configurable systems: A qualitative analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 3, pp. 10:1–10:34, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3149119

[5] A. V. Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, "Variability-aware static analysis at scale: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 4, pp. 18:1–18:33, Nov. 2018. [Online]. Available: http://doi.acm.org/10.1145/3280986

[6] P. Gazzillo, "Kmax: Finding all configurations of kbuild makefiles statically," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 279–290. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106283

[7] T. Nguyen, U. Koc, J. Cheng, J. S. Foster, and A. A. Porter, "igen: Dynamic interaction inference for configurable software," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 655–665. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950311

[8] "axTLS." [Online]. Available: http://axtls.sourceforge.net

[9] "Toybox." [Online]. Available: https://github.com/landley/toybox

[10] "cppcheck." [Online]. Available: https://github.com/danmar/cppcheck

[11] "Infer static analyzer." [Online]. Available: https://github.com/facebook/infer

[12] "LLVM." [Online]. Available: https://llvm.org

[13] "Busybox." [Online]. Available: https://busybox.net

[14] "Linux." [Online]. Available: https://www.kernel.org

[15] "CBMC." [Online]. Available: https://github.com/diffblue/cbmc

[16] "Ikos." [Online]. Available: https://github.com/NASA-SW-VnV/ikos