An Empirical Study of Real-World Variability Bugs Detected by Variability-Oblivious Tools

Austin Mordahl
University of Texas at Dallas
USA
austin.mordahl@utdallas.edu

Jeho Oh
University of Texas at Austin
USA
jeho.oh@utexas.edu

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

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

Paul Gazzillo
University of Central Florida
USA
paul.gazzillo@ucf.edu

ABSTRACT

Many critical software systems developed in C utilize compile-time configurability. The many possible configurations of this software make bug detection through static analysis difficult. While variability-aware static analyses have been developed, there remains a gap between those and state-of-the-art static bug detection tools. In order to collect data on how such tools may perform and to develop real-world benchmarks, we present a way to leverage configuration sampling, off-the-shelf "variability-oblivious" bug detectors, and automatic feature identification techniques to *simulate* a variability-aware analysis. We instantiate our approach using four popular static analysis tools on three highly configurable, real-world C projects, obtaining 36,061 warnings, 80% of which are variability warnings. We analyze the warnings we collect from these experiments, finding that most results are variability warnings of a variety of kinds such as NULL dereference. We then manually investigate these warnings to produce a benchmark of 77 confirmed true bugs (52 of which are variability bugs) useful for future development of variability-aware analyses.

CCS CONCEPTS

• **General and reference → Empirical studies**; • **Software and its engineering → Automated static analysis**; **Software testing and debugging**.

KEYWORDS

static analysis, configurable C software, variability bugs

ACM Reference Format:
Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. 2019. An Empirical Study of Real-World Variability Bugs Detected by Variability-Oblivious Tools. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338967

1 INTRODUCTION

Systems developed in C form some of the largest and most important software infrastructure. This software, such as the Linux kernel or the BusyBox embedded toolkit, is used in a broad range of applications, from large-scale datacenters to millions of Internet-of-Things devices. C programmers use *compile-time variability* to enable a single codebase to be customized to this diverse range of settings. They implement software configurations in the Makefile and C preprocessor to decide which part of the source code is built by the compiler, allowing for billions or trillions of variations of the compiled program.

Variability bugs are bugs that only exist under certain configurations of the software. These are made possible by the build and configuration system, which modify the source code to fit a chosen configuration.[1] Such *configurable code* is necessary for codebases to support a wide variety of hardware and application settings, but studies show it also creates problems for debugging and maintenance of highly configurable codebases [1–3].

Variability bugs create a serious challenge for automatic bug detection. Most static analysis tools operate on one configuration at a time, i.e., are *variability-oblivious*, or use ad-hoc heuristics [4, 5], making them blind to code in other configurations. Checking configurations one-at-a-time is intractable when even small configurable systems have trillions of configurations [6].

To address this problem, researchers have developed *variability-aware* analyses [4, 5, 7–13] that process all configurations simultaneously. To the best of our knowledge, the state-of-the-art variability-aware bug detector applies control- and data-flow analyses to discover bugs such as double free and freeing of static memory [7]. While these advances in bug detection are highly promising, they are still in the early stages when compared to widely-used variability-oblivious bug detectors such as Infer [14], which uses separation logic to find memory safety bugs, and CBMC [15], which uses abstract interpretation and model checking. These analyses, which can detect bugs such as buffer overflow, array-out-of-bounds, and security vulnerabilities, are in wide use.

[1] In this work we focus on variability bugs due to compile-time variability, but such bugs can also be due to run-time variability.

The continued development of variability-aware analyses is important. But given the gap between such tools and state-of-the-art bug detectors, there is still a lack of data about the utility of such analyses when compared to their variability-oblivious counterparts. Moreover, there is a lack of benchmarks to provide ground truths for developing variability-aware analysis. To the best of our knowledge, the largest variability bug database [1] contains bugs encountered by users over the past decade, making it difficult to reproduce these bugs in the original software.

How can we find variability bugs that existing variability-aware analysis may not detect? Ideally, we would like to take off-the-shelf bug detectors and convert them to be variability-aware to gather data and develop benchmarks. But this Herculean task requires large time resources and would itself benefit from having ground-truth benchmarks during development. In order to collect data on how such tools may perform and to develop real-world benchmarks, we propose *simulating* variability-awareness with off-the-shelf bug detectors.

Simulating variability-awareness is based on a simple technique: run bug detectors on a sample of configurations. Sampling of configuration spaces has been studied extensively [10, 16–30], and some previous work has applied a bug detector to samples of configurations [30] in the context of comparing sampling algorithms. But to be a *simulation*, we argue that (i) the sample of configurations should be *representative* of complete variability-awareness and (ii) it should *integrate* the results of the many variability-oblivious runs into variability-aware results. To achieve a representative sample, we use a recent advance in configuration sampling that guarantees uniformity yet scales to colossal configuration spaces [31]. Previous sampling algorithms could do one or the other, but not both.

To achieve integration we have developed a new simulation framework that wraps existing bug detectors. It automatically applies a bug detector to all sampled configurations, aggregates and deduplicates the resulting warnings, and finds feature interactions, thereby simulating the output of a variability-aware analysis. Using this framework, we analyze the warnings from four off-the-shelf bug detection tools (Infer, CBMC, Clang, and Cppcheck) on three highly-configurable codebases (the axTLS webserver, the Toybox and BusyBox embedded toolkits). These codebases have hundreds of configuration options leading to trillions of valid configurations. Representing weeks of computing time, we collected 36,061 warnings, of which 28,631 (almost 80%) are variability warnings.

We performed data analysis of the resulting warnings that confirms most warnings are due to variability, that shows variability warnings are of many types, even those not currently supported by existing variability-aware analysis tools, and that proves warnings may appear in few configurations, indicating the need for variability-awareness. Perhaps surprisingly, we also show that all warnings produced by any tool/program combination can be covered by a small number of configurations. Our data demonstrate a trade-off between the added complexity of variability-awareness and the need to cover all possible defects, e.g., for safety-critical software.

Finally, we construct a reproducible dataset including variability bugs in recent, real-world software, useful as a benchmark for future analysis development. We manually investigated many of our variability warnings to confirm whether they are true bugs. This investigation yielded a set of 77 true positive bugs showing that our simulation framework can be used to find new variability bugs. We have made our framework and dataset publicly available.[2]

```
1  #ifdef CONFIG_BIGINT_SLIDING_WINDOW
2      for (j = i; j > 32; j /= 5) /* work out an optimum size */
3          window_size++;
4      /* work out the slide constants */
5      precompute_slide_window(ctx, window_size, bi);
6  #else /* just one constant */
7      ctx->g = (bigint **)malloc(sizeof(bigint *));
8      ctx->g[0] = bi_clone(ctx, bi); // warning
9      ctx->window = 1;
10     bi_permanent(ctx->g[0]);
11 #endif
```

Figure 1: An example of a confirmed bug found during our experiments. From axTLS, crypto/bigint.c.

The contributions of this paper are the following:

- A framework that simulates variability-aware analysis by integrating off-the-shelf static analysis tools and running them on uniform random samples of the configuration space (Section 3).
- An empirical evaluation of the warnings produced by four static bug detectors on three highly-configurable C codebases and an analysis of warnings that shows potential defects are indeed obscured by variability (Section 4).
- A real-world variability bug benchmark found by manually investigating the warnings from our evaluation as well as the feature interactions that lead to these bugs. This benchmark is beneficial for tool developers to evaluate future analyses (Section 5).

2 WHAT ARE VARIABILITY BUGS?

Figure 1 is an example of a variability bug, in this case a NULL dereference, found during our experiments running the Infer static analysis tool on axTLS. Line 7 sets `ctx->g` to the return value of `malloc`, which can be NULL, and line 8 dereferences that value, `ctx->g[0]`, without checking for NULL, a well-known defect.[3] This is a variability bug, because it only appears in certain configurations of the codebase.

Even though static analyses such as Infer are sound,[4] they only operate on a single configuration at a time, i.e., the soundness guarantees only apply to the chosen configuration. To see why this is the case, notice that this vulnerable source code is guarded by a *preprocessor conditional* indicated by the `#ifdef`, `#else`, and `#endif` on lines 1, 6, and 11, respectively. Preprocessor conditionals are not part of the C language, but are evaluated before the C source code is compiled to implement variations of the source code.

`CONFIG_BIGINT_SLIDING_WINDOW` is a *preprocessor macro* that the preprocessor conditional tests to decide whether to include either lines 2–5 or lines 7–10, but never both. The macro is set via the configuration system by the user. This usage of the preprocessor illustrates how compile-time variability is typically implemented.

In our example in Figure 1, the NULL dereference bug on line 8 only appears when the `CONFIG_BIGINT_SLIDING_WINDOW` configuration option is *disabled*, because that line is never compiled

[2]https://github.com/paulgazz/kconfig_case_studies/releases/tag/v1.0
[3]CWE-690. https://cwe.mitre.org/data/definitions/690.html
[4]They are sound with respect to a subset of the language semantics, i.e., soundiness [32].

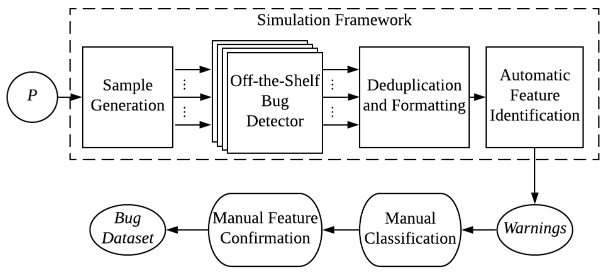

Simulation Framework
P
Sample Generation
Off-the-Shelf Bug Detector
Deduplication and Formatting
Automatic Feature Identification
Warnings
Manual Classification
Manual Feature Confirmation
Bug Dataset

Figure 2: An overview of our approach.

into the final binary. Therefore, if a static analysis tool, even a sound one, is only run on a configuration that enables the option, it will not find the bug. Interestingly, there is a different NULL dereference bug when the option is enabled. Inside of the `precompute_slide_window` function called on line 5 is a code clone of lines 7–10. These two bugs live in mutually exclusive configurations, so finding and fixing one does nothing to help with the other.

If there were a small number of configurations, trying every combination of configuration options might be feasible, but axTLS has 94 configuration options and around two trillion valid combinations of them. Checking all configurations is not feasible, and we do not know a priori which configurations may be vulnerable.

In C, configurations are typically implemented using the preprocessor, as shown in the example above, and with Makefiles or a related build specification language (e.g., Kbuild). The configuration system takes configuration option settings from the user and passes them to the build system and the preprocessor, which then select a subset of the source code to compile and link into the final binary. Prior work shows how extensively configurable code is implemented with the preprocessor and Makefiles [1, 4–6, 33]. Compile-time variability results in codebases that are both highly-configurable and efficient, resulting in a smaller, faster binary, particularly important for low-level systems software.

As we see in Figure 1, however, this configurability also obscures software defects. Previous studies show that such configurable code is dangerous: it has been correlated with more bugs [2] and shown to be more difficult for developers to debug [3]. We use our simulation framework to find variability bugs.

3 FRAMEWORK AND STUDY SETUP

Figure 2 shows an overview of our simulation framework. It takes in a codebase and produces a set of warnings for later manual investigation. The framework itself consists of four main steps: (i) the sample generation, using an existing tool to get uniform random samples for the given codebase [31]; (ii) automatically applying a chosen static analysis tool on all sampled configurations; (iii) aggregating and deduplicating the results, finding which warnings appear in multiple configurations; and (iv) identifying features, using the list of configurations each warning occurred in to infer which configuration options (i.e., features) produce the warning.

We apply this framework to multiple static analysis tools and highly configurable codebases. After running our framework on each combination of tool and codebase, we have a collection of warnings for each combination that simulates variability-aware analysis results. The remainder of Figure 2 shows our process for manually investigating these variability warnings to collect a dataset of true bugs, useful as a benchmark.

We now describe each step of our approach in detail.

Table 1: Details of the target programs.

Program	Version	C SLoC	Options	Valid Configs
axTLS	2.1.4	17,232	94	2.0×10^{12}
Toybox	0.7.5	42,190	316	1.4×10^{81}
BusyBox	1.28.0	162,732	998	2.0×10^{213}

3.1 Sample Generation

Generating a configuration sample is not as simple as randomly combining configuration options, as constraints exist between different options, e.g., some options are mutually exclusive. A configuration is invalid if the constraints defined by the configuration system are not satisfied. Off-the-shelf bug detectors often require compilation of the program to resolve preprocessor directives and generate intermediate representations for the analysis. We thus require every configuration in the sample be valid.

The state-of-the-art tool for generating representative samples from highly-configurable software is Smarch [31]. Smarch is a uniform sampling algorithm for software product lines based on a #SAT solver [34]. Smarch scales to large configurable software by obviating the need to exhaustively enumerate configurations from the constraints. Instead, Smarch generates a unique configuration from a randomly-chosen number on demand. Sampling of valid configurations requires first knowing the set of constraints between configuration options, i.e., the feature model. The state-of-the-art tool, Kclause [31] from the Kmax [35] project is used for automatically extracting these constraints from the Kconfig specification language used in our target projects. It works by interpreting Kconfig language constructs as formal logical models and optimizing them to reduce the size of the reuslting model.

While our framework is modular and accepts samples from any sampling algorithm that generates valid configurations, a uniform random sampling algorithm allows us to draw statistically sound conclusions about the population of configurations based on sample results, within a small margin of error and confidence level. For example, with a configuration sample having a 5% margin of error and 1% confidence level, if we observe that 80% of all bugs detected in a sample are variability bugs, we can say we are 95% sure that between 79% and 81% of all bugs that exist in the target software are variability bugs, allowing us to foresee the results of a variability-aware analysis.

We used the programs listed in Table 1 for our empirical study, because they all (i) are open source, (ii) are highly configurable, and (iii) use Kbuild, which allows to automatically extract configuration constraints with Kclause. For each target program, we generated 1,000 valid configurations. This sample size ensures the results are within a 5% margin of error and 1% confidence interval.

3.2 Using Off-the-Shelf Bug Detectors

We chose four off-the-shelf bug detectors from a list [36] for our investigation: CBMC 5.3 [15], Facebook Infer 0.15.0 [14], Cppcheck

Table 2: Descriptions of the warning types.

Warning Type	Description
Array_Bounds	Array accessed beyond allocated length.
Assertion	Failed assertions either provided by the user or generated automatically by the model checker.
Overflow	Overflow of integers.
NaN	Floating point arithmetic producing NaN (i.e., infinities or the results of computations with infinities).
Pointer	Mismatched types between pointer usage and definition.
Null_Deref	Dereferencing a null pointer.
Dead_Store	Values that are stored in variables but never used.
Memory_Leak	Memory that is dynamically allocated but never freed.
Resource_Leak	Resources (e.g., file descriptors, sockets) that are opened but never closed.
Uninitialized_Val	Values that are used without being initialized.
API	Improper use of various APIs.
Unix_API	improper usage of a Unix API.
Logic_Error	A wide variety of warnings, such as unallowed function calls after `vforks` and undefined behaviors.
Memory_Error	Memory and resource leaks.
Security	Use of insecure function calls (e.g., *vfork*).
Undef_Behavior	Undefined behavior of constructions or expressions.

1.72 [37], and the built-in analyzer in Clang 7.0 [38]. Our criteria for choosing bug detectors were that (i) they worked on C code, and (ii) they produced bug warnings as opposed to other code metrics, such as line counts.

The detectors we chose reported a variety of warnings, defined in Table 2. CBMC reports Array_Bounds, Assertion, NaN, Overflow, Pointer, and Null_Deref. Infer reports Dead_Store Memory_Leak, Null_Deref, Resource_Leak, and Uninitialized_Val. Clang reports API, Unix_APIs, Dead_Store, Logic_Error, Memory_Error, and Security. Cppcheck does not report discrete warning types, so we manually mapped Cppcheck warnings to types that were found in other tools: Array_Bounds, Memory_Error, Null_Deref, Overflow, and Uninitialized_Val. Additionally, we defined the Undef_Behavior warning type for Cppcheck.

These detectors represent two different ways of running an off-the-shelf bug detector. CBMC and Infer attach themselves to the build process (in all of our target programs, this was `make`), collecting information about the build process and generating intermediate representations of the program. For these, we use a script that iteratively runs the appropriate tool as an attachment to the build process. Cppcheck and Clang both run on individual files, so for those, we instead use custom scripts that first preprocess the code. We then run the tools iteratively on each preprocessed file.

In our study, we ran these experiments on two different machines. Preprocessing of BusyBox took place on a Desktop PC with an Intel Core i5-3570K CPU@3.40GHz and 16GB RAM running Debian 9. All other experiments took place on a server with 24 Intel Xeon Silver 4116 CPUs@2.10GHz and 128GB RAM running Ubuntu 16.04 LTS. We used a virtual machine to ensure the consistencies in environment across the machines. In total, experiments took on the order of weeks of processor time to run; however, we were able to reduce this in real time through parallelization.

3.3 Deduplication and Formatting

Although configuration options govern the inclusion or exclusion of different parts of source code, some of a program's codebase is common across all configurations. Therefore, many of the warnings obtained by running a bug detector on each configuration in the sample are duplicates. We perform post-processing to deduplicate these warnings, and then output them in a unified format.

We consider two warnings equivalent if they refer to the same line in the same source code file.[5] We write the unique warnings set for each tool/program combination in JSON format for easier processing. Figure 3 shows the JSON output of the variability bug we discussed in Section 2. The "variability" field is our automated estimation of whether this warning is a variability warning or not. In our study, a warning is estimated to be *generic* (i.e., the variability field is false) if it was detected by the tool in all the configurations in our sample.[6] Otherwise, we regard a warning as *variability* (i.e., the variability field is true). The "automatic_features" field refers to the configuration options identified in the following step.

```
{
  "variability": true,
  "description": "pointer `ctx->g` last assigned on line 1372
        could be null and is dereferenced at line 1373, column 5.",

  "num_configs": 503,
  "tool": "infer",
  "filename": "crypto/bigint.c",
  "line": 1373,
  "type": "NULL_DEREFERENCE",
  "configs": [ "263", "562", "575", "..."],
  "target": "axtls_2_1_4"
  "automatic_features": [ "CONFIG_BIGINT_SLIDING_WINDOW" ]
}
```

Figure 3: The JSON format of the warning in Figure 1. The list of configurations is truncated for space.

3.4 Automatic Feature Identification

We perform automatic feature identification by referencing the list of configurations each warning is present in. For any warning w, let C_w be the set of configurations in which w was emitted, and let C'_w be the set of configurations in which w was not emitted. We automatically determine the set of configuration options F that are common to C_w, and the set of configuration options F' that are common to C'_w. If for any configuration option f, where f denotes the option being turned on and f' denotes the option being turned off, if $f \in F$ and $f' \notin F$, then we select f as the configuration option associated with this warning (similarly, if $f' \in F$ and $f \in F'$, then we consider f' to be associated with w). We do not consider a configuration option to be associated with a warning if f is common across both C_w and C'_w. This methodology can be extended to determine when conjunctions or disjunctions of options are associated with a warning. As an automated algorithm, this process succeeded in estimating associated configuration options in most cases.

[5]Recall that Cppcheck and Clang run on preprocessed files. The preprocessed files contain linemarkers that allow us to map preprocessed code to its original source code line.

[6]A warning we estimate as generic may be a variability warning if there exists some valid configuration (not in the sample) on which a tool cannot report the warning. We believe this estimation is accurate based on our investigation of bugs in Section 5.

3.5 Manual Classification

We perform manual classification of warnings by code inspection and team reviews. Our criterion for determining whether a warning is a true or false positive is whether the description emitted by the bug detector is correct. For each warning, we hand-traced execution around the report, recording the values of variables and pointers. The manual classification was performed by the first author of the paper, and then presented to all members in this project. We as a team examined the warnings classified as true positives and confirmed them. Our primary goal was soundness in our bug dataset, i.e., avoid reporting a warning as a true positive when it was a false positive. It is possible, however, that there were still true positives mistakenly labeled as false positives, since we did not construct exploits.

3.6 Manual Feature Confirmation

We manually verify the results of the automatic feature identification through code inspection. We perform the following tasks:

- Find C preprocessor directives that surround the source code with the bug, and extract the configuration options constraining the directives.
- Find Makefile commands that compile the C file containing the bug, and extract the configuration options that activate the commands.
- To ensure correctness, check whether the configuration options found from the above two steps appeared in the automatic feature identification result.

3.7 Discussion: New Analyses and Programs

Each step in the simulation framework is automated, and is designed to permit "plugging in" new static analysis tools and new codebases to investigate. To add a new codebase, we need two things (1) a feature model describing the configuration constraints for sampling and (2) a small set of shell instructions to configure and build the tool for a given configuration. Technically, our framework can support any sampling algorithm, because the execution engine takes only a set of configurations to evaluate. Currently, our evaluation is on codebases that use Kbuild, the Linux build system, because the sampling tool we use is known to sample these uniformly, but we have tested it on other codebases for which we do not have uniform sampling.

Adding new static analysis tools is as simple as creating a new script that executes the tool. The challenge of supporting a new static analysis tool, however, is due to their idiosyncrasies. For instance, the Clang static analyzer provides a convenient tool called scan-build that uses "poor man's interposition" to analyze an entire project [39]. This tool does not always work, as the manual warns, requiring us to customize the analysis process for codebases we evaluate but are not supported by scan-build. For each combination of tool and codebase, we have created plugins that overcome most of the limitations of the static tools for the complex build systems of our codebases.

4 EMPIRICAL EVALUATION OF WARNINGS

We applied our framework to the three highly configurable codebases shown in Table 1, and the four off-the-shelf tools described in Section 3.2.

The goal of this empirical evaluation is to evaluate what kinds of variability warnings off-the-shelf bug finders detect and to what extent such warnings are affected by configurability. To this end, we ask three research questions:

RQ1 What variability warnings can off-the-shelf bug detectors find?
RQ2 How are variability warnings distributed over the space of sampled configurations?
RQ3 How do our results compare to checking a maximum or minimum configuration?

RQ1. We expect to discover how often and what types of variability warnings are found. If there are many, this confirms prior work that shows variability bugs are a serious problem for highly-configurable software [1, 7]. Moreover, if the types of variability warnings include serious potential defects that current analyses do not support, e.g., Null_Deref, then we provide further justification for the research community to continue developing such analyses. On the other hand, if variability warnings are rare, variability-aware analyses may not be needed.

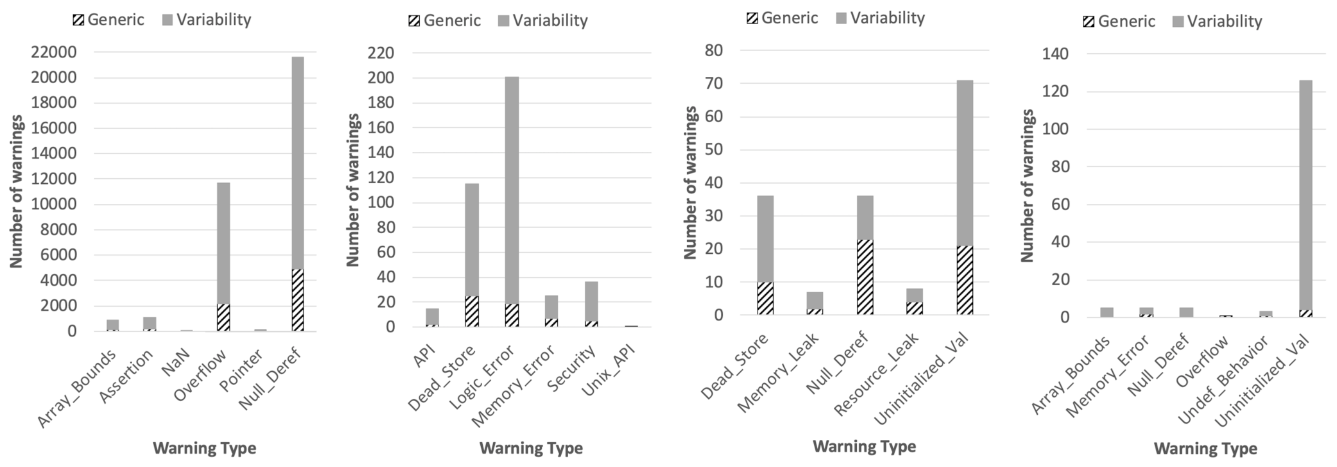
RQ2. Examining the distribution of various types of warnings across the set of samples will show us how difficult it is to identify such warnings without variability-aware analyses. If only certain types of warnings are variability warnings, perhaps the research community should initially focus on analyses for specific kinds of bugs. If more serious warnings tend not to be variability warnings, then this would provide evidence that such analyses may not be necessary. Moreover, we examine how many configurations are required to cover the set of warnings. If many configurations are required, this provides further justification for variability-awareness, while few configurations may mean that further effort on algorithms for configuration coverage is needed.

RQ3. We compare the results of running the bug detectors on the maximum configuration defined by each of our codebases. If we find that the maximum configurations find very few warnings, this provides further justification for variability-aware analyses. Otherwise, such analyses may not be much more successful than selecting one good configuration. Note that our results are only from a sample of configurations, so we cannot rule out all variability warnings in all configurations. Additionally we compare against the minimum and the default configuration (if provided by the codebase) as a baseline for the number of warnings in our dataset.

4.1 RQ1: What Variability Warnings Can Off-the-Shelf Bug Detectors Find?

We use a large, representative configuration sample with our approach, allowing us to examine how often the same warning appears across different configurations. By using the deduplication approach described in Section 3.3, we can examine the subset of configurations in which a warning occurs to discover how it is distributed across the configuration space. When a warning appears in all sampled configurations, we can assume it is unlikely to be a variability warning. Ruling out such warnings, our results still show a large number of variability warnings, i.e., those that only appear in a subset of the sampled configurations.

Table 3 shows the number of warnings produced by each static analysis tool for each target codebase. Each row shows the number



(a) CBMC (b) Clang (c) Infer (d) Cppcheck

Figure 4: Number of variability and generic warnings, over all programs, categorized by warning type.

Table 3: Summary of warnings found for each combination of target codebase and static analysis tool. *No warnings were found for Cppcheck. †Infer did not run on BusyBox.

	CBMC	**Clang**	**Infer**	**Cppcheck**
axTLS				
Total warnings	2,010	13	46	0
Variability (count)	668	9	20	0
Variability (percent)	33%	69%	43%	*
Toybox				
Total	4,292	56	112	20
Variability (count)	3,498	49	78	19
Variability (percent)	82%	88%	70%	95%
BusyBox				
Total	29,188	324	†	125
Variability (count)	23,896	276	†	118
Variability (percent)	82%	85%	†	94%

of total warnings and the number and percentage of the total that are variability warnings. Each column shows these numbers for each of the four static analysis tools tested. Note that Infer did not run on BusyBox because it threw an Assertion failure from its Clang backend.

Overall, the data show that variability warnings are not uncommon. In most combinations of target system and analysis tool, the majority of warnings found are variability warnings. When looking at Toybox and BusyBox, the proportion of variability warnings is high and fairly consistent across all four static analysis tools, ranging from 70% to 95%. axTLS had a lower number of variability warnings. Indeed, Cppcheck reported no warnings at all in our axTLS sample. In spite of this, the Clang static analyzer appears to find a higher percentage of variability warnings. Because the number of warnings is so small in this case, it is difficult to come to any conclusions for axTLS.

While not necessarily conclusive for all configurable systems, this provides evidence that more configurability could be correlated to more potential defects made harder to find due to configurability. We observed that most C files in Toybox and BusyBox require turning on at least one configuration option to be included in the compilation. Therefore, warnings from these files will be classified as being due to variability as long as there exist some configurations in our sample that do not compile them.

We conclude from the data that variability warnings are very frequent, and that *existing static analyses would reveal more potential defects if made variability-aware.*

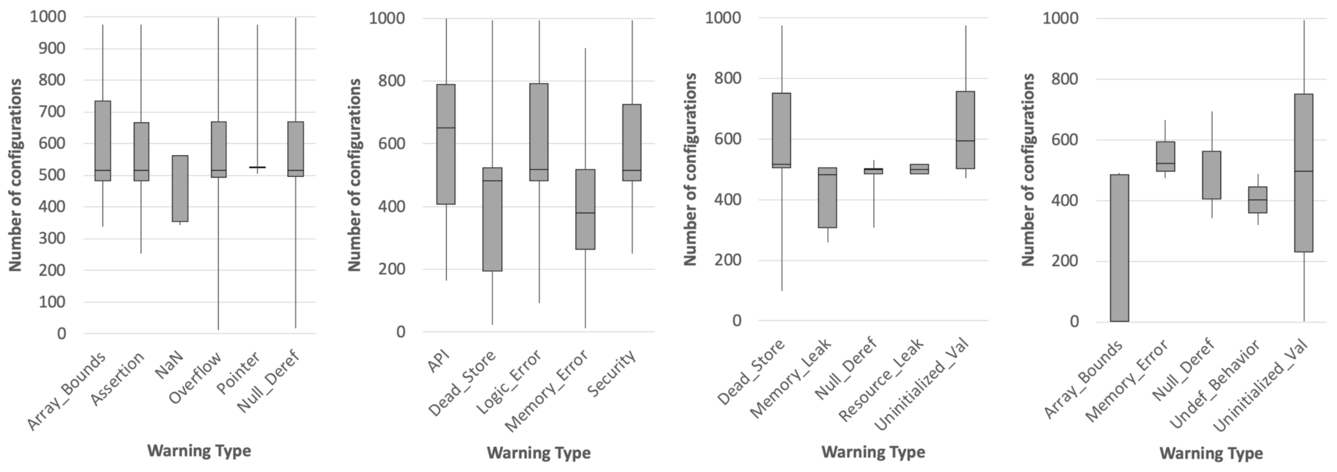
To further understand the above results, we categorize the warnings by their types. Figure 4 shows the distribution of variability and generic warnings under each type. Each bar in Figures 4a to 4d aggregates the warnings from all three programs of a certain type.

Overall, variability warnings span all types of warnings, except for the single Unix_API generic warning reported by Clang. Certain types dominate the total number of warnings for each tool. 94% of the CBMC warnings are Null_Deref or Overflow; 80% of the Clang warnings are Logic_Error or Dead_Store; Uninitialized_Val counts for 86% of Cppcheck warnings; 91% of the Infer warnings are Uninitialized_Val, Null_Deref, or Dead_Store. Nevertheless, the percentage of variability over all warnings of each warning type is largely consistent with the overall tool performance. For example, 85% of Clang warnings are due to variability. When categorized by types, the percentage of variability warnings (except the Unix_API warning) ranges from 72% (for Memory_Error) to 90% (for Logic_Error). The only type that has more generic warnings than variability warnings is Null_Deref by Infer, i.e., 23 generic warnings vs. 13 variability warnings.

In summary, these data show that *variability warnings represent all types of warnings found by these static analysis tools, including the potentially dangerous Null_Deref and Overflow.*

4.2 RQ2: How Are Variability Warnings Distributed Over the Space of Sample Configurations?

RQ1 provides evidence that variability warnings are very frequent, and we would like to evaluate how these warnings are distributed



(a) CBMC - [axtls&toybox] (b) Clang (c) Infer (d) Cppcheck

Figure 5: Number of configurations reporting variability warnings, over all programs, categorized by warning type.

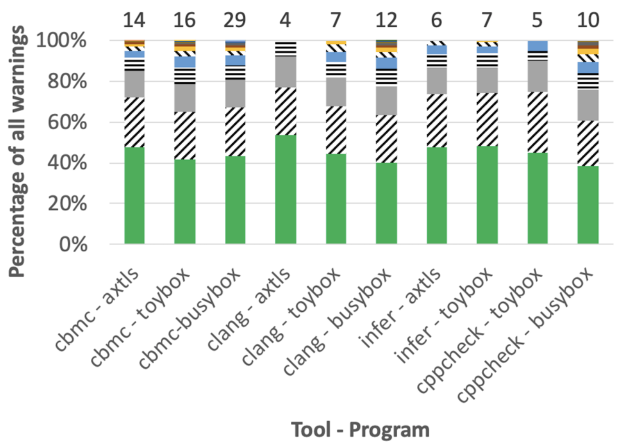


Figure 6: The subset of configurations that find all warnings.

across the sample set. For each warning, we count the number of sampled configurations in which it appears. This quantity gives us an idea about how "rare" a warning is, i.e., how likely we are to find this warning when checking a single configuration. Examining the distribution of configuration counts across all warnings gives us insight about how variability warnings are distributed. For these experiments, we only consider variability warnings, since generic warnings are assumed to appear in all configurations and RQ1 already shows how common such warnings are.

Figure 5 shows box-and-whisker plots of the distribution of variability warnings across the sampled configurations. Each chart represents the distributions for one static analysis tool by type of warning. Each box-and-whisker is an aggregate of the variability warnings from all three programs, except for CBMC in Figure 5a, which summarizes axTLS and Toybox results due to the failure to analyze every sample configuration of BusyBox. In Figure 5, the median number of configurations of most warning types is between 400 and 600. The three outliers are NaN from CBMC (355), Memory_Error from Clang (379), and API from Clang (651). In order to determine whether there were actual statically significant differences between the different warning types or if the different types were independent, we performed ANOVA tests on each tool's output. In all cases except Cppcheck, ANOVA rejects the null hypothesis of independence at $\alpha = 0.01$.

We additionally observe wide distributions of the number of configurations in most warning types. This means that some warnings were detected in either a small or a large subset of sampled configurations. For example, Infer discovered 9 Toybox warnings in 976 configurations. We found that these variability warnings exist in the same Toybox file (i.e., `ps.c`). This file is included in compilation if any one of five configuration options is turned on; specifically, these options are `PS`, `PKILL`, `PGREP`, `IOTOP`, and `TOP`. This disjunctive constraint is satisfied in most configurations in our sample, resulting in the detection of these variability warnings.

On the other hand, the Toybox warning that was detected by Cppcheck with least number of configurations (i.e., 98) appears in the file `axhttpd.c`. Upon checking the feature interaction associated with this warning, we found that it is a conjunction of 5 configuration options, which explains why it can only be detected in a few configurations. In another example, one Toybox warning was only discovered by Cppcheck in 4 configurations. This warning is emitted by the tool only if the configuration option `LSM_NONE` is turned on. In our sample, only 4 configurations turned on this option because Toybox's build system only allows this option to be turned on when many other options are turned off. The above observations illustrate that the detection of variability warnings from a sample is affected by the build system implementation of the target program, as well as the generation of samples. Overall, Figure 5 has shown that *while the majority of variability warnings are detected in many configurations in our sample, some can only be detected in a few due to the configurability of the target software.*

Figure 6 shows the subset of configurations in our sample that may produce the same results, for each tool and program combination. The number above each column shows the size of the subset, and the different colors/patterns in each column indicate different configurations. We computed this subset by greedily searching for the configuration that adds the most warnings. Algorithm 1 describes this process in detail.

```
Data: $C$ is the set of configurations, $W$ is the set of warnings, $S$ is a
      subset of $C$
Result: $S$
$C' := C, W' := W, S := \varnothing$;
Let $W'_c$ denote the set of warnings $w \in W'$ that are in configuration
  $c$;
while $W'$ is not empty do
    find $c \in C'$ that produces the most $w \in W'$;
    $S := S \cup \{c\}; W' := W' - W'_c; C' := C' - c$;
end
```

Algorithm 1: Find a subset of configurations with all warnings.

In Figure 6, at most 29 configurations (for CBMC-BusyBox) were needed to cover all warnings that were found by our samples. It only required 4 configurations to detect all Toybox warnings by Clang. On average, the subsets cover 92% and 80% of values of a single option and of the 2-way combinations of options in the configuration sample, respectively. This result suggests that *running static bug detectors on a small, well-constructed set of configurations may reveal many variability warnings.* We regard this as an important future research direction.

For developers deciding between variability-aware or variability-oblivious analyses, these results indicate a tradeoff. For situations where software reliability is less of a priority than faster bug detection, using a variability-oblivious analysis may be sufficient, albeit, finding the right set of configurations to test may be difficult. On the other hand, for critical software, finding every potential defect requires using variability-aware analyses to guarantee the safety of every configuration.

4.3 RQ3: How Do Our Results Compare to Checking a Maximum or Minimum Configuration?

To further understand how the variability warnings can be detected and affected by the choice of configurations we compare the warnings of sample configurations to that of minimum, default, and maximum configurations.

Table 4 shows the results of our comparison to maximum, minimum, and default configurations. Toybox and BusyBox ship with a default, but not axTLS. Turning on more configuration options typically means more code, therefore a configuration with more options should then result in more warnings. While the `allyesconfig` command for these codebases' build systems will generate a maximum configuration, due to the system dependencies, none of the `allyesconfigs` yield buildable configurations on which to run bug finders successfully. Instead, we generated a "maximum" configuration for each target program using the optimization algorithm presented by Oh et al. [31]. Similarly, to test the tools' capabilities in discovering warnings when most options are set to false, we generated a "minimum" configuration using the same optimization algorithm. We made the following observations from Table 4.

Table 4: Comparisons to warnings detected by maximum, default and minimum configurations. Each cell shows the number of configurations that are shared by the configuration and our results, only in the single configuration, or only in our sampled results, i.e., "shared · single · sample". *No warnings were found by Cppcheck on axTLS. †Infer did not run on BusyBox.

	cbmc	clang	infer	cppcheck
axTLS				
Maximum	1.9k · 0 · 82	10 · 0 · 3	42 · 0 · 4	*
Minimum	1.6k · 382 · 434	5 · 1 · 8	25 · 8 · 19	*
Toybox				
Maximum	4.2k · 0 · 29	48 · 0 · 8	110 · 0 · 2	14 · 0 · 6
Minimum	757 · 0 · 3.5k	10 · 0 · 46	31 · 0 · 78	4 · 2 · 16
Default	4.2k · 0 · 69	52 · 0 · 4	110 · 0 · 2	16 · 0 · 4
BusyBox				
Maximum	26k · 128 · 2.7k	256 · 3 · 68	†	82 · 9 · 43
Minimum	3.1k · 3 · 26k	21 · 2 · 303	†	4 · 5 · 117
Default	3.1k · 3 · 26k	249 · 2 · 75	†	83 · 11 · 42

First, *maximum configurations often produced a large portion of warnings discovered by running the tool on our sample, but never discovered all warnings.* More than 90% of CBMC and Infer warnings were found by maximum configurations. In other cases, maximum configurations discovered 66% (Cppcheck-BusyBox) to 86% (Clang-Toybox) of all warnings produced by our sample.

Second, *minimum configurations discovered fewer warnings, but also discovered some new warnings.* It is expected that with most configuration options turned off, minimum configurations would miss most warnings found by our study samples. For example, only 21 out of 324 and 4 out of 121 BusyBox warnings were found by Clang and Cppcheck running on the minimum configuration, respectively. Note, however, that minimum configurations also discovered several warnings that were never found in our samples, for example, 382 new axTLS warnings when using CBMC. Upon inspection, we believe one cause is that a static bug detector may consider certain code feasible only when most/all configuration options are set to false. Another reason is that some options were never enabled in our configuration samples, due to the constraints we enforce.

Third, *many warnings may be missed if tools are only run on default configurations.* Almost 90% of BusyBox warnings from CBMC would have been missed if only the default configuration was used. Similarly, about half of the axTLS warnings from Clang and Infer would have been missed.

The above results suggest that *it is not sufficient to only test and analyze the default (or another single) configuration of the target software, while it may be worthwhile to include the special configuration (e.g., minimum) as part of the test.*

5 BUG DATASET

Using the methodology described in Section 3.5, we manually classified all Clang, Infer, and Cppcheck warnings except for Clang's warnings on BusyBox and Uninitialized_Val warnings emitted by Cppcheck on BusyBox. All warning types except Dead_Store[7] were

[7] While a Dead_Store can be a symptom of another bug, the confirmation of Dead_Store warnings often results in optimization instead of bug fixes in the code.

Table 5: Manually inspected true positive bugs from our study of variability warnings. *No warnings were found for Cppcheck. †Infer did not run on BusyBox. ‡We did not investigate Clang BusyBox warnings.

	clang	infer	cppcheck
axTLS			
Variability bug	6	11	*
Generic bug	1	18	*
Toybox			
Variability bug	16	6	9
Generic bug	1	5	0
BusyBox			
Variability bug	†	‡	2
Generic bug	†	‡	2

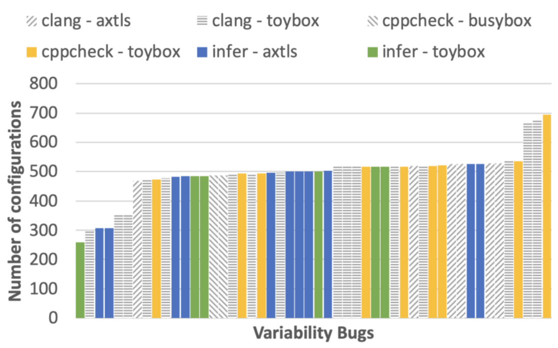


Figure 7: The distribution of number of configurations reporting variability bugs.

inspected. Because we confirm each true bug and its associated feature interactions as a team to gain more confidence on the results, manual classification of all warnings were prohibitive for the thousands of CBMC results, the 324 Clang on BusyBox warnings, and the 117 Uninitialized_Val warnings raised by Cppcheck on BusyBox. Nevertheless, we have generated a true bug dataset consisting of 77 previously-unpublished bugs from recent versions of axTLS, Toybox, and BusyBox (Table 5). Among them, 52 are variability bugs. This bug dataset also covers a wide range of bug types: Null_Deref (32), Logic_Error (14), Uninitialized_Val (7), Security (6), Memory_Error (6), Resource_Leak (5), Memory_Leak (4), Array_Bounds (2), and Undef_Behavior (1). We now study the bug characteristics to provide more insights on this dataset.

Figure 7 shows the distribution of the number of configurations on which variability bugs were discovered. The majority of variability bugs were detected by about half of the configurations in our samples, consistent with our observation from Figure 5. Figure 8 shows the number of configuration options affecting the bugs. Recall that these options were initially automatically identified, then manually confirmed (Section 3.6). In our dataset, 40 variability bugs are associated with only 1 configuration option. Among these bugs, 30 are defined in Makefiles, and 10 are defined with C preprocessor directives in the source code. Additionally, we see 9 bugs associated with some kind of dependency. Figure 9 shows an example of a configuration option, AXHTTPD, in a Makefile, which governs the inclusion of a file, axhttpd.c, in which we found four bugs. Figure 10 shows another configuration option, HTTP_HAS_AUTHORIZATION, which depends on AXHTTPD (i.e., HTTP_HAS_AUTHORIZATION can only be enabled if AXHTTPD is enabled). HTTP_HAS_AUTHORIZATION includes the file htpasswd.c, in which we found two bugs.

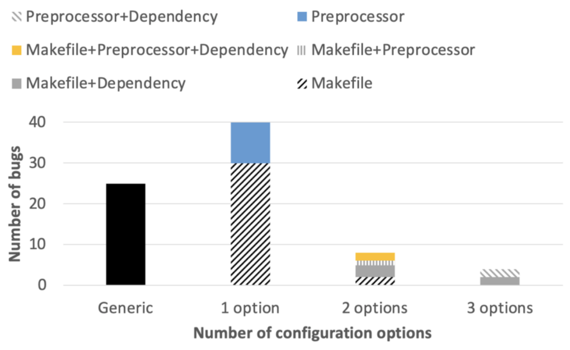


Figure 8: Feature interactions of variability bugs.

```
ifndef CONFIG_AXHTTPD
    web_server:
else
    web_server :: $(TARGET)
```

Figure 9: An example of variability in an axTLS Makefile.

```
menu "Axhttpd Configuration"
depends on CONFIG_AXHTTPD
...
config CONFIG_HTTP_HAS_AUTHORIZATION
    bool "Enable authorization"
    default y
...
endmenu
```

Figure 10: An example of dependencies between configuration options in an axTLS Kconfig file. All options defined in this menu depend on CONFIG_AXHTTP.

The other 12 bugs are associated with 2 or 3 options. Figure 8 shows that regardless of the number of options, those options can come from either preprocessor directives or from Makefiles. In bugs that are caused by the interaction of 2 or 3 options, those options can originate from any combination of Makefile, preprocessor, and dependency on another option. This result suggests that it is important to consider the build system constraints, Makefile, and preprocessors accurately to detect some variability bugs.

We believe this bug dataset can serve as a valuable benchmark for future studies, both in testing variability-aware analyses and in testing methods of finding variability bugs with variability-oblivious analysis. We also believe that although we could only manually classify a small percent of the emitted warnings, this dataset supports our methodology as being useful for finding variability bugs.

6 THREATS TO VALIDITY

We identify two threats to the validity of our conclusions. First, our empirical study used four static bug detectors and three open source C programs. Although these are real-world tools and programs, they may not be representative of bug detectors and configurable C software as a whole. Second, there are approximations in our approach that could lead to inaccurate results. Our deduplication process may identify two warnings as equivalent even if they are not, so long as they are on the same line and in the same source code file. Our estimation of variability may incorrectly label some variability warnings as generic warnings. Our manual classification approach may produce incorrect results. We increase the confidence that the true bugs we report are true by confirming these as a group.

7 RELATED WORK

Our work is related to (i) recent studies on variability bugs and dataset generation, (ii) research in finding variability bugs, and (iii) methods that identify feature interactions and dependencies.

Variability Bug Study and Dataset. Abal et al. [1, 40] studied previously reported (and fixed) variability bugs from the bug repositories of Apache, Busybox, Marlin, and Linux software. They created a database of 98 variability bugs, each of which has a feature interaction and simplified code version. These bugs are of similar types to the ones that we find. Bugs in this dataset span over a decade of software development and ones that exist in old software versions are challenging to be reproduced. Our work was inspired by their effort to create a variability bug dataset. Instead, we focus on previously-unpublished bugs in recent versions of the target programs that can be found by static analysis tools to support the evaluation of future analysis.

Other empirical work has been performed to study variability bugs [2, 29, 41–43]. For example, Medeiros et al. [41] investigated C programs to find undeclared/unused variables and functions arising from variability through global analysis using TypeChef [4]. They found how those issues were introduced from revision histories, and observed that those issues often took a long time to be fixed. Proper sampling strategy is beneficial for early detection of those issues. This work created a dataset consisting of 39 variability bugs.

Variability bug detection. Two important lines of research in finding variability bugs are Combinatorial Interaction Testing (CIT) [16, 44–46], and variability-aware static analysis. CIT aims to systematically sample configurations that satisfy certain coverage conditions [17–20]. Many adaptations of CIT have been proposed to address the needs of different configuration spaces, e.g., accounting for configuration constraints [47–49], test case constraints [23, 24], and cost aware-CIT [22]. Although CIT approaches have been shown to be effective in finding variability bugs, CIT test suites are not adequate to isolate and identify the specific interactions of configuration options that lead to the detected variability bugs. Using uniform random sampling, our approach focuses on studying warnings and bugs to make observations that are representative of the complete configuration space. Also, we integrate static analysis tools to detect variability bugs.

Over the last decade, researchers have developed new variability-aware static analysis techniques [7, 50–56]. Rhein et al. [7] presented a framework that implemented multiple variability-aware bug detectors. The results of these analyses were compared with three sampling-based approaches, demonstrating the effectiveness and efficiency of variability-aware analyses. Characteristics of the warnings detected were also studied. Our study complements this work to understand the types of variability warnings reported by off-the-shelf static bug detectors. In addition, we have generated a bug dataset via manual classification, potentially useful for evaluating this and other variability-aware analyses.

Understanding Feature Models of Existing Systems. Understanding the features and their dependencies from existing systems is vital for understanding the variability. Researchers developed methods and tools to identify feature interactions and dependencies statically from source code [57–59], make systems and configuration specifications [35, 60–62], or dynamically running the systems [63–66]. For example, Nguyen et al. [64] presented iGen that dynamically discovers feature interactions with counter example guided refinement. Our framework presents an algorithm for automatically identifying features, and we also study the feature interactions of the dataset. Several works in this line of research (e.g., iGen) can potentially be integrated into the framework we built as an alternative approach to identifying feature interactions.

8 CONCLUSIONS & FUTURE WORK

Variability bugs in configurable C software present a serious challenge for automatic bug detection through static analysis. State-of-the-art variability-aware static analyses are promising, but there is currently a gap between them and variability-oblivious tools. Our work shows how to simulate variability-awareness with these tools through our framework. We applied this framework to several state-of-the-art, but variability-oblivious, bug detectors and several highly-configurable codebases. With our results, we gain a deeper understanding of how these tools would perform, showing that variability warnings are very frequent, represent many kinds of potential defects, and are distributed across the configuration space. To support future research on highly-configurable code and analyses, our bug dataset provides 77 true positives bugs that we know can be found with static analysis tools. We hope this dataset will be a valuable benchmark for future tool developers.

We believe that the findings presented in Sections 4 and 5 will help drive future research into variability-aware analysis. One of our most interesting findings was that given any set of warnings obtained from 1,000 configurations, that same set of warnings could be obtained from a small subset of those configurations. Finding an algorithm to identify these configurations in advance would improve variability-aware testing. Future analyses of the small configuration sets presented in Section 4 may lead to insights on how to carefully construct configuration sets that exhibit a wide variety of variability warnings.

We have plans to continue to develop our toolchain and dataset to gather further data on variability bugs. We intend to make our framework pluggable to new bug detectors. We plan to continue to improve our framework with better feature identification and deduplication algorithms. Specifically, we would like to deduplicate bugs across tools, allowing us to compare different tools. We also plan to perform more classification of warnings to expand the bug dataset. Leveraging testing tools like program slicers and tracers can both speed up classification and enable classification of more complex warnings.

ACKNOWLEDGMENTS

This work is supported by NSF grants CCF-1840934 and CCF-1816951.

REFERENCES

[1] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: A qualitative analysis," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 421–432. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642990

[2] G. Ferreira, M. Malik, C. Kästner, J. Pfeffer, and S. Apel, "Do #ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel," in *Proceedings of the 20th International Systems and Software Product Line Conference*, ser. SPLC '16. New York, NY, USA: ACM, 2016, pp. 65–73. [Online]. Available: http://doi.acm.org/10.1145/2934466.2934467

[3] J. Melo, F. B. Narcizo, D. W. Hansen, C. Brabrand, and A. Wasowski, "Variability through the eyes of the programmer," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 34–44. [Online]. Available: https://doi.org/10.1109/ICPC.2017.34

[4] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 805–824.

[5] P. Gazzillo and R. Grimm, "Superc: Parsing all of c by taming the preprocessor," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 323–334. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254103

[6] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Static analysis of variability in system software: The 90, 000# ifdefs issue." in *USENIX Annual Technical Conference*, 2014, pp. 421–432.

[7] A. von Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, "Variability-aware static analysis at scale: An empirical study," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 4, p. Article No. 18, 2018.

[8] A. Garrido and R. Johnson, "Analyzing multiple configurations of a C program," in *ICSM*, Sep. 2005, pp. 379–388.

[9] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini, "SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years," in *PLDI*. ACM, 2013.

[10] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 81–91.

[11] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden, "Variational Data Structures: Exploring Tradeoffs in Computing with Variability," in *Onward!* ACM, 2014, pp. 213–226.

[12] C. Kästner, K. Ostermann, and S. Erdweg, "A Variability-aware Module System," in *OOPSLA*. ACM, 2012, pp. 773–792.

[13] A. F. Iosif-Lazar, J. Melo, A. S. Dimovski, C. Brabrand, and A. Wasowski, "Effective Analysis of C Programs by Rewriting Variability," *CoRR*, 2017.

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

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

[16] C. Yilmaz, S. Fouché, M. B. Cohen, A. Porter, G. Demiroz, and U. Koc, "Moving Forward with Combinatorial Interaction Testing," *Computer*, vol. 47, no. 2, pp. 37–45, Feb. 2014.

[17] K.-C. Tai and Y. Lei, "A test generation strategy for pairwise testing," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 109–111, Jan. 2002.

[18] S. Oster, F. Markert, and P. Ritter, "Automated incremental pairwise testing of software product lines," in *International Conference on Software Product Lines*. Springer, 2010, pp. 196–210.

[19] M. F. Johansen, Ø. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM, 2012, pp. 46–55.

[20] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu, "Practical pairwise testing for software product lines," in *Proceedings of the 17th international software product line conference*. ACM, 2013, pp. 227–235.

[21] C. Nie and H. Leung, "A Survey of Combinatorial Testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29, Feb. 2011. [Online]. Available: http://doi.acm.org/10.1145/1883612.1883618

[22] G. Demiroz, "Cost-aware Combinatorial Interaction Testing (Doctoral Symposium)," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 440–443, event-place: Baltimore, MD, USA. [Online]. Available: http://doi.acm.org/10.1145/2771783.2784775

[23] C. Yilmaz, "Test Case-Aware Combinatorial Interaction Testing," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 684–706, May 2013.

[24] U. Koc and C. Yilmaz, "Approaches for computing test-case-aware covering arrays," *Software Testing, Verification and Reliability*, vol. 28, no. 7, p. e1689, 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1689

[25] C. Yilmaz, E. Dumlu, M. B. Cohen, and A. Porter, "Reducing Masking Effects in CombinatorialInteraction Testing: A Feedback DrivenAdaptive Approach," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 43–66, Jan. 2014.

[26] C. Song, A. Porter, and J. S. Foster, "iTree: Efficiently Discovering High-coverage Configurations Using Interaction Trees," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 903–913. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337329

[27] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero, "Configuration coverage in the analysis of large-scale system software," in *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*. ACM, 2011, p. 2.

[28] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon, "Combining multi-objective search and constraint solving for configuring large software product lines," in *ICSE*, 2015.

[29] J. Melo, E. Flesborg, C. Brabrand, and A. Wasowski, "A quantitative analysis of variability warnings in linux," in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, ser. VaMoS '16. New York, NY, USA: ACM, 2016, pp. 3–8. [Online]. Available: http://doi.acm.org/10.1145/2866614.2866615

[30] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 643–654.

[31] J. Oh, P. Gazzillo, D. Batory, M. Heule, and M. Myers, "Uniform sampling from kconfig feature models," The University of Texas at Austin, Department of Computer Science, Tech. Rep. TR-19-02, 2019.

[32] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: A manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2644805

[33] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of C preprocessor use," *IEEE TSE*, pp. 1146–1170, Dec 2002.

[34] M. Thurley, "sharpsat–counting models with advanced component caching and implicit bcp," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2006, pp. 424–429.

[35] 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

[36] M. Endler, "Awesome static analysis!" 2018. [Online]. Available: https://github.com/mre/awesome-static-analysis

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

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

[39] "scan-build." [Online]. Available: https://clang-analyzer.llvm.org/scan-build.html

[40] 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

[41] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi, "An empirical study on configuration-related issues: Investigating undeclared and unused identifiers," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2015. New York, NY, USA: ACM, 2015, pp. 35–44. [Online]. Available: http://doi.acm.org/10.1145/2814204.2814206

[42] J. Melo, C. Brabrand, and A. Wasowski, "How does the degree of variability affect bug finding?" in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 679–690. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884831

[43] R. Muniz, L. Braz, R. Gheyi, W. Andrade, B. Fonseca, and M. Ribeiro, "A qualitative analysis of variability weaknesses in configurable systems with #ifdefs," in *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VAMOS 2018. New York, NY, USA: ACM, 2018, pp. 51–58. [Online]. Available: http://doi.acm.org/10.1145/3168365.3168382

[44] R. Brownlie, J. Prowse, and M. S. Phadke, "Robust testing of at&t pmx/starmail using oats," *AT&T Technical Journal*, vol. 71, no. 3, pp. 41–47, 1992.

[45] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 38–48. [Online]. Available: http://dl.acm.org/citation.cfm?id=776816.776822

[46] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, 1997.

[47] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.

[48] P. Danziger, E. Mendelsohn, L. Moura, and B. Stevens, "Covering arrays avoiding forbidden edges," *Theoretical Computer Science*, vol. 410, no. 52, pp. 5403–5414, 2009.

[49] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *2009 1st International Symposium on Search Based Software Engineering*. IEEE, 2009, pp. 13–22.

[50] K. Lauenroth, K. Pohl, and S. Toehning, "Model checking of domain artifacts in product line engineering," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 269–280.

[51] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer, "Type safety for feature-oriented product lines," *Automated Software Engineering*, vol. 17, no. 3, pp. 251–300, 2010.

[52] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1.* ACM, 2010, pp. 335–344.

[53] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1.* ACM, 2010, pp. 105–114.

[54] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake, "Analysis strategies for software product lines," *School of Computer Science, University of Magdeburg, Tech. Rep. FIN-004-2012*, 2012.

[55] C. Kästner, S. Apel, T. Thüm, and G. Saake, "Type checking annotation-based product lines," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 3, p. 14, 2012.

[56] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable Analysis of Variable Software," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 81–91, event-place: Saint Petersburg, Russia. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491437

[57] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *Proceedings of the 36th International Conference on Software Engineering.* ACM, 2014, pp. 140–151.

[58] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 820–841, Aug. 2015.

[59] M. Lillack, C. Kästner, and E. Bodden, "Tracking Load-Time Configuration Options," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1269–1291, Dec. 2018.

[60] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, "A robust approach for variability extraction from the linux build system," in *Proceedings of the 16th International Software Product Line Conference-Volume 1.* ACM, 2012, pp. 21–30.

[61] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A study of variability models and languages in the systems software domain," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1611–1640, 2013.

[62] S. Zhou, J. Al-Kofahi, T. N. Nguyen, C. Kästner, and S. Nadi, "Extracting Configuration Knowledge from Build Files with Symbolic Analysis," in *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, May 2015, pp. 20–23.

[63] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, and G. Saake, "On essential configuration complexity: Measuring interactions in highly-configurable systems," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970322

[64] 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

[65] L. S. Ghandehari, Y. Lei, R. Kacker, D. R. R. Kuhn, D. Kung, and T. Xie, "A Combinatorial Testing-Based Approach to Fault Localization," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[66] L. R. Soares, J. Meinicke, S. Nadi, C. Kästner, and E. S. de Almeida, "Exploring Feature Interactions Without Specifications: A Controlled Experiment," in *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2018. New York, NY, USA: ACM, 2018, pp. 40–52, event-place: Boston, MA, USA. [Online]. Available: http://doi.acm.org/10.1145/3278122.3278127