Variability-Aware Static Analysis at Scale: An Empirical Study

ALEXANDER VON RHEIN, CQSE GmbH, Germany
JÖRG LIEBIG, 4Soft GmbH, Germany
ANDREAS JANKER, Capgemini Deutschland GmbH, Germany
CHRISTIAN KÄSTNER, Carnegie Mellon University, USA
SVEN APEL, University of Passau, Germany

The advent of variability management and generator technology enables users to derive individual system variants from a configurable code base by selecting desired configuration options. This approach gives rise to the generation of possibly billions of variants, which, however, cannot be efficiently analyzed for bugs and other properties with classic analysis techniques. To address this issue, researchers and practitioners have developed sampling heuristics and, recently, variability-aware analysis techniques. While sampling reduces the analysis effort significantly, the information obtained is necessarily incomplete, and it is unknown whether state-of-the-art sampling techniques scale to billions of variants. Variability-aware analysis techniques process the configurable code base directly, exploiting similarities among individual variants with the goal of reducing analysis effort. However, while being promising, so far, variability-aware analysis techniques have been applied mostly only to small academic examples. To learn about the mutual strengths and weaknesses of variability-aware and sample-based static-analysis techniques, we compared the two by means of seven concrete control-flow and data-flow analyses, applied to five real-world subject systems: BUSYBOX, OPENSSL, SQLITE, the x86 LINUX kernel, and UCLIBC. In particular, we compare the efficiency (analysis execution time) of the static analyses and their effectiveness (potential bugs found). Overall, we found that variability-aware analysis outperforms most sample-based static-analysis techniques with respect to efficiency and effectiveness. For example, checking all variants of OPENSSL with a variability-aware static analysis is faster than checking even only two variants with an analysis that does not exploit similarities among variants.

ACM Reference Format:
Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. 1, 1 (October 2018), 34 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Compile-time configurability enables users to tailor a software system to particular application scenarios and hardware platforms. From the early days of C, developers exploit the facilities of the C preprocessor CPP to support such configurability. Optional and alternative code fragments are annotated with preprocessor directives (`#ifdef`s), which are included or excluded at compile time, depending on the values of configuration options that end users specify. Compile-time configurability provided by CPP is widely accepted and used in open-source projects and industry [9, 25, 39, 43]. Existing systems, such as the

Authors' addresses: Alexander von Rhein, CQSE GmbH, Germany; Jörg Liebig, 4Soft GmbH, Germany; Andreas Janker, Capgemini Deutschland GmbH, Germany; Christian Kästner, Carnegie Mellon University, USA; Sven Apel, University of Passau, Germany.

XXXX-XXXX/2018/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Linux kernel, provide up to several thousands of configuration options, giving rise to a huge configuration space, with up to 2^n system configurations for n configuration options, in the worst case. The problem is that certain properties of interest (e.g., the presence of code smells or bugs) may depend on the values of configuration options and their specific interactions in the system. This is not a theoretical problem, severe bugs in the past, such as Heartbleed[1] in OpenSSL, were related to configuration options [1, 42]. KBuild-robot, a testing infrastructure of the Linux kernel, currently finds over 700 build errors and warnings per month by building random configurations of the Linux kernel.[2]

There are many quality assurance strategies to uncover bugs (and other problems). Static analyses are interesting because they can reason about a program and all its inputs without actually executing it. However, current static analysis approaches struggle with compile-time variability: They only analyze individual configurations (e.g., C code after resolving compile-time variability by a preprocessor), not the entire configuration space. To handle large configuration spaces, developers typically sample a (small) subset of configurations (e.g., a set of configurations covering all pairs or triples of configuration options [26, 57] or even only one configuration with most configuration options enabled, such as the *allyes* configuration in Linux [55]) of a given configurable system and analyze them with the target analysis tool at hand. That is, even though standard static analysis tools analyze a program with regard to all inputs, they are typically executed only for a (small) sample of compile-time configurations. In general, configuration sampling does not cover the entire configuration space. Researchers claim that t-wise sampling is effective for finding issues in configurable systems, but the ground truth (i.e., whether sampling detects all bugs or how many, and whether it scales to large software systems) is unknown [48].

As an alternative to cover huge configuration spaces without relying on sampling, researchers have developed the notion of variability-aware analysis, which processes the configurable code base directly, exploiting similarities among individual variants to reduce analysis effort [2, 5, 15, 17, 35, 38, 57] (which is roughly analogous to how static analyses reason about all inputs). As an important property, variability-aware analysis techniques are sound and complete with regard to the system's configuration space, that is they report results for *all* system variants equivalent to what a brute force analysis of all configurations separately would report. At the same time, variability-aware analysis analyzes common and variable parts of the source code only once and is able to detect potential bugs in every valid system variant.

We implemented seven variability-aware intra-procedural static control-flow and data-flow analyses (*case termination*, *dangling switch*, *dead store*, *double free*, *error handling*, *function return*, and *freeing static memory*) and evaluated them on five real-world systems (the Linux kernel, the Busybox tool suite, the cryptographic library OpenSSL, the database-management system SQLite, and the C standard library uClibc). In particular, we compare the efficiency and effectiveness of running our analyses (1) in a variability-aware fashion and (2) in a sample-based fashion using three state-of-the-art sampling techniques (single configuration, code coverage, and pair-wise sampling). To evaluate efficiency, we measure and compare the analyses' execution times. To evaluate effectiveness, we compare the absolute number of reported warnings. We call the analysis reports "warnings" instead of "bugs", as any static analysis usually produces false positives and most of our analyses point to *sloppy* code, which is likely to be error-prone (e.g., unchecked returns from system functions). In this work, we do not formally investigate how many of the reported warnings are false positives. To gain deeper insights into the warnings reported by the different strategies, we quantify the complexity of configuration-related warnings (i.e., the difficulty of detecting the warnings). To this end, we measure the *interaction degree* (a.k.a., the order of feature interaction [4]) of each warning and compare the interaction degrees of warnings reported by the different

[1] http://heartbleed.com/

[2] Statistics by Jesper D. Brouer, https://lists.01.org/pipermail/kbuild-all/

analysis strategies. The interaction degree of a configuration-dependent warning denotes the minimal number of configuration options that must be explicitly enabled or disabled to trigger the warning in a configurable system, which denotes to how many system variants the warning applies.

Our experiments reveal that variability-aware analysis is often more efficient than sample-based analysis, while still covering all configurations instead of only a (typically small) sample. With respect to effectiveness, we found that warnings that involve many configuration options are reported more often by variability-aware analysis than sample-based analysis. Having many configuration options in a warning is an indicator that it affects fewer configurations (than a warning with a low number of options), which means that it is rather difficult to detect with sampling or by chance. In our evaluation, we found many warnings with high interaction degrees, which suggests that the incompleteness of sampling might be a serious issue in practice.

In summary, we make the following contributions:

- We have developed a variability-aware static analysis tool based on an extended data-flow analysis framework (Monotone Frameworks). The framework is a generalization of previous work on variability-aware liveness analysis [41].
- We introduce and describe variability-aware versions of seven static intra-procedural analyses for bug detection: *case termination*, *dangling switch*, *dead store*, *double free*, *error handling*, *function return*, and *freeing static memory*. We include a detailed discussion of the analyses in the Appendix A.
- Using our static analysis, we compare the efficiency and effectiveness of variability-aware analyses against three different sampling techniques (single configuration, code coverage, and pair-wise).
- We report and analyze a number of potential bugs in five highly configurable real-world systems (the tool suite Busybox, the x86 part of the Linux kernel, the cryptographic library OpenSSL, the embedded database-management system SQLite, and the C-standard library uClibc).

All raw data, measurement results, and further information are publicly available on a supplementary Web site[3].

This paper extends previous work [41] in which we compared variability-aware and sample-based analysis by the examples of type checking and liveness analysis. Here, we extend this work by applying our analyses on more subject systems (SQLite and uClibc). We confirm the results of variability-aware analysis outperforming most sample-based analysis. We generalize the former implemented liveness analysis to a general data-flow framework based on Monotone Frameworks and use it to evaluate different static analyses for detecting common programming errors. We investigate the interaction degrees of reported warnings identified by our static analyses; we found that warnings depend on up to 16 interacting configuration options (Section 4).

2 BASICS OF VARIABILITY-AWARE ANALYSIS

Before we introduce our new variability-aware control-flow and data-flow analyses, we introduce the basic concepts of variability-aware reasoning about compile-time variability. Specifically, we outline the development of configurable systems using the C preprocessor cpp (Section 2.1) and the representation of variable code in abstract syntax trees (Section 2.2), before we introduce the basic mechanics of variability-aware analysis using type checking as a simple example (Section 2.3). We conclude the introduction with an overview of generation design principles that we identified for efficient variability-aware analyses (Section 2.4).

[3]http://fosd.net/var_stat_analysis/

```
1 #ifdef A
2   #define EXPR (x<0)
3 #else
4   #define EXPR 0
5 #endif
6
7 int foo(int x #ifdef B , int y #endif) {
8   int r;
9   if (EXPR) {
10    r =
11 #ifdef B
12        -y;
13 #else
14        -x;
15 #endif
16  }
17  int z = x;
18  if (z) {
19    z += y;
20 #ifdef B
21    z += r;
22 #endif
23  }
24 #ifdef C
25  return z;
26 #endif
27 }
```

Fig. 1. Running example in C with variability expressed using preprocessor directives (e.g., Lines 1–5, 7, and 21); for brevity, we use `#ifdef` directives inside single code lines.

2.1 Preprocessor-Based Variability

CPP is a frequently applied tool for the development of configurable software [39]. It enables developers to implement a variable code base using conditional-inclusion macros (a.k.a. `#ifdef`s). For instance, our running example in Figure 1 contains five configurable pieces of code: an alternative macro expression (Lines 1–5), an optional function parameter (Line 7), and optional statements (e.g., Line 21). The inclusion or exclusion of such annotated code is controlled by the values of *configuration options* (here: A, B, and C), which can be combined using logical operators.

Often, not all combinations of configuration options are valid, so developers use *variability models* to express relations between configuration options and to define which of their combinations are valid. One widely used tool in practice to express variability models is KCONFIG,[4] which is used, for example, in the development of LINUX and BUSYBOX. Variability models can be transformed into propositional formulas, which enables efficient reasoning about them using current SAT-solver technology [45].

[4]https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

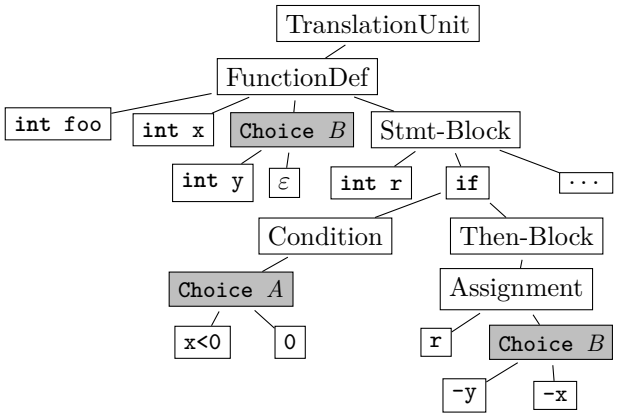


Fig. 2. Excerpt of the corresponding variational AST of our running example of Figure 1

Symbol	Type	Declaration	Presence Condition
`foo`	`(int,int)→int`	Line 7	B
`foo`	`int→int`	Line 7	$\neg B$
`x`	`int`	Line 7	`true`
`y`	`int`	Line 7	B
`r`	`int`	Line 8	`true`
`z`	`int`	Line 17	`true`

Fig. 3. Variational symbol table for the running example. Each line shows a symbol declaration. For optional declarations, we show the declaration's presence condition (or `true` for non-optional declarations). Line numbers refer to Figure 1.

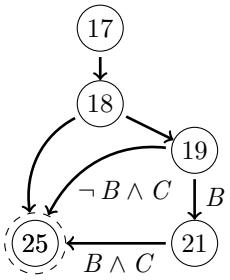


Fig. 4. Excerpt of the variational CFG of the running example in Figure 1. The dashed node represents the `return` statement.

2.2 Variational Abstract Syntax Trees

Many static analyses operate on the abstract syntax tree (AST). As we want to analyze the configurable system as a whole, we need an AST that covers all variants of a system and including information regarding configuration options. For this purpose, we base our analyses on the variability-aware parser TypeChef [31], which generates such a *variational AST*. A variational AST is an extension of a standard AST, with additional nodes for expressing compile-time variability. To create variational ASTs, the parser relies on variability-model extraction and build-system analysis [7, 8].

A `Choice` node in a variational AST denotes the choice between two or more alternative subtrees, reflecting the choice arising from `#ifdef`s in the source code (similar to ambiguity nodes in GLR parse forests [58]; explored formally in the choice calculus [22]). For example, `Choice(`A`,x<0,0)` in Figure 2 expresses the alternative of the two expressions `x<0` and `0` controlled by configuration option A, after macro expansion. We use `Choice` nodes to represent `#ifdef`-annotated code in our running example. In general, the conditional of a `Choice` node is a propositional formula involving one or more configuration options (e.g., $(A \vee B) \wedge C$). These propositional formulas are called *presence conditions* [18], because they denote the condition for whether a code fragment is present. One alternative of a `Choice` node may be

empty (denoted by ε; Figure 2), which makes the other, in fact, optional. In principle, we could use a single `Choice` node on top of the AST with one large branch per system variant; but a variational AST is more compact, because it shares parts that are common across multiple variants (e.g., in Figure 2, we store only a single node for the function name `foo`, which is shared by all variants). In Section 2.4, we discuss this kind of sharing as a design principle for efficient variability-aware analysis.

The construction of a variational AST from a real-world system, such as Linux, is not trivial. Whereas parsing preprocessed C code of an individual system variant is well established, parsing a configurable system with `#ifdef`s is challenging [31]. To make matters worse, conditional-compilation directives in the C preprocessor (`#ifdef`) may interact with the build system, with macros (`#define`), and with file-inclusion directives (`#include`) across file boundaries. Fortunately, such interactions are resolved by TypeChef [31], and, in the remainder of this paper, we use TypeChef's parser infrastructure as a black box and work on the resulting variational ASTs.

2.3 Variability-Aware Analysis

Variability-aware analysis (also known as family-based analysis [57]) takes advantage of the similarities between the variants of a configurable system to speed up the analysis process. Although individual variability-aware analysis techniques differ in many details [57], an idea that underlies all of them is to analyze code that is shared by multiple system variants only once. To this end, variability-aware analysis does not operate on individual system variants, but on the raw code artifacts that still contain variability and available configuration knowledge, prior to the variant-generation (or preprocessing) step.

Variability-aware analysis has been applied in several academic projects, showing promising performance improvements by orders of magnitude [2, 3, 5, 15–17, 31–33, 35, 38, 57]. Since many real-world software systems are implemented in C and use `#define` and `#ifdef` directives (and a build system) to implement compile-time variability, we have developed a standard set of static variability-aware analyses *for C* and applied them to a number of *large-scale* projects.

Type checking as a simple example. We describe variability-aware type-checking as a basic example for a variability-aware analysis, as it is rather simple and straightforward to explain. To check type correctness of a configurable program, the type checker scans the variational AST for symbol declarations (variables, structs, and functions) and statements where the symbols are used. Based on the symbol declarations the type checker checks whether all variables are declared in all program variants in which they are used and whether used and declared types are compatible. The variational symbol table in Figure 3 shows all declarations of variables in our running example (Figure 1 and 2) and their presence conditions. For example, variable `y` is declared in Line 7, if configuration option B is enabled. Next, the type analysis scans the code for uses of `y`. It is used in Line 10, if B is enabled, and in Line 19, in all variants. For each use, we check whether its presence condition (B for Line 10 and for `true` Line 19) implies the presence condition of the variable declaration (B in Line 7) and whether the types are compatible in all configurations (in our example, all types are `int`; hence they are trivially compatible). As there are program variants in which B is not enabled, we have discovered a type error in all program variants without B (variable `y` is used in Line 19, but not defined in Line 7). Type checking provides the necessary information (including presence conditions for uses and declarations of variables[5]) that are used in other static analyses, which we describe in Section 4.2.

Interaction faults. In the previous example, we have identified a type error that occurs in all program variants in which configuration option B is not enabled. In practice, it is also common that errors occur

[5]For example, `y` is declared with condition B and used with condition `true`.

only when multiple options are combined in specific combinations; these issues are called *interaction faults*. For a found issue (error or warning), a variability-aware analysis reports the issue together with a presence condition, such as B or $\neg A \wedge B$. In the latter case, two options need to be assigned to a specific value for the warning to appear. Typically, the more options are involved in an issue's presence condition, the fewer configurations contain the issue and, for better and worse, the smaller the chance to find the issue in a random configuration.

Sharing vs. abstraction. Variability-aware analysis reasons about huge configuration spaces by exploiting *sharing* among configurations and by performing analysis steps only once when they are shared among many or all configurations. As result, it is typically sound and complete with regard to a brute-force execution of an underlying traditional analysis. That is, variability-aware analysis will find the same issues and not more, than the underlying analysis would find if applied to each compile-time configuration separately. Variability-aware analysis does not introduce or use any additional abstractions with regard to the configuration space beyond those abstractions used by the underlying analysis. A variability-aware analysis does not introduce any decidability issues (beyond those of the underlying analysis), because configuration spaces are finite and reasoning about compile-time variability remains decidable (and is often efficient).

Note how variability-aware analysis adds an additional dimension of compile-time variability to the traditional challenges of quality assurance. To overcome decidability and scalability issues for infinite input domains, static analyses use *abstraction* to reason about a whether a specific compile-time configuration of a program fulfills a specification across all possible inputs. Variability-aware analysis aims at *sharing* to lift an underlying traditional analysis and apply that analysis to large but finite configuration spaces. In principle, it is possible to also use abstraction as part of variability-aware analysis to conservatively assure configurations—for example, during type checking, we could reject all programs that call a function that is not defined in all configurations, thus achieving a sound result, but also rejecting programs that are well-typed in all configurations. The class of variability-aware analyses discussed in this article does use abstraction only as part of the underlying analysis and not with regard to the configuration space and can, therefore, pinpoint problems to exact configurations. Similarly, we discuss sampling as an alternative strategy only with regard to the configuration space, not as part of an underlying analysis for individual configurations.

2.4 Design Principles of Variability-Aware Analysis

We and others have implemented different kinds of variability-aware analyses in recent years [57]. Although the specifics differ based on the problem addressed (e.g., parsing, type checking, model checking), we have observed general design principles, which explain why these analyses can efficiently analyze even real-world systems with myriads of possible variants. The key is *keeping variability local*. Parsing in TypeChef already maximizes sharing in the AST and keeps variability local: First, code without `#ifdef` directives is represented only once, since it is common to all variants. Second, if choices are necessary, they are introduced only locally where code differs between variants (see Section 2). We preserve this sharing and locality throughout our variability-aware analyses, as far as possible. Specifically, we have identified three patterns that maximize sharing: *late splitting*, *local variability representation*, and *early joining*.

First, *late splitting* means that we perform the analysis without variability until we encounter it. For example, type checking processes the declaration of symbol `y` in Line 7 of the running example (Figure 1) only once, and adds it to the symbol table only once, whereas a brute-force technique or a sampling technique would process this declaration multiple times (once per analyzed system variant). Also, when

we use symbol `y` later, it has only one type. Variability-aware analysis only splits and considers smaller parts of the configuration space when it actually encounters variability, for example, in the declaration of parameter `y`. Late splitting is similar to path splitting in on-the-fly model checking, where splitting is also performed only on demand [14].

Second, *local variability representation* aims at storing variability local in intermediate results. For example, instead of copying the entire symbol table for a single variational entry, we have only a single variational symbol table with conditional entries. In our running example, we only store a single type for `y`, independently of B, even after the conditional declaration of parameter `y`. Technically, we use `y:Choice(B, int, ϵ)` instead of `Choice(B, y:int, ϵ)` to achieve this locality (`Map[ID,Choice[Type]]`[6] instead of `Choice[Map[ID,Type]]`). [7]

Third, *early joining* attempts to *join* intermediate results as early as possible. For example, if we have a choice of two identical types `Choice(`A`,int,int)`, we can simply join them to `int` for further processing. So, even if we need to compute the Cartesian product on some operations with two variable types, the result can often be joined again to a more compact representation. For example, we get a compact representation if the parameters of a function are variational and its return type is not (`#ifdef` X `int foo(int) #else int foo() #endif` can be represented as `foo:Choice(X, int, ϵ)→int`, corresponding to `int foo(#ifdef` X `int #else /*empty*/ #endif`)). This way, variability from parts of the AST leaks into other parts, if and only if variability *actually makes a difference* in the internal representations of types, names, or other structures. Also, we need to consider only combinations of configuration options that occur in different parts of the AST if they actually produce different (intermediate) results when combined, otherwise the results remain orthogonal.

Note that the three patterns of late splitting, local variability representation, and early joining apply to any kind of variability-aware analysis; although not always made explicit, these patterns can also be observed in other variability-aware analyses [5, 11, 31, 33].

3 VARIABILITY-AWARE CONTROL-FLOW AND DATA-FLOW ANALYSIS

To evaluate the efficiency and effectiveness of variability-aware static analysis (Section 4), we implemented seven commonly used intra-procedural analyses as variability-aware analyses: three control-flow analyses (*function return*, *case termination*, and *dangling switch*), and four data-flow analyses (*deadstore*, *error handling*, *double free*, and *freeing of static memory*).[8] These analysis are commonly used in compilers to issue warnings and many of them are codified as rules or recommendations for secure programming in the CERT C Coding standard [51]. The control-flow analyses are based on variational control-flow graphs (CFGs), which we explain in Section 3.1, using a dangling-switch analysis as an example. The data-flow analyses [34] are also variability-aware and are based on an extension of Monotone Frameworks [49], which we developed for this purpose and which makes use of variational CFGs for efficient data-flow computation.

3.1 Variational Control-flow Graphs

For most static analyses, we need to construct a CFG, which represents all possible execution paths of a program. Nodes of the CFG correspond to instructions in the AST, such as assignments and function

[6] We use square brackets to indicate type parameters, as in the Scala programming language.
[7] For a more complex example consider the function `foo` from Figure 1: We represent the type of `foo` as `foo:Choice(B, (int,int)→int, int→int)` instead of `Choice(B, foo:(int,int)→int, foo:int→int)`.
[8] We provide a more detailed discussion of the analyses in Appendix A.

calls; edges correspond to possible successor instructions, according to the execution semantics of the programming language.

To cover all variants of a configurable program, we make its CFG variational. To create a CFG for a program without compile-time variability, we compute the successors of each node.[9] In the presence of variability, the successors of a node may differ, so we implement a variability-aware successor function that takes variability during the computation of successor nodes into account (expressed with the Choice type, as discussed by Walkingshaw et al. [22, 61]). The result of this successor function contains for every possible successor, a corresponding presence condition, which we store as an edge annotation in the variational CFG. In our running example (Figure 1), the successor of the statement in Line 19 is one of the statements in Line 21 or Line 25: `succ` (19) = `Choice`(B, 21, 25).[10]

Let us illustrate variational CFGs by means of the statement in Line 19 of our running example of Figure 1. In Figure 4, we show an excerpt of the corresponding variational CFG (node numbers refer to line numbers of Figure 1). The successor of the instruction `z += y` in Line 19 is configuration-dependent: if B is selected, statement `z += r` in Line 21 is the direct successor; if B is not selected, then `return z` in Line 25 is the (only) successor. Note that, by evaluating the presence conditions on edges, we can reproduce the CFG of each system variant.[11]

Next, we illustrate variability-aware control-flow analysis by the example of a *function return* analysis applied to our running example. This analysis inspects the control flow in non-void functions to detect undefined program behavior: According to the C standard, the control flow of all functions with a non-void return type should execute a `return` statement. Using the return value of a function that does not execute a `return` leads to undefined behavior. Our running example (Figure 1) shows an instance of this warning: the `return` statement in Line 25 is only executed if option C is enabled. Using the CFG of the running example (Figure 4), we determine whether all control-flow paths of the function's CFG end in a `return` statement. To this end, our analysis determines all CFG statements that can possibly be executed at last in a function's context. For any CFG statement that is not a `return` statement, our analysis issues a warning. In our example, only paths that have the option C enabled end in a `return` statement, and, therefore, our analysis issues a warning when C is disabled. For our experiments, we implemented two additional intra-procedural control-flow analyses, which we describe in Section 4.2.

3.2 Variability-aware Monotone Frameworks

Many data-flow analyses can be implemented on top of a general intra-procedural framework for solving data-flow equations, called Monotone Frameworks [49]. A Monotone Frameworks analysis traverses a system's CFG (using a flow function such as `succ`) and tracks data-flow properties of interest until a fixpoint is reached. To make the framework variability-aware, we modified the framework and applied the three principles of variability-aware analysis (see Section 2.4): late splitting, local variability representation, and early joining.

[9] The signature of the non-variational successor function is `succ: Node→List[Node]`.

[10] The signature of the variational successor function is `succ: Node→List[Choice[Node]]`.

[11] Alternatively, we could have dropped the presence conditions on edges and express variations of the control flow with `if` statements [60]. On an `if` statement, a normal CFG does not evaluate the expression, but conservatively approximates the control flow by reporting both alternative branches as possible successor statements (e.g., in Figure 4, both nodes 19 and 25 may follow node 18). Such sound but incomplete approximation is standard practice to make static analysis tractable or decidable. However, we do not want to lose precision when reasoning about static variability. Furthermore, we have only propositional formulas to decide between execution branches, which makes computations decidable and comparably cheap, so we decided in favor of presence conditions on edges, which is in line with prior work on configurable Java programs [10, 11].

MONOTONE FRAMEWORKS is the generalization for four different data-flow analyses, *available expressions*, *live variables*, *reaching definitions*, and *very busy expressions*, which again can be used for more specific analyses. For example, reaching definitions determines all assignments to variables that reach a given CFG element without being overridden. Reaching definitions can be used for detecting *double-free* errors, which occur if dynamically allocated memory (using `malloc` or similar) is freed multiple times (using `free`), resulting in undefined behavior. This kind of error can be detected by tracking pointers referring to freed memory and by determining whether they are passed to `free` again. By creating a variability-aware version of MONOTONE FRAMEWORKS, we can also easily create variability-aware versions of analyses such as reaching definitions and double-free errors (Section 4.2).

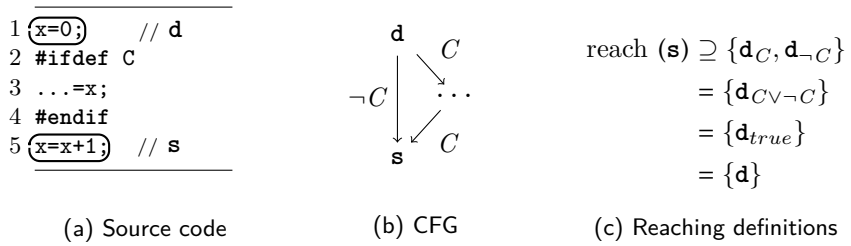
MONOTONE FRAMEWORKS describes a generic framework that can be parameterized with relevant functions for specific analyses. Its most central abstraction is to track data-flow facts across nodes of the control-flow graph, where a transfer function indicates how to change facts on a control-flow edge and a join function indicates how to merge data-flow facts from multiple paths. The specific facts and functions differ for each analysis, for example, in a *double free* analysis, data-flow facts would be the freed variables.

To implement variability-aware MONOTONE FRAMEWORKS, we systematically work with variational data.

- *Nodes and successors:* Where the original algorithm works on traditional CFGs, we work on variational CFGs, in which each node may occur only in some configurations (denoted as $\texttt{Node}_{PC}$; a choice of nodes can always be translated to a list of optional nodes). As described previously, successor and predecessor functions of an optional node return all successors in the node's presence condition, which may include optional and alternative nodes (denoted as $\texttt{Set[Node}_{PC}\texttt{]}$, which can be trivially derived from `List[Choice[Node]]`).
  In variability-aware MONOTONE FRAMEWORKS, this successor or predecessor function (depending on the specific analysis) is F (with $F : \texttt{Node}_{PC} \rightarrow \texttt{Set[Node}_{PC}\texttt{]}$).
- *Facts:* Where the original algorithm tracks a set of data-flow facts for each node (`Set[Fact]`), we track a variational set in which each fact may be relevant only for a subset of configurations, expressed by a presence condition PC (denoted as $\texttt{Set[Fact}_{PC}\texttt{]}$).
- *Transfer:* As facts are transferred along a CFG edge, facts are added or removed depending on the specifics of the analysis. As is common convention, we split the transfer function into *gen* and *kill* functions, which return the facts to be added or removed for a given node.
  In variability-aware MONOTONE FRAMEWORKS, *gen* and *kill* return variational sets of facts for a node with a corresponding presence condition ($gen, kill : \texttt{Node}_{PC} \rightarrow \texttt{Set[Fact}_{PC}\texttt{]}$).
- *Join:* The join function ($\sqcup$) determines how to join facts from different paths; the specific function depends on the analysis, but may be as simple as the union or the intersection of two sets of data-flow facts.
  In our variability-aware version of MONOTONE FRAMEWORKS, the join function joins two variational sets ($\sqcup : \texttt{Set[Fact}_{PC}\texttt{]} \times \texttt{Set[Fact}_{PC}\texttt{]} \rightarrow \texttt{Set[Fact}_{PC}\texttt{]}$), for example returning only facts in configurations shared by both sets ($F \sqcup G = \{x_{P \vee Q} |\ (x_P \in F) \wedge (y_Q \in G) \wedge (x = y)\}$).[12]
  In the example of Figure 5, if a *reaching definitions* analysis determines that a variable definition `d` can reach a statement `s` along two different paths, depending on a configuration option C, then the analysis computes that `d` reaches `s` under the condition $C \vee \neg C$, which evaluates to *true*.

Using variational sets ensures a redundancy-free storage of data-flow properties and obeys our local variability-representation principle. Presence conditions of data-flow properties stem from annotations

[12]It is straightforward to define other operations on variational sets in a similar way; for example, variational set union includes all elements from either sets, each with the presence condition as the disjunction of the conditions from either set.



(a) Source code (b) CFG (c) Reaching definitions

Fig. 5. Example of a presence-condition join in a *reaching definitions* analysis.

on AST nodes and annotations on edges during the CFG traversal. Internally, presence conditions are represented with BDDs and variational sets are represented as maps from facts to conditions (`Map[T, PC]`, see Walkingshaw et al. [61]), providing an efficient way to cope with variability information.

Equations (1) and (2) define our variability-aware extension of Monotone Frameworks. As in the original Monotone Frameworks [49], the computation of data-flow properties is based on the iterative computation of the functions $\text{Analysis}_\circ$ and $\text{Analysis}_\bullet$ for all CFG elements until the results stabilize at a fixed point. Both functions correspond to a node's entry and exit with respect to a forward analysis (using the flow function *succ*) or backward analysis (using the flow function *pred*). Specifically, $\text{Analysis}_\bullet$ applies the analysis' transfer function through the *gen* and *kill* functions (Equation 2). Our variability-aware version works just the same, except that it considers variational data structures in all of its parts, as described above. The fixed-point computation incorporates presence conditions to satisfy Monotone Frameworks' ascending chain condition. It ensures that at the analysis' fixed point no additional analysis information will be gained.

$$\text{Analysis}_\circ(l) = \bigsqcup \{ \text{Analysis}_\bullet(l') \mid l' \in F(l) \} \sqcup \iota^l_{\text{E}} \tag{1}$$

$$\text{where } \iota^l_{\text{E}} = \begin{cases} \iota & \text{if } l \in E \\ \bot & \text{if } l \notin E \end{cases}$$

$$\text{Analysis}_\bullet(l) = (\text{Analysis}_\circ(l) \setminus \mathit{kill}(l)) \cup \mathit{gen}(l) \tag{2}$$

F	is a flow function (either *succ* for forward analyses or *pred* for backward analyses)
E	denotes the set of entry or exit nodes (including presence condition), respectively of the intra-procedural CFG
ι^l_{E}	represent the $\mathit{initial}(\iota)$ or $\mathit{final}(\bot)$ variational facts in the data-flow computation
$\setminus$ and $\cup$	are standard set operations, extended to variational sets

Example. Using the definitions of variability-aware Monotone Frameworks, we illustrate a concrete analysis by instantiating its parameters (F, *gen*, *kill*, $\sqcup$, ...) for a variability-aware *double free* analysis. Double free is a forward analysis (F = *succ*) tracking which variables point to freed memory. Starting with empty data-flow facts ($\bot = \emptyset$, $\iota = \emptyset$), we generate facts with corresponding conditions for every

invocation of `free` and remove facts for assignments:

$$gen(\mathrm{s}_A) = \begin{cases} \{x_A\}, & \text{if s = } \texttt{free(x)} \\ \emptyset, & \text{otherwise} \end{cases} \tag{3}$$

$$kill(\mathrm{s}_A) = \begin{cases} \{x_A\}, & \text{if s = } \texttt{x = e} \\ \emptyset, & \text{otherwise} \end{cases} \tag{4}$$

When control-flow paths are joined, we also join the data-flow facts ($\sqcup = \cup$) to track if a variable was already freed in any path.

In addition to *double free*, we implemented four other intra-procedural variability-aware data-flow analyses, which we describe in Section 4.2.

4 EMPIRICAL STUDY

Despite a compact representation of variability, variability-aware analyses cannot beat exponential worst-case complexity. Nonetheless, we conjecture that variability-aware static analyses are possible and efficient in real-world systems, despite huge configuration spaces. Specifically, we evaluate the feasibility and scalability of different analysis techniques by implementing seven variability-aware static analyses (Section 4.2) and by evaluating them on five real-world, large-scale configurable systems (Section 4.3)—a scale and realism that substantially increases external validity, compared to previous work, which concentrated mostly on formal foundations, made limiting assumptions, or relied on comparatively small and academic case studies (see our discussion of related work in Section 5). In particular, we compare variability-aware analysis and three state-of-the-art sampling techniques (*single conf*, *code coverage*, and *pair-wise*), which we introduce in Section 4.4.

In Section 4.1, we describe our research questions and hypotheses. We describe the static analyses in Section 4.2 and the highly configurable systems that we use in our experiments in Section 4.3. In Section 4.6, we describe our experiment setup. In Section 4.7, we report on our experience with sampling and in Sections 4.8 and 4.9 we present and discuss our results.

4.1 Research Questions and Hypotheses

In our study, we address two research questions on the efficiency and effectiveness of variability-aware static analysis compared to sampling:

- Is variability-aware analysis efficient (i.e., scalable) on our corpus of large-scale case studies?
- How many more warnings does a variability-aware analysis find (which covers the entire configuration space) compared to state-of-the-art sample-based analyses?

Based on the research questions and properties of variability-aware analysis and sampling, we formulate three hypotheses.

- *Efficiency:* Analyzing all system variants simultaneously using variability-aware analysis is slower than analyzing only a sample set of variants. The reason is that the variational program representation (e.g., the variational CFG) covering all variants is larger than the one of any single variant, and most sampling techniques generate relatively small sample sets.

> HYPOTHESIS H_1
> *The execution times of variability-aware static analysis are* longer *than the corresponding times of analyzing the individual variants derived by sampling.*

- *Effectiveness:* With respect to effectiveness, we hypothesize that variability-aware analysis generates a significant number of warnings whose reasons are missed by sampling.

> HYPOTHESIS H_2
> *Variability-aware analysis reports a significantly* larger *number of warnings than any analysis based on sampling.*

- *Interaction degrees:* Furthermore, we are interested in how many variants of a program are affected by the warnings that our analyses report, to assess the warning's relevance. For this purpose, we use a measure of interaction degree (see Section 4.5) that characterizes how many options are involved in an interaction. The higher an interaction degree of a warning, the fewer system configurations are affected and the harder the issue is to detect with sampling [42]. We compare for each file the interaction degrees of warnings of the variability-aware analyses against the interaction degrees of warnings reported by sampling.

> HYPOTHESIS H_3
> *An analysis based on sampling reports* fewer *warnings with high interaction degrees than variability-aware analysis.*

4.2 Static Analyses

We have implemented different, variability-aware static analyses to detect common potential programming errors (according to CERT secure coding recommendations [51]) in all system variants. In this section, we give a short description of these analyses. A detailed description is available in the appendix (*function return* and *double free* are also described in Section 3.1 and Section 3.2, respectively).

- *Case termination* (control-flow analysis): A `case` label of a `switch` statement is followed by a number of statements and an optional `break` statement. The `break` causes the control flow to jump beyond the `switch`. Even though, a `case` statement without a accompanying `break` statement is valid C code, omitting the `break` statement may lead to unintended control flows and should be avoided.[13] Our analysis inspects all control flows through `case` code and asserts that each flow ends in a `break` statement.
- *Dangling switch* (control-flow analysis): A `switch` body may contain any sequence of statements including declarations of variables. If a programmer places code before the first `case` label, the code is never executed (i.e., it dangles in the `switch` statement), which is usually unintended. To find such code, our analysis computes CFGs for `switch` bodies and checks whether, apart from declarations without initialization code, any control-flow statements occur that are not preceded by `case` or `default` labels in any configuration.
- *Function return* (control-flow analysis): According to the C standard, the control flow of all functions with a non-void return type should execute a `return` statement. Using the return value of a function that does not execute a `return` leads to undefined behavior. Our analysis checks whether there is a path in the variational CFG of a function with a non-void return type that does not end in a `return` statement.
- *Dead store* (data-flow analysis): Assignments to local variables that are not used in subsequent program code are called *dead store*s. A dead store is usually the result of a logic programming error

[13] Several C code guidelines state the use of a `break` statements after processing a `case` block (e.g., Seacord [51]), and static-analysis tools such as, COVERITY, KLOCWORK, and POLYSPACE BUG FINDER, support the detection of unterminated `case` statements.

and should be removed. To detect dead stores, we use a variability-aware liveness analysis [41, 49] and determine whether assigned variables are read in subsequent program code.

- *Double free* (data-flow analysis): Double free errors in C programs occur when `free` is called more than once with the same memory address as an argument. Calling `free` twice on the same pointer can lead to a memory leak. To identify double-free errors, our analysis determines whether a pointer variable passed as an argument to the function `free` is passed to another `free` call without any reassignments of memory.
- *Error handling* (data-flow analysis): In C, error codes and messages are often encoded as function return values. In particular, library functions, such as `malloc`, return either an address to newly allocated memory or an error code. To avoid undefined program behavior, the return value should be checked for this error code before it is used. To determine missing error-code checks, our analysis tracks the results of 31 standard-library calls with a variant of a reaching-definition analysis [49]. The analysis checks whether variables assigned with the results of standard-library calls reach in control structures, checking them for predefined error codes.
- *Freeing static memory* (data-flow analysis): Freeing memory that was not allocated dynamically using memory-management functions can result in serious errors (e.g., heap corruption or abnormal program termination). For this reason, only pointer variables that point to memory previously allocated with memory-management functions, such as `malloc` or `realloc`, should be freed with function `free`. To detect such errors, our analysis tracks pointer variables that are not initialized with dynamic memory, and it determines whether they are passed correctly to memory-management functions.

4.3 Subject Systems

To test our hypotheses and to answer our research questions, we analyzed five subject systems. We looked for publicly available systems (for replicability) that are of substantial size, actively maintained by a community of developers, used in real-world scenarios, and that contain substantial compile-time variability controlled by the C preprocessor. The systems must provide, at least, an informal variability model that describes configuration options and their valid combinations [39]. Table 1 summarizes the most important figures from the subject systems. We would like to acknowledge the pioneering work on variability-model extraction [8, 54] and build-system analysis [7], which enabled us, for the first time, to conduct variability-aware analysis on substantial, real-world systems that are an order of magnitude larger then previous work on Java-based subjects [5, 11, 30].

- The Busybox tool suite implements most standard Unix tools for resource-constrained platforms. With 792 configuration options, it is highly configurable; most of the options refer to independent and optional subsystems; the variability model in conjunctive normal form has 993 clauses. Specifically, we use BUSYBOX version 1.18.5 (522 files and 191 615 lines of source code).
- The Linux *kernel* (x86 architecture, version 2.6.33.3) is an operating-system kernel with millions of installations worldwide, from high-end servers to mobile phones. With 6 918 configuration options it is highly configurable. In a previous study, we identified the LINUX kernel as one of the largest and most complex (w.r.t. variability) publicly available configurable software systems [39]. It has 7 691 source code files with 6.5 million lines of code. Note that already the variability model of LINUX is of substantial size: the corresponding formula in conjunctive normal form (CNF) has over 60 000 variables[14] and about 300 000 clauses; a typical satisfiability check requires half a second on a standard computer.

[14]The number of variables in the CNF formula exceeds the number of configuration options due to extra variables for the equi-satisfiable transformation to CNF and due to extra variables to encode trystat variables [28, 52].

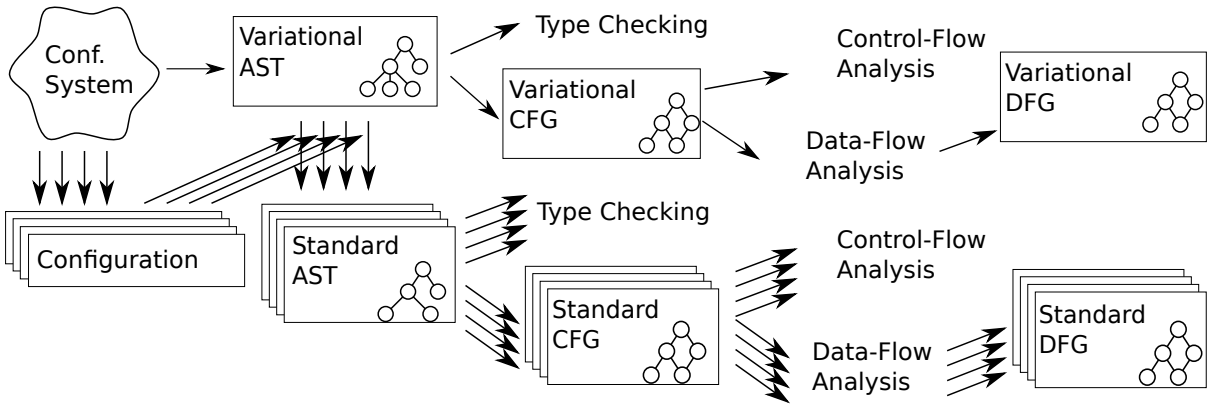


Fig. 6. Experiment setup showing type checking, one control-flow analysis and one data-flow analysis.

- The cryptographic library OpenSSL implements different protocols for secure Internet communication. OpenSSL can be tailored to many different platforms, and it provides a rich set of 589 configuration options. We consider OpenSSL version 1.0.1c with 733 files and 233 450 lines of code. Since OpenSSL does not come with a formal variability model, we extracted a variability model based on lexer, parser, and type errors using the approach of Nadi et al. [47]. The resulting variability model has 15 clauses.
- SQLite is a library implementing a relational database-management system. It gained much attention in software development due to a number of desirable properties (e.g., cross-platform support, transactions, and small footprint) and is considered the most widespread database-management system worldwide, with installations as part of Android, Firefox, and PHP. To ease embedding SQLite into other software systems, SQLite's code base consists of only two large source-code files (amalgamation version; one header file and one code file). We use a recent version of SQLite (3.8.1) with 143 614 lines of C code, which can be configured using 93 configuration options.
- uClibc is an alternative, resource-optimized C library for embedded systems. We analyze the x86_64 architecture in uClibc (v0.9.33.2), which has 1 628 C source files (63 147 lines of C code) and 367 configuration options described in a Kconfig model. Our implementation is based on the work of Nadi et al. [47].

Table 1. Lines of code and number of configuration options for all subject systems

	Lines of code	Number of configuration options
Busybox	191 615	792
Linux	6 572 434	6 918
OpenSSL	233 450	589
SQLite	143 614	93
uClibc	63 147	367

4.4 Sampling Techniques

Using sampling in the analysis of configurable systems has its roots in early approaches of testing software [48]. Due to the sheer size and complexity of real-world configurable systems (the number of variants can grow exponentially with the number of configuration options), a brute-force approach of analyzing all variants individually is not feasible. Instead, sample-based approaches use knowledge about dependencies between configuration options and derive sample sets of configurations. The sample configurations are used to derive system variants, which are analyzed either with testing or, in our case, with static analyses. We selected three sampling heuristics that are common in practice: *single conf*, *code coverage*, and *pair-wise coverage*.

- *Single conf*: The simplest sampling heuristic is to analyze only a single representative system variant that enables most, if not all, of the configuration options of the configurable system. Typically, the variant is selected manually by a domain expert. For example, according to Dietrich et al. [21], in the Linux development community, it is common to analyze only one predefined variant with most configuration options selected, called *allyesconfig*. Similarly, many software systems come with a default configuration that satisfies most users and that usually enables many configuration options.
- *Code coverage*: The goal of code-coverage sampling is to select a minimal sample set of variants, such that every lexical code fragment of the systems' code base is included in, at least, one system variant. In contrast to single-conf sampling, code-coverage sampling covers also mutually exclusive code fragments.
  Although there is an algorithm to compute an optimal solution (a minimal set of variants) by reducing the problem to calculating the chromatic number of a graph, this algorithm is NP-complete and by far too slow for our case studies.[15] Instead, we resort to the conservatively approximated solution of Tartler et al. [55], which speeds up the computation of the sample set significantly at the cost of producing a sample set that is possibly larger than necessary.
  A subtle problem of this technique arises from the issue of how to treat header files [42]. Header files often exhibit their own variability, not visible in the C file without expanding macros. Due to the common practice of including files that themselves include other files, a single `#include` statement in the source code can bloat the code base of a single file easily by an order of magnitude, something we frequently observed in Linux, in which, on average, 300 header files are included in each C file [31]. Furthermore, some header files may be included only conditionally, depending on other `#ifdef` directives, such that, for a precise analysis of all header code, sophisticated analysis mechanisms become necessary (e.g., using symbolic execution of the preprocessor code) [24, 31, 37]. This explosion and cost can make precise analyses that include header files unpractical or infeasible, even with Tartler's approximate solution [54]. For some of our case studies, computing code coverage including configurable code in header files is not feasible (cf. Section 4.7). Therefore, we focus our code-coverage heuristic on variability in C files (including variability induced by macros) and ignore the variability from header files (see experiment setup in Section 4.6 and threats to validity in Section 4.10).
- *Pair-wise*: Pair-wise sampling is motivated by the hypothesis that many faults in software systems are caused by interactions of, at most, two configuration options [12, 36, 50, 53]. Using pair-wise sampling, the sample set consists of a minimal number of samples that cover all pairs of configuration options, whereby each sample is likely to cover multiple pairs.
  The computation of pair-wise sample sets is not trivial if constraints, such as "A implies B or C", exist in a variability model [42]; in fact, it is NP-complete (similar to the minimum set cover

[15] For more details on the optimal algorithm, see https://github.com/ckaestne/OptimalCoverage/.

problem) [26]. Hence, existing tools apply different conservative approximations to make the computation feasible for large systems with many configuration options. For our experiments, we use the state-of-the-art tool SPLCATool[16] by Johansen et al., which computes pair-wise samples using covering arrays [26]. The computed sample set covers all pair-wise interactions in a given system, but it is not guaranteed to be the minimal sample set. For our evaluation, we generate one set of pair-wise configurations for each subject system based on its variability model. In some subject systems, files may have additional constraints on features (i.e., files are only compiled if certain options are active). Therefore, some of our global pair-wise configurations might not apply for some files, and we may fail at a complete coverage of pair-wise interactions for some files. Furthermore, it may happen that some sample variants of a file are very similar, as the difference in the respective sample configurations affect the content of a file only to a minor or no extent. Alternatively, we could generate a pair-wise sampling set for each file (including the file constraints), but due to the high number of files in some subject systems (7 221 in Linux), we chose to use the global pair-wise sample set.

4.5 Interaction Degree

We use *interaction degree* [23] as a measure for how many options need to be selected in a specific setting for a warning to appear. For example, for a warning with presence condition $\neg A \wedge B$, the interaction degree is two as two options need to be assigned. The interaction degree may be smaller than the number of options occuring in the presence condition: For example, the interaction degree of a warning with presence condition $(C \vee D) \wedge (E \vee (F \wedge G))$ is also two because only two options (C and E or D and E) need to be selected to satisfy the condition.

In practice, there is no efficient way to exactly measure the interaction degree for arbitrary formulas, since configuration options in presence conditions might have complex dependencies. In our evaluation, we use a heuristic: We represent the presence condition in a binary decision diagram (BDD) and take the length of the shortest path through the BDD as an approximation for the interaction degree.[17] For complex presence conditions and unfavorable BDD-variable orders, this heuristic can generate interaction degrees above the minimum, which we discuss as a threat to validity in Section 4.10. The length of the shortest path is guaranteed to be equal or greater than the precise interaction degree, though.

In addition, external knowledge regarding configuration-option dependencies make the interaction-degree measurement more difficult. Such knowledge, which might contain implications on value propagation among options, can actually result in fewer options that need to be assigned. For example, the presence condition $A \wedge B$ has degree two, but if A requires B (possibly specified in a variability model) and vice versa, then the presence condition should have degree one. Considering these dependencies while computing interaction degrees has exponential complexity in the number of configuration options in presence conditions and the variability model [59], which makes it infeasible for most practical systems. External knowledge about dependencies is rarely codified, but for Linux, OpenSSL, and uClibc, we also applied a heuristic for presence-condition simplification that we recently developed [59], to improve our interaction-degree measurements. Including external knowledge did not change the big picture of our results, though; therefore, we do not report it separately.

[16] We use the fork https://github.com/alexrhein/SPLCATool/.

[17] The shortest path trough the BDD corresponds to the shortest clause in a DNF representation of the presence condition. The set of variables in this clause is the (heuristic) minimal set of variables that need to be enabled or disabled to satisfy the presence condition. A link to the function definition for interaction-degree computation is available on our supplementary Web site.

We use the interaction degree to evaluate the relevance and complexity of warnings found by our analyses for our third hypothesis.

4.6 Experiment Setup

We implemented our control-flow and data-flow analyses on top of the TypeChef framework. For each file, we run TypeChef once. It first builds a variational AST (lexing and parsing) and then type-checks the AST. Second, we generate a variational CFG for each function and run our control-flow analyses on the generated CFG. Third, we run our data-flow analyses, each of which generates a variational data-flow graph (DFG). The data-flow analyses rely on type information obtained in the first step. For each analysis run, we measure the execution time and log all warnings along with their presence conditions.

To avoid bias due to different analysis implementations, we use our infrastructure for variability-aware analysis also for the analyses based on sampling. To this end, we generate individual variants (ASTs without variability) based on the sample sets generated by the sampling heuristics (cf. Section 4.7). We create an AST for a given configuration by pruning all irrelevant branches of the variational AST, so that no `Choice` nodes remain. As there is no variability in the remaining AST, the analysis never splits, and there is no overhead due to reasoning about presence conditions, because the only possible presence condition is *true*.

Note that our implementation is certainly not as precise or efficient as related static-analysis tools (e.g., data-flow analysis in the Coverity tool), which makes our evaluation of sampling likely slower and less accurate than possible. However, existing tools can not be easily reused for configurable-system analysis beyond sampling and, therefore, we have to use a dedicated implementation (TypeChef). Furthermore, we need tool implementations with comparable analysis capabilities to compare the results of variablity-aware analysis against sampling heuristics. Therefore, using TypeChef uniformly is a justifiable decision.

As our analyses are intra-procedural, it would have been possible and more efficient to apply sampling to individual functions and not to files, as done by Brabrand et al. for Java product lines [11]. Unfortunately, preprocessor macros in C rule out this strategy, as we cannot even parse functions individually without running the preprocessor first or without performing full variability-aware parsing [31]. In our running example of Figure 1, we would not even have noticed that function `foo` is affected by configuration option A, because variability comes from variational macros defined outside the function. Variational macros defined in header files are very common in C code [31]. To give additional evidence for the variability arising from macros, we analyzed the number of configuration options in functions before/after macro resolution. We used srcML[18] to measure the number of configuration options in functions before macro resolution and our tool TypeChef to get statistics on functions after macro resolution. Then, we compared the number of configuration options for functions that occurred in both statistics and show the mean in Table 2. In all systems, macro resolution significantly increases the variability in functions, especially in Linux. This supports our decision against parsing functions individually, without macro and header preprocessing.

In Figure 6, we provide an overview of our experiment setup. Depending on the sampling technique, one or multiple configurations are checked. For each file of the five subject systems, we ran the variability-aware analyses and the three sampling approaches (the latter consisting of multiple internal runs)—in total, four analysis strategies per subject system: one variability-aware and three sampling. For each file and for each sample configuration, we ran the static analyses in separate runs, measured the time spent in the analyses, and logged the reported warnings including their presence conditions for later analysis. In

[18] http://www.srcml.org/

Table 2. Average number of configuration options per function with and without macro resolution

	Options per function without macros	Options per function with macros
BUSYBOX	0.4	0.8
OPENSSL	0.3	0.8
SQLITE	0.4	2.4
LINUX	0.07	9.5
UCLIBC	0.7	1.6

Section 4.8, we show the speedups of variability-aware analysis, compared to sampling and per subject system. Furthermore, we provide statistics on the interaction degrees of reported presence conditions and their distribution.

We ran all measurements on LINUX machines (Ubuntu 14.04) with an Intel Xeon E5-2690 at 3.0 GHz. Each analysis was started with 5 GB RAM. For one file, we had to increase the memory limit to 8 GB.

4.7 Experience with Sampling

Before we discuss the performance measurements, we would like to share our experience with applying sampling in practice, which was surprising to us. We expected that contemporary sampling tools can quickly compute representative sample sets. However, we found that deriving samples at this scale is far from trivial, and we even failed to compute sample sets for some sampling techniques (code coverage with variability from header files for LINUX). We generated a global pair-wise sample set per subject system and not per file, even though we analyze files separately (see Medeiros et al. [42], for an in-depth discussion).

Single-conf sampling worked well. LINUX has a commonly used configuration *allyesconfig*, which is maintained by the community and frequently used for analysis purposes.[19] For BUSYBOX and UCLIBC, we used the system's build script (`make defconfig`). For SQLITE and OPENSSL, we created large configurations by manually selecting as many configuration options as possible.

In contrast, sampling based on code coverage needs to investigate every file separately. We reimplemented the conservative algorithm of Tartler et al. for this task [55], excluding variability from header files (cf. Section 4.4).

To compute pair-wise sample sets, we only found one research tool (SPLCATOOL), which is able to compute a complete set of pair-wise configurations for a given variability model. SPLCATOOL did reasonably well for BUSYBOX, OPENSSL, SQLITE, and UCLIBC, but the larger configuration space of LINUX made the computation of the sample set very expensive (>20 h for generation of the pair-wise sample set). Also in this case, the computation time exceeded the times for performing variability-aware static analysis.

The high upfront investment for sample-based analyses (generation of the sample sets) must be seen in relation to the high implementation effort for a variability-aware analysis. In practice, a decision must be based on the sampling technique, the complexity of the configurable system (number of variants), and the availability of a variability-aware analysis, which is as also confirmed by a recent study comparing sampling strategies [42].

[19] http://kernel.org/doc/Documentation/kbuild/kconfig.txt

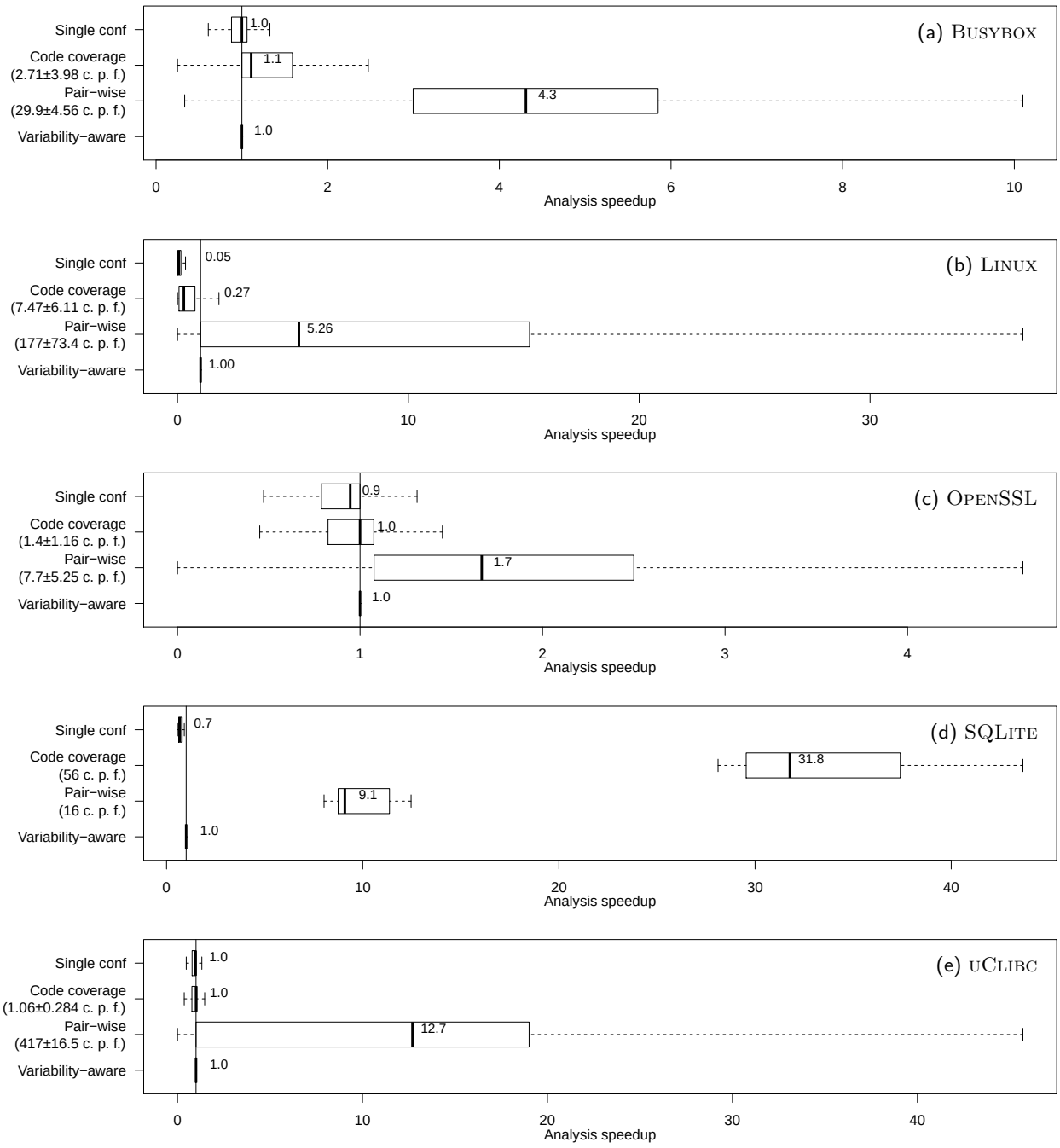


Fig. 7. Speedup of variability-aware analysis compared to sampling (outliers omitted). We compute the speedup separately per file and analysis and plot the distribution of speedups per sampling strategy. For code-coverage and pair-wise sampling, we show the size of the sample set (configurations per file; c.p.f.). Single-conf sampling analyzes one configuration and variability-aware covers all configurations. We write the median speedup value next to the median marker in the boxplots.

4.8 Efficiency (H_1)

Results. We compute the *speedup* of variability-aware analysis over a particular sampling strategy per file and per analysis for each subject system. The speedup over a sampling strategy X is computed as (*time for strategy X*) / (*time for variability-aware analysis*).[20] With variability-aware analysis being the reference, a speedup value < 1.0 means that sample-based analysis is faster than variability-aware analysis, as for example, single-conf sampling for Linux. A speedup value > 1.0 means that variability-aware analysis is faster than the respective sample-based analysis, as for example, pair-wise sampling for Linux (i.e., here variability-aware analysis is 5.26 times faster).

In Figure 7, we plot the distribution of speedups across all files and analyses (boxplots on a logarithmic scale). For each of the sample-based analyses, we provide in Figure 7 the number of analyzed configurations per file, which is the primary cost driver for sampling-based approaches (below the name of the analysis, 'configs per file' or short 'c.p.f.' in terms of mean $\pm$ standard deviation; single-conf sampling analyzes one variant for each file). In Table 3, we show the speedup across all files of a subject system *per analysis*. We highlight non-negligible speedups: speedups > 1.2 with green and speedups < 0.8 with red background color. In addition, we provide the raw measurement results on the supplementary Web site. For clarity, we report sequential times, though parallelization would be possible in all cases.

As visible in Table 3, the results are not entirely consistent, but a fairly clear pattern emerges:

- As expected, for nearly all subject systems and analyses, variability-aware analysis is *slower than single-conf sampling* or as fast. The slowdown is often not severe (>0.5), except for Linux, the system with the highest amount of variability. A notable exception are runs of the *case termination* analysis on Busybox and uClibc (marked with '†' in the table). A closer look at the raw data reveals that the runtimes of *case termination* are very fast (usually below 10 ms), so the differences can be attributed to measurement noise (a t test cannot identify statistically significant time differences across all files).
- Variability-aware analysis is consistently faster than pair-wise sampling, often by an order of magnitude, with a single exception (*dead store* on Linux, marked with ‡; likely caused by a high number of warnings in this analysis, which leads to many SAT calls in variability-aware analysis).
- The results regarding code-coverage sampling are mixed. They depend more on implementation aspects of the subject system that drive the sampling cost (see number of samples reported in Figure 7). For example, variability-aware analysis is much fastr for *dangling switch* in SQLite (speedup 43.7; 56 configurations sampled per file) and much slower for *error handling* in Linux (speedup 0.04; 7 configurations sampled per file). The reason that code-coverage sampling requires often only small samples is that it ignores `#ifdef`s that occur in header files; whereas SQLite already includes most functionality that is normally included by headers. Variability-aware analysis considers variability in header files, so it has to analyze larger configurations with additional header code.

Summary

Overall, except for few outliers, variability-aware analysis is slower than single-conf sampling and faster than pair-wise sampling. The performance of code-coverage sampling depends on coverage effectiveness, influenced by variability from header files.

[20] If a sampling strategy or the variability-aware strategy reached a timeout or out-of-memory error for a file, then we skip this file–strategy combination.

Table 3. Speedup of variability-aware analysis across all files (excluding timeouts and errors); a speedup < 1.0 means that sampling is faster; a speedup > 1.0 means that variability-aware analysis is faster; we highlight speedups > 1.2 with green and speedups < 0.8 with red background color. Cells with † and ‡ mark outliers, as explained in the text.

	Variability-aware vs.	*function return*	*case termination*	*dangling switch*	*dead store*	*error handling*	*double free*	*freeing static memory*
BusyBox	Single conf	1.02	1.58 †	1.03	1	0.997	0.789	1
	Code coverage	1.5	2.03	1.46	1.44	1.38	1.03	1.37
	Pair-wise	4.46	4.84	4.54	6.34	5.09	2.86	5.18
Linux	Single conf	0.0978	0.017	0.192	0.0182	0.00635	0.177	0.125
	Code coverage	0.575	0.0641	1.34	0.106	0.0403	1.03	0.981
	Pair-wise	9.19	0.843	22.8	1.99	0.641 ‡	15.6	14.6
OpenSSL	Single conf	0.948	0.988	0.975	0.932	0.925	0.785	0.924
	Code coverage	1.07	1.1	1.06	1.08	1.05	0.866	1.05
	Pair-wise	1.52	1.48	1.55	2.8	2.52	1.71	2.55
SQLite	Single conf	0.904	0.559	0.862	0.635	0.724	0.734	0.652
	Code coverage	39.5	33.4	43.7	28.1	35.3	29.4	30.2
	Pair-wise	12.1	9.22	12.5	8.03	10.6	8.95	8.96
uClibc	Single conf	0.943	1.31 †	1.01	0.895	0.934	0.795	1.02
	Code coverage	0.936	1.39	1.01	0.907	0.953	0.83	1.03
	Pair-wise	2.73	4.4	4.92	15	23.9	15.4	24

Discussion. Our experiments show mixed results concerning hypothesis H_1: For all five subject systems, variability-aware analysis is faster than pair-wise sampling (averaged over all analyses), but slower than using single-conf sampling. Variability-aware analysis is sometimes faster and sometimes slower than code-coverage sampling. The performance of code-coverage sampling depends on the implementation of variability in the respective files; the number of sampled variants and performance results differ considerably between files inside each subject system (Figure 7; annotations on the y-axes). So, the performance of the code-coverage heuristic is hard to predict and depends strongly on individual implementation patterns.

A further observation is that the speedup of variability-aware analysis in relation to sampling is very different across all static analyses (Table 3). This can be explained by the different types of statements that are considered by the analyses (e.g., `switch` or `return`) and by the fact that there are considerable differences in the number of warnings reported by the analyses.

The experimental results for BusyBox, Linux, and OpenSSL demonstrate that variability-aware analysis is in the range of the execution times of sampling with multiple samples (code coverage and pair-wise). So, we conclude that variability-aware analysis is practical for large-scale systems. An important finding is that the overhead induced by SAT solving during variability-aware analysis is not a bottleneck, not even for large systems, such as the Linux kernel. Overall, variability-aware analysis (compared to single conf) of BusyBox, OpenSSL, SQLite, and uClibc takes as much time as checking two variants (median values; 61 variants in Linux). All values are very low compared to the number of the possible variants of the respective system, showing that a (complete) variability-aware analysis is already possible at the cost of an (incomplete) sampling heuristic.

Table 4. Effectiveness of the three sampling strategies given as the percentage of warnings reported by a sampling strategy as compared to variability-aware analysis; the overall effectiveness is reported as geometric mean.

	Warnings	Effectiveness in percent		
		Pair-wise	Code coverage	Single conf
BusyBox	6 379	98	99	85
Linux	2 657 677	97	59	40
OpenSSL	7 844	90	94	79
SQLite	1 750	100	100	64
uClibc	23	43	100	91
Overall		82	89	69

Insight

In most systems, analyzing a file with variability-aware analysis takes, on average, as much time as analyzing two variants individually (61 variants in Linux*).*

4.9 Effectiveness (H_2 and H_3)

Results. To evaluate the effectiveness of variability-aware analysis as compared to sampling (H_2), we inspect and compare the reports of the static analyses. We report effectiveness of a sampling strategy as the percentage of warnings found with this sampling strategy out of all warnings found with the variability-aware strategy. By construction, variability-aware analysis finds all warnings (that a specific analysis can find) across all configurations, whereas sampling strategies typically find only a subset (but cannot find additional ones).

In all subject systems, we found a large number of warnings. In Table 4, we report the number of warnings as well as the effectiveness of each sampling strategy across all subject systems. As hypothesized (H_2), variability-aware analysis finds more warnings than the sampling strategies in most systems. While the specifics differ from system to system, depending on their implementation, as discussed below, we see that pair-wise sampling and code-coverage sampling are generally more effective than single-conf sampling.

To understand better how these results are influenced by the interaction degree of the warnings (H_3), we plot the number of warnings and the relative effectiveness of the sampling strategy per interaction degree in Figures 8 and 9. Overall, although the distributions differ again by subject system, warnings occur most commonly at low interaction degrees, with fewer warnings at higher degrees (Figure 8); Linux and OpenSSL contain more variability and generally skew toward higher degrees. The percentage of warnings found by sampling strategies, as expected (H_3), is lower for higher interaction degrees (Figure 9): All sampling strategies find all warnings with interaction degree 0 (which occur in all variants), hence the high warning rates in many systems. However, with higher interaction degrees, the effectiveness of some sampling strategies drops; warnings are no longer detected reliably but based on randomness, which is inherent to some extent in our sampling strategies.

Overall, analyzing only a single configuration, single-conf sampling always reports the smallest number of warnings. Code-coverage sampling and pair-wise sampling report much higher numbers of warnings, often most of the warnings across all variants.[21]

One interesting observation is that there is a warning with interaction degree 16 in BUSYBOX that is found by pair-wise and code-coverage sampling, but not by single-conf sampling. We manually inspected this warning (uninitialized-memory access in BUSYBOX) and found that the presence condition is a conjunction of 16 deactivated configuration options ($\neg A \wedge \neg B \wedge \neg C \wedge \ldots$). Pair-wise sampling generated (by chance) a configuration that covers this condition. Code-coverage sampling necessarily covers the condition because there is an AST node that is only enabled when the condition holds. Single-conf sampling does not cover the condition since the warning only applies to variants that have none of these options enabled.

Note that the total number of reported warnings is high. This might be due to false positives (inaccuracy in the underlying analysis) or because some programming guidelines checked by our analyses are not adopted in the subject systems. For example, dead stores might be more common because programmers initialize variables in all variants and make code that uses the variables optional, relying on the compiler to remove variables in variants where they are not used. Note that the majority of warnings are reported by analyses for *dead store* and *case termination*; therefore, our results—concerning interaction degrees—apply primarily to these static analyses.

SUMMARY

In all subject systems, we found a large number of warnings. The three sampling strategies can find many but not all of these warnings, with pair-wise and code-coverage finding more warnings than single-conf sampling. The effectiveness of sampling strategies drops with interactions of higher degrees, which occur in all systems.

Discussion. Our experiments confirm hypothesis H_2 and H_3: The sample-based analyses are not able to cover all warnings reported by variability-aware analysis, and their effectiveness drops at higher interaction degrees, across all subject systems. The reason is that sampling ignores code paths or even entire code files depending on the sample set, whereas variability-aware analysis reports results for all code blocks in all variants and files of each system. It is, however, interesting that pair-wise and code-coverage sampling reach almost as many reported warnings as variability-aware analysis in most systems. Code-coverage sampling usually misses more.

Concerning the observation of few warnings reported by single-conf sampling, we note that alternative code blocks (`#ifdef`s with `#else` case) are used very often in LINUX. The *allyes* configuration of LINUX can always cover only one of the respective alternatives, which excludes large parts of the source code and, due to build-system constraints, even entire files.

To evaluate how relevant the detected warnings are, we took a closer look at the interaction degrees of warnings reported by the different strategies. Figure 9 shows that many reported warnings have a degree of 3 to 6. That is, they require more than 3 configuration options enabled/disabled to be triggered. Likely, a warning with a *low* interaction degree (e.g., $A \vee B$) applies to *more* configurations than a warning with

[21] Some subtle points might be surprising in the data. Pair-wise sampling does not always detect all warnings for interaction degree up to 2 as one might expect: We generate global pair-wise sample sets per subject system, which are applied to all files, but some sample configurations may not be valid for certain files and are thus excluded from the analysis, loosing the guarantee of covering all pairs of options. Furthermore, pair-wise sampling finds many warnings of interaction degrees higher than 2, since each sample configuration contains many pairs that together form configurations that may detect certain higher-level interactions by chance, but coverage of higher-level interactions is not guaranteed.

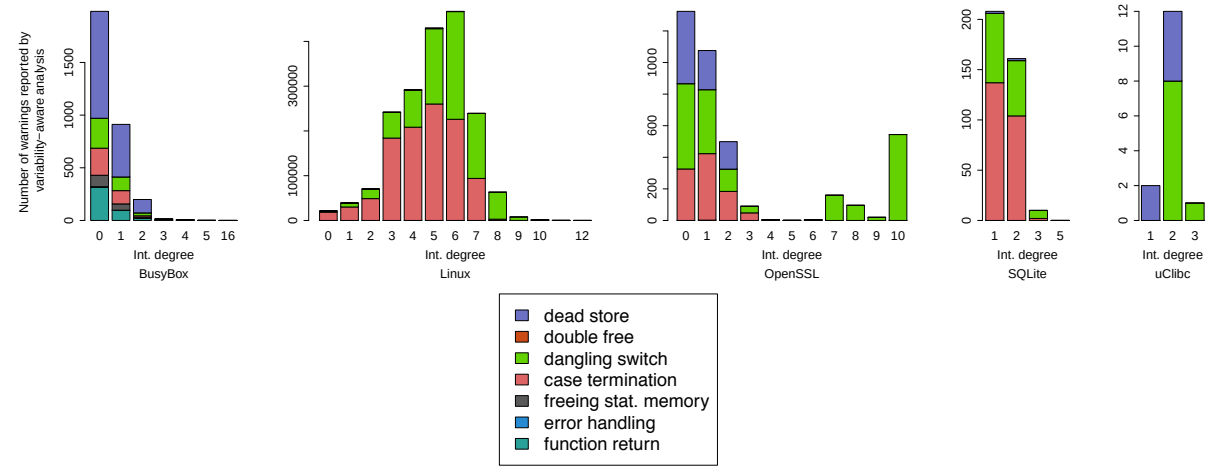


Fig. 8. Warnings reported by the variability-aware analysis by subject system and by interaction degree. For each interaction degree, warnings are reported by analysis, stacked in the plot in the same order as in the legend.

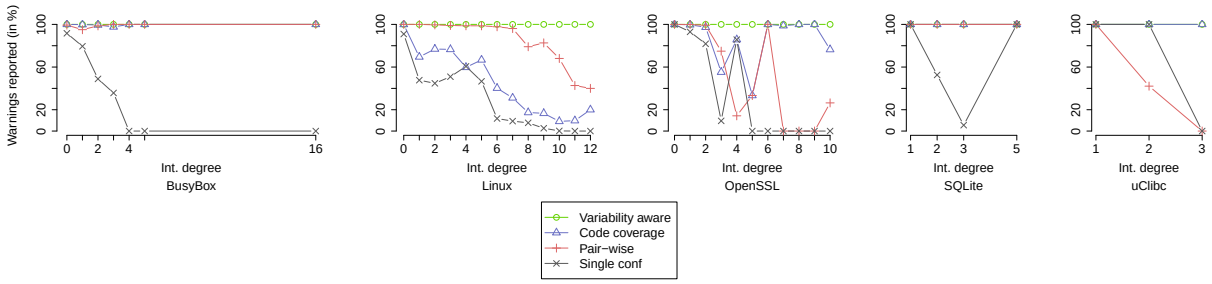


Fig. 9. Percentage of warnings reported per sampling technique compared to warnings found across all variants with variablity-aware analysis (Fig. 8).

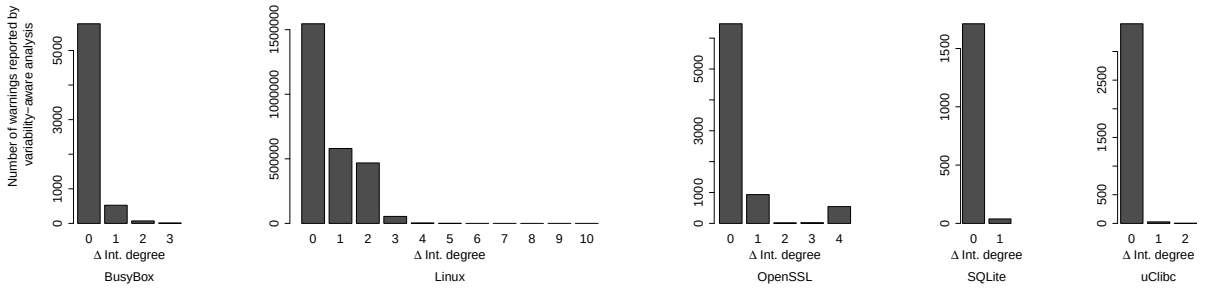


Fig. 10. Deltas of interaction degrees of warnings and their surrounding methods.

a high interaction degree (e.g., $A \wedge B \wedge C$). Therefore, a reported warning with a low interaction degree likely affects more configurations (and customers) but can also be more likely detected with sampling.

Our experiments also confirm hypothesis H_3: Warnings with high interaction degrees (especially between degree 3 and 6) are reported much less reliably when using sampling. This shows that it is unlikely to achieve a full coverage of smells and defects in the presence of high interaction degrees. This is especially grave for single-conf sampling, which reports only about half of the warnings found with variability-aware analysis.

INSIGHT

Pair-wise and code-coverage sampling report a relatively high number of warnings (compared to variability-aware analysis). Single-conf sampling reports much fewer warnings, especially with interaction degrees between 3 and 6.

4.10 Threats to Validity

A threat to internal validity is that our implementation of variability-aware analysis supports ISO/IEC C, but not all GNU C extensions used in the subject systems (especially LINUX); our analyses simply ignore corresponding code constructs. Also due to the textual and verbose nature of the C standard, the implementation may not align entirely with the behavior of the GNU C compiler. Due to these technical problems, we had to exclude few problematic files (6 files from BUSYBOX, 11 from OPENSSL and 19 from LINUX). Furthermore, we had timeouts and out-of-memory errors in some analyses/files of OPENSSL and LINUX (less than 2% of the files). All affected files have been excluded from sampling as well. Still, we argue that the large numbers of 517 files of BUSYBOX, 723 files of OPENSSL, 7 637 files of LINUX, and 1 288 files of UCLIBC[22] successfully analyzed with variability-aware analysis deems our approach practical and our evaluation representative.

We rely on warnings that have been reported by a number of standard static control-flow and data-flow analyses, which are known to generate false positives [27]. A manual analysis of a sample of the reported warnings confirmed the presence of false positives. But, although we do not know how many of our reported warnings represent real bugs, we have no reason to believe that the distribution of reported warnings would be significantly different from the distribution of real faults in the systems. More importantly, we argue that a system developer would have to inspect warnings including false positives (or ignore them altogether), so we can evaluate our hypotheses based on warnings one would report to developers. To reduce bias from false positives from a single analysis, we employ a diverse set of control-flow and data-flow analyses. Finally, our analysis tool is implemented based on TYPECHEF, which is an active research project and, for example, includes only a basic pointer analysis. The analysis of an industry-level tool (e.g., data-flow analysis in the COVERITY tool) might be more precise, but honing our tool to this level is beyond the scope of a research prototype. Also, we could have used existing industry-level tools for our sample-based analysis (likely improving both precision and analysis time), but this would have compromised the comparison (i.e., internal validity) of analysis results between the sample-based and variability-aware analysis.

All analyses are intra-procedural and, thus, consider only the variability that occurs inside methods (although variability may have non-local sources in header files and macros). So, the question is whether a warning's interaction degree is more or less just the interaction degree of the surrounding or whether there is additional variability involved in the warning. To better understand this issue, we show in Figure 10

[22] SQLITE has only one very large file in the amalgamation version.

histograms of the deltas of the interaction degrees of warnings and surrounding methods. In many cases, the interaction degree of a warning corresponds to the interaction degree of the surrounding method (i.e., of the condition that the entire method is included), but in some cases it is higher due to interactions inside a method, which are the most interaction cases. Although we cannot generalize to warnings for inter-procedural analyses, we conjecture that interaction degrees for their warnings are biased to higher interaction degrees, because they are more likely to depend on interactions among multiple code fragments with different conditions.

Regarding sampling, the system variants generated by sampling represent only a small subset of possible variants (which is the idea of sampling). For pair-wise sampling, it may happen that some variants of a file are very similar, as the difference in the respective variant configurations affect the content of a file only to a minor or no extent. However, we argue that our conclusions are still valid, as this lies in the nature of sampling, and all sampling techniques we have used are common in practice. We did not include code-coverage sampling with header files, as it decreases the performance of the sampling analysis to a degree that renders it practically infeasible [41, 42]. Our implementation of code-coverage sampling computes sample sets without taking header-defined variability into account. Therefore, the corresponding sample sets might miss system behavior that depends on variability in the header. However, that some system behavior is not analyzed is a core property of sampling and cannot be avoided.

Finally, a (standard) threat to external validity is that we considered only five subject systems. We argue that this threat is largely compensated by their size and the fact that many different developers and companies contributed to the development of these systems.

5 RELATED WORK

Our implementations of variability-aware static analysis are inspired by earlier work in two fields: variability-aware analysis and configuration sampling.

We and others have developed variability-aware consistency and type checking for academic languages such as Featherweight Java [3, 30], Lightweight Java [20], the Lambda calculus [13], various dialects of Java [30, 56], and for modeling languages [19, 46]. First conceptual sketches reach back over 15 years [6].

Along similar lines, several researchers proposed variability-aware approaches for data-flow analysis. Closest to our work, Brabrand et al. compared three different algorithms for variability-aware, intra-procedural data-flow analysis for Java against a brute-force approach [11]. Similarly, Bodden et al. proposed an approach to extend an existing inter-procedural, data-flow analysis framework to make it variability-aware [10]. Both approaches are limited to an academic setting in which the input Java programs contain `#ifdef`-like variability annotations managed by a research tool. There are no substantial configurable real-world systems that use this technique. Furthermore, both approaches make limiting assumptions on the form of variability (in particular, type uniformity [30] and annotation discipline [29, 40]), which do not hold in real-world software systems [31, 40]. In Bodden's approach, sample configurations are computed for each function separately (sufficient for intra-procedural analyses), whereas we compute sample configurations based on the variability of the entire system. The reason is that our intra-procedural data-flow analyses are based on type information, which is not available in the function and which might depend on configuration options that do not appear in the function.

There is a long history of research on sampling, in particular on variations of combinatorial sampling [12, 36, 50], which is applicable for large input domains beyond compile-time configuration. Nie and Leung provide a survey of the field [48]. Sampling based on code coverage is less explored, as it is tailored primarily to compile-time configuration with the C preprocessor [55]. Since our original conference publication [41], Medeiros et al. [42] investigated the fault-detection capability and size of sample sets for

different sampling heuristics (not covering variability-aware analysis). Their study overlaps with ours with regard to subject systems (e.g., Linux and Busybox); they find that incorporating header files has the potential to detect additional faults (which variability-aware analysis does by construction). Their investigation is based on a much smaller set of confirmed bugs, whereas we focus on all warnings issued by static analyses. Since the authors did not compare to variability-aware analyses, they were able to use state-of-the-art tools for finding the bugs (in particular, Cppcheck[23]). We confirm the authors experience regarding the substantial increase in generation cost and sample-set size when considering the variability model in the generation of pair-wise sample sets. Apel et al. specifically compared sampling strategies against a variability-aware strategy for model checking, but results were limited to small programs [5].

Regarding configuration-related bugs, Garvin and Cohen carefully define the degree of an interaction and identified a few variability-related bugs reported previously in Firefox and gcc [23]; we build on their definition for our measure of interaction degrees. Several studies have analyzed configuration-related bugs found in practice or with sampling strategies regarding their characteristics. For example, Abal et al. analyzed bug fixes of the Linux kernel [1], finding that many bugs are configuration-related, and most analyzed bugs require two configuration options to be selected or deselected (which corresponds to an interaction degree of 2). Medeiros et al. additionally studied how developers introduced variability-related issues in the past, the number of configuration options involved, and the time that these issues remain in the code [42, 44]. These studies tend to report configuration-related issues with mostly low interaction degrees (usually involving, at most, two options), which is consistent with our observations for Busybox, SQLite and uClibc, but not with our observations of Linux and OpenSSL, where we found much higher interaction degrees (see Figure 8). These differences can be explained through differences in methodology: Whereas we used a variability-aware analysis that exhaustively covers all configurations, Garvin and Cohen as well as Abal et al. focused on bugs that have been found, reported, and fixed, and Medeiros et al. used heuristic analyses to detect unused-code issues across many systems, with a heuristic that does ignore header files. The former induces a bias towards code with low interaction degrees, because this code is included in more tested configurations and probably used by more users and testers than code with high interaction degree. The latter may ignore a large amount of variability in common C programs. At the same time, we cannot be sure how many of the our detected warnings represent actual issues, whereas Garvin's, Abal's, and Medeiros' studies consider only confirmed bugs. In that sense, our and their results are not contradictory but complementary.

6 CONCLUSION

In this article, we reported on our experience with the implementation and performance of practical, scalable, variability-aware and sample-based static analyses for real-world, large-scale systems written in C, including preprocessor directives. In a series of experiments on five real-world, large-scale subject systems, we compared the performance of seven variability-aware control-flow and data-flow analyses with the performance of corresponding state-of-the-art sampling techniques (single configuration, pair-wise, and code coverage). In our experiments, we found that the performance of variability-aware analysis scales to large software systems, such as the Linux kernel, and even outperforms some of the sampling techniques, while still being complete with regard to the configuration space. In contrast to most previous work on sampling, we faced many problems and found several limiting factors that render state-of-the-art sampling techniques, such as pair-wise, less effective.

Our experiments show that variability-aware static analysis of a file with variability-aware analysis takes, on average, as much time as analyzing two variants without variability individually (61 variants

[23] http://cppcheck.sourceforge.net/

in Linux). Thus, variability-aware analysis is often slower than single-conf sampling but faster than pair-wise sampling. However, variability-aware analysis reports warnings for all system variants, whereas sampling only selectively checks variants.

To learn about effectiveness, we analyzed the interaction degrees of warnings reported by the different analysis techniques. We found that sampling has a higher miss-rate for warnings of high interaction degree, because the probability of detecting these with a sample configuration are below average. Overall, our evaluation provides further evidence for the efficiency of variability-aware analysis. Furthermore, we show that high interaction degrees occur in intra-procedural control flows and data flows, which are relevant for functional correctness. This fact can be used to focus future work on analysis of configurable-systems to relevant interactions.

ACKNOWLEDGMENTS

This work has been supported by the DFG grants AP 206/4 and AP 206/6, by NSF awards 1318808, 1552944, and 1717022, the Science of Security Lablet (H9823014C0140), and the U.S. Department of Defense through the Systems Engineering Research Center (H9823008D0171).

REFERENCES

[1] I. Abal, C. Brabrand, and A. Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 421–432.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer.

[3] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. 2010. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering* 17, 3 (2010), 251–300.

[4] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. 2013. Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge. In *Proceedings of the International Workshop Feature-Oriented Software Development (FOSD)*. ACM, 1–8.

[5] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. 2013. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 482–491.

[6] L. Aversano, L. Di Penta, and I. Baxter. 2002. Handling Preprocessor-Conditioned Declarations. In *Proceedings of the Working Conference on Source Code Management and Manipulation (SCAM)*. IEEE, 83–92.

[7] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski. 2010. Feature-to-Code Mapping in Two Large Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*. Springer, 498–499.

[8] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. 2010. Variability Modelling in the Real: A Perspective from the Operating Systems Domain. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 73–82.

[9] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640.

[10] E. Bodden, M. Mezini, C. Brabrand, T. Tolêdo, M. Ribeiro, and P. Borba. 2013. SPL$^{\text{LIFT}}$ — Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 355–364.

[11] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. 2012. Intraprocedural Dataflow Analysis for Software Product Lines. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 13–24.

[12] M. Calder and A. Miller. 2006. Feature Interaction Detection by Pairwise Analysis of LTL Properties: A Case Study. *Formal Methods in System Design* 28, 3 (2006), 213–261.

[13] S. Chen, M. Erwig, and E. Walkingshaw. 2012. An Error-Tolerant Type System for Variational Lambda Calculus. In *Proceedings of the International Conference on Functional Programming (ICFP)*. ACM, 29–40.

[14] E. Clarke, O. Grumberg, and D. Peled. 1999. *Model Checking*. The MIT Press.

[15] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and their Application to LTL Model Checking. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1069–1089.

[16] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. 2010. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 335–344.

[17] M. Cordy, P. Heymans, A. Legay, P.-Y. Schobbens, B. Dawagne, and M. Leucker. 2014. Counterexample Guided Abstraction Refinement of Product-line Behavioural Models. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. ACM, 190–201.

[18] K. Czarnecki and M. Antkiewicz. 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. Springer, 422–437.

[19] K. Czarnecki and K. Pietroszek. 2006. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 211–220.

[20] B. Delaware, W. Cook, and D. Batory. 2009. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. ACM, 243–252.

[21] C. Dietrich, R. Tartler, W. Schröder-Preikshat, and D. Lohmann. 2012. Understanding Linux Feature Distribution. In *Proceedings of the International Workshop on Modularity in Systems Software (MISS)*. ACM, 15–20.

[22] M. Erwig and E. Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *ACM Transactions on Software Engineering and Methodology* 21, 1 (2011), 6:1–6:27.

[23] B. Garvin and M. Cohen. 2011. Feature Interaction Faults Revisited: An Exploratory Study. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 90–99.

[24] Y. Hu, E. Merlo, M. Dagenais, and B. Lagüe. 2000. C/C++ Conditional Compilation Analysis using Symbolic Execution. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 196–206.

[25] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Lessenich, M. Becker, and S. Apel. 2016. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering* 2, 21 (2016), 449–482.

[26] M. Johansen, Ø. Haugen, and F. Fleurey. 2011. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 638–652.

[27] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[28] C. Kästner. 2017. Differential Testing for Variational Analyses: Experience from Developing KConfigReader. *CoRR* abs/1706.09357 (2017).

[29] C. Kästner, S. Apel, and M. Kuhlemann. 2008. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 311–320.

[30] C. Kästner, S. Apel, T. Thüm, and G. Saake. 2012. Type Checking Annotation-Based Product Lines. *ACM Transactions on Software Engineering and Methodology* 21, 3 (2012), 1–39.

[31] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 805–824.

[32] C. Kästner, K. Ostermann, and S. Erdweg. 2012. A Variability-Aware Module System. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 773–792.

[33] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. 2012. Toward Variability-Aware Testing. In *Proceedings of the International Workshop Feature-Oriented Software Development (FOSD)*. ACM, 1–8.

[34] G. A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*. ACM, 194–206.

[35] C. Kim, S. Khurshid, and D. Batory. 2012. Shared Execution for Efficiently Testing Product Lines. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 221–230.

[36] D. Kuhn, D. Wallace, and A. Gallo. 2004. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421.

[37] M. Latendresse. 2003. Fast Symbolic Evaluation of C/C++ Preprocessing using Conditional Values. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 170–179.

[38] K. Lauenroth, S. Toehning, and K. Pohl. 2009. Model Checking of Domain Artifacts in Product Line Engineering. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 269–280.

[39] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM,

105–114.
[40] J. Liebig, C. Kästner, and S. Apel. 2011. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 191–202.
[41] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. 2013. Scalable Analysis of Variable Software. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 81–91.
[42] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 643–654.
[43] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. 2015. The Love/Hate Relationship with The C Preprocessor: An Interview Study. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 495–518.
[44] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi. 2015. An Empirical Study on Configuration-related Issues: Investigating Undeclared and Unused Identifiers. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 35–44.
[45] M. Mendonça, A. Wasowski, and K. Czarnecki. 2009. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 231–240.
[46] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval. 2007. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *Proceedings of the International Conference on Requirements Engineering (RE)*. IEEE, 243–253.
[47] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 140–151.
[48] C. Nie and H. Leung. 2011. A Survey of Combinatorial Testing. *Comput. Surveys* 43, 2 (2011), 1–29.
[49] F. Nielson, H. Nielson, and C. Hankin. 1999. *Principles of Program Analysis*. Springer.
[50] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Traon. 2012. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journal* 20, 3-4 (2012), 605–643.
[51] R. Seacord. 2008. *The CERT C Secure Coding Standard*. Pearson.
[52] S. She and T. Berger. 2010. Formal Semantics of the KConfig Language. (2010). Technical Note. Available from http://gsd.uwaterloo.ca/sites/default/files/kconfig_semantics.pdf.
[53] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. 2012. Predicting Performance via Automated Feature-Interaction Detection. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 167–177.
[54] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. 2011. Feature Consistency in Compile-Time Configurable System Software. In *Proceedings of the EuroSys Conference*. ACM, 47–60.
[55] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. 2012. Configuration Coverage in the Analysis of Large-scale System Software. *SIGOPS Operating Systems Review* 45, 3 (2012), 10–14.
[56] S. Thaker, D. Batory, D. Kitchin, and W. Cook. 2007. Safe Composition of Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 95–104.
[57] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (2014), 6:1–6:45.
[58] M. Tomita. 1984. LR Parsers for Natural Languages. In *Proceedings of the International Conference on Computational Linguistics (ACL)*. ACL, 354–357.
[59] A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 178–188.
[60] A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, and S. Apel. 2016. Variability Encoding: From Compile-Time to Load-Time Variability. *Journal of Logical and Algebraic Methods in Programming* 85, 1 (2016), 125–145.
[61] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. 2014. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proceedings of the International Symposium on New Ideas in Programming and Reflections on Software (Onward!)*. ACM, 213–226.

A STATIC ANALYSES

In what follows, we describe the control-flow and data-flow analyses that we use in our study (Section 4) in detail. The analyses *function return* and *double free* are also described in Section 3.1 and Section 3.2, respectively.

Case termination. The conditional statement `switch` consists of an expression, several `case` labels, and an optional `default` label. Each label is followed by a series of statements and should be ended using a `break` statement (by convention), so that the control flow jumps beyond the `switch` statement. The `break` statement is optional. Hence, control flow can *fall through* the next `case` in a `switch` statement executing further statements of the `switch` body. Omitting the `break` statement may lead to unintended control flows and should be avoided. Figure 11a shows a `switch` statement with two `case` blocks. If configuration option `A` is selected, `case 1` contains code without a corresponding `break`, for which our analysis issues an warning.

Dangling Switch. Depending on the condition's value, control flow in a `switch` statement jumps to the body of the `switch`, to one of several `case` labels, to the optional `default` label, or it jumps beyond the `switch` statement. A `switch` body may contain any sequence of statements including declarations of variables. If a programmer places code before the first `case` label, the code is never executed (i.e., it dangles in the `switch` statement). Depending on the kind of code (e.g., the initialization of variables), dangling `switch` code may result in unexpected or undefined behavior. Figure 11c shows an example of dangling `switch` code. If configuration option `A` is not selected, the initialization and the increment of variable `b` (Lines 6 and 7) are not preceded by a `case` statement. Our analysis computes variational CFGs for `switch` bodies and checks whether, apart from declarations without initialization code, any control-flow statements occur that are not preceded by `case` or `default` labels in any configuration.

Function return. According to the C standard, the control flow of all functions with a non-void return type should execute a `return` statement. Using the return value of a function without executing a `return` leads to undefined behavior. Figure 11b shows an example of such a situation. If option A is enabled, function `foo` with return type `int` contains a control-flow path without executing the `return` statement in Line 5. If the condition in Line 3 is not satisfied, the function returns to its call site with an incorrect return value. The result of `foo` is used in `bar`, which may lead to undefined behavior. Our analysis checks whether there is a path in the variational CFG of a function with a non-void return type that does not end in a `return` statement.

Based on Monotone Frameworks, we implemented four variability-aware data-flow analyses. In the following paragraphs, we describe these analyses in detail.

Dead store. Assignments to local variables that are not used in subsequent program code are called *dead store*s. Although dead stores do not particularly threaten program security, they are usually the result of a logical programming error and should be removed. A dead store is a particular instance of *dead code*, the existence of unnecessary code that leads to an increase in program size and an increase in run time. Our example in Figure 11d contains a dead store in Line 4 if configuration option `A` is enabled. The assignment to the variable `a` with a value of `2` is overridden in Line 6 without being used in the meantime. To detect dead stores, we use variability-aware liveness analysis [41] and determine whether assignments to variables „live out“, that is, whether the variables are read in subsequent program code. If a variable is not used in subsequent code, we issue a warning.

Double free. In C, the programmer is responsible for the management of dynamically allocated memory. To this end, the C standard defines a group of standard-library functions for memory allocation (e.g., `malloc`) and memory deallocation (e.g., `free`). To avoid memory leaks, unneeded memory should be freed. Dynamically allocated memory should be freed only once, since freeing memory multiple times leads to undefined behavior, which can be exploited by attackers. Our example in Figure 11f contains a double-free error. If configuration option `A` is not selected, the allocated memory in Line 2 is freed twice (in Lines 7 and 9), because variable `a` in Line 5 (which changes the binding of variables with the name `a`) is not available. To identify double-free errors, we use a variant of reaching definitions. More concretely, our analysis determines whether a pointer variable passed as an argument to the function `free` is passed to another `free` call without any reassignments.

Error handling. High-level constructs for exception or error handling are not available in C. As a result, a common approach is to encode the corresponding information (e.g., the error code) in the form of a return value. For example, for dynamically allocating memory, programmers use the standard library function `malloc`, as in Figure 11e. Given the size of the desired memory chunk, the function returns either the reserved memory as a pointer or zero (`NULL` or `(void*)0`) as return code to indicate that memory allocation has failed. To avoid undefined program behavior, the return value of a call to `malloc` should be checked for this error code before it is used. In our example (cf. Figure 11e), if the configuration option `A` is selected, the pointer `p` is dereferenced before having been checked for the error-return code of `malloc` properly. The C standard defines this error-encoding technique for many standard-library functions, including functions for opening and closing files, for formatting input strings, and many more. To determine missing error-code checks, our analysis tracks the results of 31 standard-library calls with a variant of a variability-aware reaching-definition analysis. In particular, the analysis computes whether variables assigned with the results of standard-library calls reach in control structures, checking them for predefined error codes.

Freeing static memory. Freeing memory that was not allocated dynamically using memory-management functions can result in serious errors (e.g., heap corruption or abnormal program termination). For this reason, only pointer variables that contain memory previously allocated with memory-management functions, such as `malloc` or `realloc`, should be freed with `free`. Our example in Figure 11g contains such an error. Variable `s` in Line 2 is assigned dynamically allocated memory in Line 4 only if configuration option `A` is enabled. As the initialization of `s` with static memory in Line 6 is part of a valid CFG path (if `A` is disabled), passing the variable to `free` in Line 8 is a freeing-static-memory error. Specifically, our analysis tracks pointer variables that are not initialized with dynamic memory, and it determines whether they are passed to memory-management functions.

```
1  void foo(int a) {
2    switch (a) {
3      case 1:
4  #ifdef A
5      a = 2;
6  #endif
7      case 0: a = 1;
8      break;
9    }
10 }
```

(a) Example of a case block without a terminating break.

```
1  int foo(int x) {
2  #ifdef A
3    if (x > 0)
4  #endif
5      return 1;
6  }
7  void bar(int y) {
8    if (foo(y+2)) ...
9  }
```

(b) Example of a control-flow trace without return in a non-void function.

```
1  void f(int a) {
2    switch (a) {
3  #ifdef A
4      case 0:
5  #endif
6        int b=0;
7        b++;
8      case 1:
9        default: a+3;
10   }
11 }
```

(c) Example of dangling switch code.

```
1  void foo() {
2    int x;
3  #ifdef A
4    x = 2; // dead store
5  #endif
6    x = 3;
7  }
```

(d) Example of a dead store.

```
1  void foo() {
2    int* p = malloc(
3         15*sizeof(int));
4  #ifdef A
5    *p = 1;
6  #endif
7    if (p != NULL) {
8        ...
9    }
10 }
```

(e) Example of missing error handling.

```
1  int foo() {
2    int* a = malloc(2);
3    if (a) {
4  #ifdef A
5      int* a = malloc(3);
6  #endif
7      free(a);
8    }
9    free(a); // double free
10   return 0;
11 }
```

(f) Example of a double free.

```
1  int foo(int l) {
2    char *s;
3  #ifdef A
4      s = (char *)malloc(12);
5  #else
6      s = "usage: ... ";
7  #endif
8    free(s); // freeing st. mem.
9    return 0;
10 }
```

(g) Example of freeing memory that was allocated statically.

Fig. 11. Examples for defects that can be detected with our static analyses.