Automated Software Engineering (2022) 29:35
https://doi.org/10.1007/s10515-022-00333-1

Static data-flow analysis for software product lines in C

Revoking the preprocessor's special role

Philipp Dominik Schubert[1] · Paul Gazzillo[2] · Zach Patterson[3] · Julian Braha[2] · Fabian Schiebel[4] · Ben Hermann[5] · Shiyi Wei[3] · Eric Bodden[1,4]

Received: 27 July 2021 / Accepted: 15 February 2022 / Published online: 25 March 2022

Abstract
Many critical codebases are written in C, and most of them use preprocessor directives to encode variability, effectively encoding software product lines. These preprocessor directives, however, challenge any static code analysis. SPLlift, a previously presented approach for analyzing software product lines, is limited to Java programs that use a rather simple feature encoding and to analysis problems with a finite and ideally small domain. Other approaches that allow the analysis of real-world C software product lines use special-purpose analyses, preventing the reuse of existing analysis infrastructures and ignoring the progress made by the static analysis community. This work presents VarAlyzer, a novel static analysis approach for software product lines. VarAlyzer first transforms preprocessor constructs to plain C while preserving their variability and semantics. It then solves any given distributive analysis problem on transformed product lines in a variability-aware manner. VarAlyzer 's analysis results are annotated with feature constraints that encode in which configurations each result holds. Our experiments with 95 compilation units of OpenSSL show that applying VarAlyzer enables one to conduct inter-procedural, flow-, field- and context-sensitive data-flow analyses on entire product lines for the first time, outperforming the product-based approach for highly-configurable systems.

Keywords Inter-procedural static analysis · Software product lines · Preprocessor · LLVM · C/C++

✉ Philipp Dominik Schubert
philipp.schubert@upb.de

Extended author information available on the last page of the article

1 Introduction

Software product lines (SPLs) enable software developers to encode a set of software products in a common code base. The different variations, so-called configurations, are typically described with the help of static conditionals, so-called *features*, that enable conditional compilation. In the programming languages C and C++, developers typically use the preprocessor's functionalities, particularly the well-known `#ifdef` directives, to establish SPLs. The preprocessor's static conditionals allow developers to check the presence of a symbol or its value—an integer or a string literal. At compile time, the preprocessor transforms every compilation unit according to the given set of symbols (and their respective values), before the preprocessed compilation unit is handed over to the actual compiler. The compiler thus only compiles the code that has been included by the preprocessor, which allows it to produce efficient object code. This also means that in the worst case an SPL induces a number of software products that is exponential in the number of static conditionals.

Static data-flow analysis is not only used as a basis for compiler optimizations (GCC-Optimize-Options 2018; ICCOptimizeOptions 2018), but also for bug finding (Coverity-(SAST) 2018; CodeSonar 2018) and software hardening (Arzt et al. 2014; Krüger et al. 2017; Livshits and Lam 2005; Hermann et al. 2015; Holzinger et al. 2017). However, previous software vulnerabilities such as Apple's FileVault vulnerability (FileVaultBug 2012) show that program analysis of configurable systems is crucial. The FileVault vulnerability was caused by accidentally shipping a Mac OS X version with logging code enabled that stored the user login passwords in clear text. Such a vulnerability might have been detected early, had Apple had the capability to analyze FileVault's codebase with respect to all possible configurations.

The problem with traditional static analysis techniques, however, is that they cannot be applied to software product lines directly. Instead, one must first generate a concrete software product by preprocessing the common code base and then analyze the resulting plain C/C++ program. Due to the possibly exponential number of software products in practice, this process becomes prohibitively expensive even when analyzing only a few variants, let alone all possible software products.

SPLlift (Bodden et al. 2013) was proposed to analyze an entire SPL as a whole, a so-called family-based approach (Thüm and Apel 2012), which avoids generating all potential software products. While doing so, it avoids an exponential blowup through a time and memory efficient encoding of feature constraints in distributive flow functions. However, SPLlift is restricted to *Interprocedural Finite Distributive Subset (IFDS)* (Reps et al. 1995) problems, which include simple problems such as taint analysis, but exclude problems with large or potentially infinite domains such as constant propagation (Sagiv et al. 1996) or typestate analysis (Strom 1983; Strom and Yemini 1986). More importantly, it is a prototype for a seldom-used product-line dialect of Java (Kästner et al. 2009) and thus cannot be applied to real-world SPLs, particularly not those that use the C preprocessor.

Existing techniques that are able to analyze real-world SPLs written in C operate on un-preprocessed C code and include new or modified algorithms for

parsing (Kästner et al. 2011; Gazzillo and Grimm 2012; Garrido and Johnson 2005), data-flow analysis (Liebig et al. 2013; Rhein et al. 2018), type checking (Kästner et al. 2012), and rewriting (Iosif-Lazar et al. 2017). The only available data-flow analysis (Liebig et al. 2013; Rhein et al. 2018), however, is intra-procedural only. In addition, all those techniques are special-purpose analyses, making it infeasible to reuse existing state-of-the-art static analysis infrastructures. The situation becomes even more complicated when looking at the long term. While the research on "variability-oblivious" program analysis marches on, those variability-aware toolchains must be maintained in parallel, doubling the engineering effort, which explains why none of the above approaches has been maintained in the long term. Other works proposed new preprocessors (McCloskey and Brewer 2005; Kästner 2010). Language adoption, however, is a notoriously slow development. And even *if* those new preprocessors get adopted over time, one cannot expect that millions of lines of existing legacy code will be rewritten. Despite C's known issues, it is the most popular programming language according to the TIOBE programming index.[1]

In this work, we present the design and implementation of VarAlyzer, a novel static data-flow analysis approach built on top of SuperC (Gazzillo and Grimm 2012) and PhASAR (Schubert et al. 2019). The idea is to revoke the preprocessor's special role by first transforming preprocessor directives into ordinary C code. Preprocessor conditionals are replaced with C conditionals, preprocessor macros are replaced with C variables, and the existence of declarations is controlled via C expressions that use these declarations. The transformation uses a configuration-aware type checker which supports static behaviors at runtime that could not be implemented before, e.g. type errors caused by infeasible configurations are expressed as runtime calls to an error function. VarAlyzer allows one to automatically make any existing (or new) distributive data-flow analysis on real-world C software product lines *variability-aware* which it then solves in a single analysis run on the transformed software product line.

On top, and in contrast to SPLlift, VarAlyzer supports not just analyses encoded in IFDS (Reps et al. 1995) but also in *Interprocedural Distributive Environments* (IDE) (Sagiv et al. 1996), which includes problems with infinite domains. As a result, VarAlyzer outputs the fully context- and flow-sensitive data-flow facts along with a feature constraint describing the product configurations for which they hold. This allows developers to find bugs and vulnerabilities much earlier in the development process, requiring no product to be generated. Whereas previously developers of highly-configurable software had to identify vulnerabilities separately for each concretely preprocessed variant, using VarAlyzer they can exclude such vulnerabilities in all relevant configurations ahead of time.

We evaluate VarAlyzer 's effectiveness by conducting a typestate analysis (Strom 1983; Strom and Yemini 1986) that checks for the correct usages of OpenSSL's Envelope (EVP) APIs on 95 compilation units. Typestate analysis belongs to an important class of analyses whose efficient encoding, due to the internal state,

[1] As of March, 2021, TIOBE programming index https://www.tiobe.com/tiobe-index/.

requires IDE (Sagiv et al. 1996) or equally expressive frameworks such as weighted pushdown systems (Reps et al. 2003). The IFDS-based SPLlift approach thus could not solve such an analysis on realistic programs. The (hand-)written compilation units in C comprise realistic uses of EVP's APIs for message digest (MD), encryption/decryption (CIPHER), and message authentication codes (MAC). The compilation units, ranging from 8 to 219 lines of code, comprise preprocessor conditionals and valid as well as invalid API usages. For this work, we have to restrict ourselves to evaluating our approach on individual compilation units because several fundamental challenges that are beyond the scope of this paper currently prevent us from evaluating VarAlyzer on full SPLs. Large-scale projects not only encode variability in the preprocessor but also in other parts of the system software toolchain. To support full SPLs, an approach would additionally need to solve the difficult problem of supporting variability-aware linking and build automation. VarAlyzer provides full support for application configurations. However, system configuration macros provide yet another challenge. Not only would an approach need to support platform-dependent header file differences, but would also require one to construct a superset of all C variations. We detail on these challenges in Sect. 4.1.4.

We will make the implementation of VarAlyzer available as open source. We will subject it to artifact evaluation and make it available under the permissive MIT license. All accompanying artifacts of this paper, including processed analysis targets and result data, are available as supplemental material (Artifacts 2021).

In summary, this paper makes the following contributions:

- A novel end-to-end variability-aware static analysis approach that enables variational analysis of C software product lines. The approach transforms software product lines to ordinary C code while preserving the complete preprocessor semantics and performs an automated lifting that allows one to solve *arbitrary* distributive data-flow problems in a *variability-aware* manner.
- An open-source implementation based on SuperC (Gazzillo and Grimm 2012) and PhASAR (Schubert et al. 2019).
- An experimental evaluation of VarAlyzer, which assesses its effectiveness in solving general IDE (Sagiv et al. 1996) problems on 95 compilation units that use OpenSSL.
- An assessment of the further challenges that need to be overcome to make static analysis of arbitrary C applications a reality.

2 Motivating example

To motivate the need for variability-aware analyses, we show an example using typestate analysis on a software product line. Most APIs are required to be called in a particular order or pattern. The valid sequences of operations can be encoded using state machines. A typestate analysis (Strom 1983; Strom and Yemini 1986) or protocol analysis is a static analysis that tracks variables of a certain type and their associated states through the program. Typestates define sequences of operations that may be performed upon a variable. The state information associated

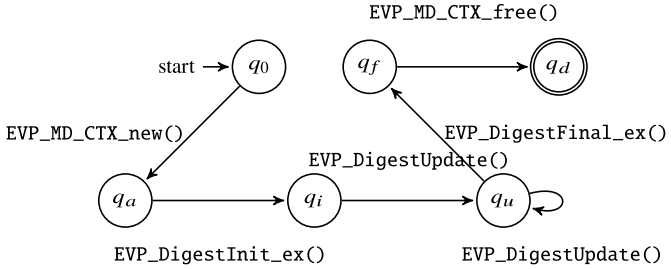

Fig. 1 State machine that describes the correct usages of the OpenSSL EVP message digest (MD) API

with each variable is used to determine—at compile-time—the validity of operations invoked upon variables. Existing analysis techniques for SPLs that rely on special-purpose analyses formulated for variability-preserving ASTs cannot solve this problem class.

The state machine shown in Fig. 1 describes the valid usages of OpenSSL's EVP message digest (MD) API. An SPL that performs a message digest using OpenSSL's EVP message digest (MD) API is shown Listing 1. The SPL comprises a debugging feature encoded with the symbol `DEBUG`. When this symbol is enabled in the preprocessor, and therefore debugging is enabled at runtime, MD's API protocol is violated as the call to `EVP_DigestFinal_ex()` at line 21 is omitted—a potential security threat. Even variability-aware intra-procedural data-flow analysis cannot properly solve this analysis problem in our example program because the variable `MDCTX` that carries the state information is processed across multiple different functions.

Traditional techniques would first generate a particular variant (and all variants we are interested in, possibly all of them) of the SPL, and then uncover this problem in a static analysis of that particular variant. A brief inspection of our example SPL using the GCC compiler shows that it comprises 6,946 preprocessor macros and (transitively) includes 221 different header files. 261 of those 6,946 macros are used in preprocessor conditionals. Therefore, traditional analysis techniques can not scale. Instead, it is desirable to analyze all potential configurations, i.e., feature combinations, at the same time. By transforming the preprocessor directives into ordinary C code, our approach allows to employ any existing C analysis tools to analyze the entire SPL as a whole. PhASAR 's traditional typestate analysis, for instance, would be able to detect the protocol breach caused by the missing call to `EVP_DigestFinal_ex()`. In more complex scenarios, however, it would also report a large number of false positives because the results are valid across all configurations, making any findings virtually impossible to debug. Traditional analysis would need to merge information at control-flow merge points even for branches that originate from static preprocessor conditionals, which is impossible in practice. Therefore, it is desirable to have an analysis that can handle preprocessor variability to produce results that are actually useful to the analysis users.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 #include <openssl/crypto.h>
5 #include <openssl/evp.h>
6
7 void digestMessage(EVP_MD_CTX *MDCTX,
8                    const unsigned char *Msg,
9                    size_t MsgLen,
10                   unsigned char **Dgst,
11                   unsigned int *DgstLen) {
12   EVP_DigestInit_ex(MDCTX, EVP_sha256(), NULL);
13   EVP_DigestUpdate(MDCTX, Msg, MsgLen);
14 #ifdef DEBUG
15   const char DebugHash[] = "Hello, Hash!";
16   *DgstLen = sizeof(DebugHash);
17   *Dgst = OPENSSL_malloc(sizeof(DgstLen));
18   strncpy((char *)*Dgst, DebugHash, *DgstLen);
19 #else
20   *Dgst = OPENSSL_malloc(EVP_MD_size(EVP_sha256()));
21   EVP_DigestFinal_ex(MDCTX, *Dgst, DgstLen);
22 #endif
23 }
24
25 int main() {
26   const char *Data = "My secret data.";
27   unsigned char *Dgst;
28   unsigned int DgstLen;
29   EVP_MD_CTX *MDCTX = EVP_MD_CTX_new();
30   digestMessage(MDCTX,
31                 (const unsigned char*) Data,
32                 strlen(Data),
33                 &Dgst,
34                 &DgstLen);
35 #ifdef DEBUG
36   printf("hashed data: %s\n", Dgst);
37 #endif
38   EVP_MD_CTX_free(MDCTX);
39   OPENSSL_free(Dgst);
40   return 0;
41 }
```

Listing 1: An SPL using OpenSSL's EVP message digest (MD) functionalities. Error handling code has been omitted for brevity.

3 Background on IFDS and IDE

In the following, we present the conceptual *Interprocedural Finite Distributive Subset (IFDS)* (Reps et al. 1995) framework and its generalization, the *Interprocedural Distributive Environments (IDE)* (Sagiv et al. 1996) framework. Both frameworks support the efficient, summary-based solving of distributive (Bodden 2018) data-flow problems. We will later use the IDE framework to encode any given distributive data-flow problem and solve it in a variability-aware manner.

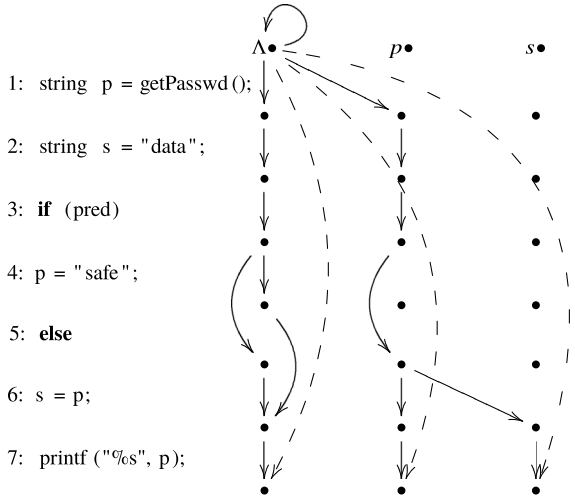

Fig. 2 An exemplary exploded super-graph for a taint analysis encoded in IFDS (Reps et al. 1995). Individual flow functions are indicated with solid edges (→) and flow function summaries (also known as jump functions) are indicated with dashed edges (⇢)

Both IFDS and IDE solve a data-flow problem by constructing an exploded super-graph (ESG). By construction, a data-flow fact d holds at statement s, if a node (s, d) in the ESG is reachable from a special, tautological node Λ. The ESG is constructed for a given program by replacing every node of its inter-procedural control-flow graph (ICFG) (oftentimes also referred to as *supergraph*) with the bipartite graph representation of the respective flow function. Every flow function that is distributive can be represented as a bipartite graph without loss of precision. The common flow functions *identity*, *generate* (Gen), and *remove* (Kill) are distributive and thus, all *Gen/Kill* data-flow problems can be encoded within IFDS and IDE.

An exemplary ESG for a taint analysis encoded in IFDS that showcases how bipartite graphs can be used to represent flow functions is shown in Fig. 2. A taint analysis tracks *tainted* variables generated by so-called *source* functions through the program and reports potential security vulnerabilities whenever a tainted variable reaches a call to a *sink* function. The function `getPasswd()` acts as a *source* in our example as it retrieves sensitive user information and the `print()` function presents a *sink* as sensitive information must not leak. The taint analysis detects the potential leak at line 7 in the program since the ESG node (stmt:7, p) is reachable by the tautological Λ fact.

To achieve fully context-sensitive, inter-procedural analysis, IFDS and IDE follow the summary-based approach (Sharir and Pnueli 1978), creating procedure summaries that can be reused and instantiated in multiple calling contexts. Summaries are created by composing the flow functions of adjacent statements. The composition $h = g \circ f$ of two flow functions f and g, called *jump function*, can be obtained by combining their bipartite graph representations. The graph of h can be produced by merging the nodes of g with the corresponding nodes of the domain of f. Once a summary ψ for a complete procedure p has been constructed, it can be re-applied in any other contexts in which the procedure p is called. Jump functions are indicated using dashed arrows in Fig. 2.

In IDE, the ESG edges carry additional distributive functions. Those *edge functions* can be used to describe an additional *value computation problem* over a value domain V that is solved while performing the reachability check in the ESG. The runtime complexity of both algorithms is $\mathcal{O}(|N| \cdot |D|^3)$, where N is the set of program statements and D is the data-flow domain, i.e., the set of data-flow facts. Importantly, the complexity is independent of $|V|$, which allows IDE to conduct efficient computations even using large or even infinite value domains (e.g., sets of states of larger state machines in a typestate analysis or the set of natural numbers as is required in constant propagation). Attempts to encode such problems in IFDS will lead to state-space explosion or even non-termination. While one can generally encode a linear constant propagation in IFDS using $D = (v, c)$, where $v \in \mathcal{V}$ is the set of program variables and $c \in \mathbb{Z}$, i.e., with tuples of program variables and associated integer values, this encoding drastically impedes performance. This is because IFDS was built to solve problems with finite domains but $\mathbb{Z}$ is infinite. Even in cases where one bounds its size artificially, solving performance will be bad. A linear constant propagation can be encoded much more efficiently instead in IDE, using $D = \mathcal{V}$ and $V = \mathbb{Z}$, such as to reduce the size of the data-flow domain and to utilize the edge functions' value domain V—computing a variable's value using the context-independent edge functions. Since the complexity of IDE's solving algorithm depends only on the size of D and not V and therefore is independent of the infinite size of $\mathbb{Z}$, such an encoding will scale (Sagiv et al. 1996). An exemplary ESG for a linear constant propagation encoded in IDE is shown in Fig. 7. We explain this ESG in detail in Sect. 4.2.

4 VarAlyzer

In this section, we detail our approach to statically analyzing C software product lines. VarAlyzer consists of two phases. First, it transforms software product lines into an intermediate representation (IR). Second, it applies a novel data-flow solver that enables *variational* analysis of arbitrary distributive analysis problems and produces precise results for all variants of a software product line in a single analysis run.

4.1 Transforming preprocessor directives

The main idea of VarAlyzer 's transformation is that the static preprocessor conditionals are automatically replaced with runtime C conditionals. The key challenge is that preprocessor conditionals may appear around any arbitrary set of C tokens, irrespective of C's syntax (Ernst et al. 2002; Liebig et al. 2011), while C conditionals may only appear around complete statements. For instance, in Fig. 3a, preprocessor conditionals appear around a declaration (lines 2–3) and a function definition (lines 6–7) of the same name. While the preprocessor technically has a language distinct from pure C, we take the view that unpreprocessed C files are effectively written in a single, mixed language. To preserve the encoding of variability in

```
1  #ifdef M1
2  extern void
3    f(int x);
4  #endif
5  #ifdef M2
6  static void
7    f(int x) { }
8  #endif
9  void g() {
10   f(10);
11 }
```

(a) Before

```
1  const bool M1, M2;
2  extern void
3    __f_1(int x);
4  static void
5    __f_2(int x) { }
6  void g() {
7   if(M1 && !M2) __f_1(10);
8   if(M2 && !M1) __f_2(10);
9   if(M1 && M2||!M1 && !M2)
10     __type_error();
11 }
```

(b) After

Fig. 3 Desugaring a variational function definition, adapted from Linux v4.18 kernel/sched/sched.h

```
1  #ifdef MACRO
2    if (cond)
3      i = 31;
4    else
5  #endif
6      i = -32;
```

(a) Before

```
1  const bool MACRO;
2  if (MACRO) {
3    if (cond)
4      i = 31;
5    else
6      i = -32;
7  } else
8    i = -32;
```

(b) After

Fig. 4 Desugaring a variational if statement, adapted from Linux v2.6.33.3 drivers/input/mousdev.c

unpreprocessed C, VarAlyzer *desugars* source files into ordinary C, which is a subset of the mixed language.

The preprocessor performs macro evaluation, header inclusion, and conditional compilation to generate C code at compile-time. With conditional compilation, the preprocessor selects which parts of the source code to send to the compiler by evaluating the values of configuration macros passed into the preprocessor at compile-time. Developers use these preprocessor conditionals to encode variability.

Developers may wrap these conditionals around any fragment of the C code. Common patterns in real-world code include putting conditionals around entire functions, declarations, and even individual C tokens. Since preprocessing happens before parsing in the compiler, these conditionals do not need to respect C's syntax. Developers may even wrap them around incomplete C constructs, so-called "undisciplined" uses (Liebig et al. 2011). Figure 4a is an example of this usage, where a preprocessor conditional surrounds all but the else-branch body of an if-then-else statement (lines 2–4).

Since our goal is to preserve the behavior of these preprocessor conditionals, we need to consider their meaning when they interact with C constructs. While a preprocessor conditional has simple semantics (i.e., it conditionally includes or excludes the contained C fragment), its effect on C program behavior depends on what C constructs it surrounds. For instance, it is illegal in C's semantics to write multiple declarations of the same variable to vary its type. By surrounding these

declarations with mutually-exclusive preprocessor conditionals, it is "legalized": the preprocessor only chooses one declaration to send to the C compiler. The only way to allow such multiple declarations in C is to use unique variable names. In contrast, a preprocessor conditional around a C statement behaves much like a C conditional, except that the preprocessor does not respect C's scoping rules and it takes configuration macros instead of C variables.

4.1.1 Phases of the desugarer

VarAlyzer takes unpreprocessed C code, such as that in Fig. 4a, and produces an equivalent C program using run-time conditionals to preserve variability (Fig. 4b). There are three phases in VarAlyzer 's desugarer: (1) parsing, (2) type checking, and (3) rewriting. Parsing takes the unpreprocessed C code and produces an AST that preserves all preprocessor behavior. Type checking collects symbols and their types across all variations of the SPL. Rewriting emits ordinary C code that corresponds to the unpreprocessed C constructs.

Parsing For parsing, we reuse an existing parser, SuperC (Gazzillo and Grimm 2012). Unlike the standard C preprocessor and parser, SuperC solves the problem of parsing all variations of a C file. It provides a complete solution to parsing C syntax even when mixed with any combinations of preprocessor usage. Eschewing incomplete heuristics, SuperC 's parsing formalism enables comprehensive parsing of unpreprocessed C, supporting complicated and even pathological cases, such as conditionally-defined macros and headers, macros with incomplete C syntax, stringification and token-pasting combined with ifdefs, and more. Listing 2 and Listing 3 present two more complex examples that use a combination of some of these features. The specifics of this parser can be found in Gazzillo and Grimm (2012). An overview of the possible interactions between the C preprocessor and C's language features is shown in Table 1. SuperC 's output is a C AST that has special "static conditional" nodes that capture every possible variation of the syntax of the input source file. The parsing algorithm ensures that conditional nodes are guaranteed to appear around complete C syntactic units, even when the unpreprocessed input file does not, by duplicating any tokens needed to comphrehensively represent all variations of the nearest ancestor construct. For instance, Fig. 4a's AST will have a static conditional node with two branches, one for `MACRO` and the other for `!MACRO`. The former branch will contain a complete if-then-else statement with no other static conditionals inside and the latter will have a single assignment statement.

Table 1 Interactions between C preprocessor and language features. Reproduced from Gazzillo and Grimm (2012)

Language Construct	Implementation	Surrounded by Conditionals	Contain Conditionals	Contain Multiply-Defined Macros	Other
Lexer					
Layout	Annotate tokens				
Preprocessor					
Macro (Un)Definition	Use conditional macro table	Add multiple entries to macro table		Do not expand until invocation	Trim infeasible entries on redefinition
Object-Like Macro Invocations	Expand all definitions	Ignore infeasible definitions		Expand nested macros	Get ground truth for built-ins from compiler
Function-Like Macro Invocations	Expand all definitions	Ignore infeasible definitions	Host conditionals around invocations	Expand nested macros	Support differing argument numbers and variadics
Token Pasting & Stringification	Apply pasting & stringification		Host conditionals around token pasting & stringification		
File Includes	Include and preprocess files	Preprocess under presence conditions		Hoist conditionals around includes	Reinclude when guard macro is not false
Static Conditionals	Preprocess all branches	Conjoin presence conditions		Ignore infeasible definitions	
Conditional Expressions	Evaluate presence conditions			Hoist conditionals around expressions	Preserve order for non-boolean expressions
Error Directives	Ignore erroneous branches				
Line, Warning, Pragma Directives	Treat as layout				
Parser					
C Constructs	Use FMLR Parser	Fork and merge subparsers			
Typedef Names	Use conditional symbol table	Add multiple entries to symbol table			Fork subparsers on ambiguous names

```
#include <assert.h>
#include <stdio.h>

#if DEBUG && WARN_LEVEL == 0
#define WARN_IF(EXP) \
  do {               \
    assert(!(EXP));  \
  } while (0)
#elif DEBUG && WARN_LEVEL > 0
#define WARN_IF(EXP)                                \
  do {                                              \
    if (EXP)                                        \
      fprintf(stderr, "Warning: " #EXP "\n"); \
  } while (0)
#else
#define WARN_IF(EXP) while (0)
#endif

int main() {
  int x = 2;
  WARN_IF(x == 2);
  return x;
}
```

Listing 2: A combination of stringification (or stringizing) of expressions, (function-like) macros and ifdefs. Modified example reproduced from [37].

```
#include <stdio.h>
#include <string.h>

// contents of state.def
// #ifndef STATE_SELECT
// #define STATE_SELECT(NAME, VALUE)
// #endif
// STATE_SELECT("First", First)
// STATE_SELECT("Second", Second)
// STATE_SELECT("Third", Third)
// STATE_SELECT("Fourth", Fourth)
// #undef STATE_SELECT

enum state {
#define STATE_SELECT(NAME, VALUE) VALUE,
#include "state.def"
  Invalid
};

enum state strToState(const char *str) {
#define STATE_SELECT(NAME, VALUE)                \
  if (strncmp(str, NAME, strlen(NAME)) == 0) { \
    return VALUE;                                \
  }
#include "state.def"
  return Invalid;
}

char *stateToStr(enum state s) {
  switch (s) {
#define STATE_SELECT(NAME, VALUE) \
  case VALUE:                     \
    return NAME;                  \
  break;
#include "state.def"
  case Invalid:
    return "invalid state!";
  break;
  }
}

int main() {
  enum state s = strToState("First");
  printf("%s\n", stateToStr(s));
  printf("%s\n", stateToStr(Second));
  return 0;
}
```

Listing 3: Code generation using the preprocessor as often used in C++ to preserve type safety when dealing with enumerations.

Type checking Traditionally, type checking ensures the absence of type errors at runtime. VarAlyzer, however, relies on the type checking phase to enable desugaring. To emit C code equivalent to the unpreprocessed C, the desugarer needs to know what variables have been declared (or left undeclared) in all the variations of

the source code. As with typical C type checking, we maintain *a symbol table* and apply *C type checking rules with semantic actions* during parsing. A symbol's entry in the table, however, depends on what variation we are analyzing. For instance, in Fig. 3a, declarations of `f` (lines 3 and 7) have incompatible type qualifiers (`extern` and `static`). However, these two declarations can never appear in the same variation. VarAlyzer 's type checker needs to track both types throughout the source file.

The symbol table binds a symbol to all of its possible types across all variations of the source code. The binding is akin to a "variational set" (Walkingshaw et al. 2014), where each type element is tagged with configuration information. The set also includes special entries to record the conditions under which the symbol is undeclared or has a type error in its declaration. This is necessary because a typical type checker will use the absence of a binding to mean undeclared and will simply halt on a type error. When desugaring, only some of the variations of the source file may have an undeclared symbol or other type errors. We continue to desugar any valid configurations instead of halting. Our type checker, in effect, tracks types in all variations of the source code simultaneously.

For instance, the symbol table entry for `f` in Fig. 3a contains a set with four elements, one for each possible variation of this source code. `f` is undeclared if *both* `M1` and `M2` are undefined. `f` is a redeclaration type error if both `M1` and `M2` are *defined*. There are two more entries for the valid type declarations, which happen when only one of `M1` and `M2` is defined, but not both. The resulting symbol table entry for `f` is as follows:

$$\texttt{f} \mapsto \begin{cases} \texttt{extern void} & \textbf{if } \texttt{M1} \wedge \neg\texttt{M2} \\ \texttt{static void} & \textbf{if } \neg\texttt{M1} \wedge \texttt{M2} \\ \texttt{<ERROR>} & \textbf{if } \texttt{M1} \wedge \texttt{M2} \\ \texttt{<UNDECLARED>} & \textbf{if } \neg\texttt{M1} \wedge \neg\texttt{M2} \end{cases}$$

Rewriting The rewriting phase produces ordinary C code that preserves the behavior of the unpreprocessed source file. The underlying parser of VarAlyzer ensures static conditionals are lifted around only complete C syntax, i.e., syntactic lifting, but our rewriter still needs to consider the behavior of static conditionals on those C constructs. When a static condition surrounds a construct, VarAlyzer lifts the construct's semantic value to the nearest ancestor that is a statement, declaration, or function definition, if not already one of these. This step ensures that VarAlyzer will output valid C code by only inserting C conditional around complete statements.

The rewriting rules depend on what C construct a preprocessor conditional surrounds: statements, declarations, etc. In general, statements are surrounded by a C conditional and configuration macros are transformed to C constant variables. Figure 4b shows the result of desugaring Fig. 4a. Recall that the parser ensures that the static conditionals appear around a complete if-then-else statement and a complete assignment statement. The desugarer declares a new C constant called `MACRO` on line 1, and then emits a C conditional that uses this variable around the two complete C constructs. Notice that any tokens shared by the two complete constructs are duplicated under the C conditional, which provides guarantees of "disciplined" uses of conditionals.

```
1  struct {
2    int x;
3  #ifdef MACRO
4    int y;
5  #endif
6  } var;
7  var.y;
```

(a) Before

```
1  const bool MACRO;
2  struct {
3    int x;
4    int y;
5  } var;
6  if (MACRO) {
7    var.y;
8  } else {
9    __type_error();
10 }
```

(b) After

Fig. 5 Desugaring variational if statement. Adapted from Linux v2.6.33.3 drivers/input/mousdev.c

Declarations and function definitions cannot be desugared by surrounding them with a C conditional, since they are not statements. VarAlyzer handles multiply-declared symbols by emitting all declarations unconditionally, resolving name clashes by renaming the symbols. VarAlyzer preserves variability at runtime by instead emitting C conditionals *where the symbols are used in statements*. In Fig. 3b, VarAlyzer creates fresh identifiers for the two declarations (lines 3 and 5). The *usage* of the symbol `f` is replaced with a C conditional (lines 7–8) and the mutual exclusion of the two declarations in different configurations is preserved in lines 9–10.

The type checking phase is instrumental in VarAlyzer. It records all variations of the original symbol, which enables VarAlyzer to assign a fresh name to each of the variational set's entries, e.g., `__f_1` and `__f_2` in Fig. 3b. In addition, the type checker records which configurations have type errors. Type errors are normally emitted at compile-time. VarAlyzer, however, cannot halt with such errors when only some variations have them. Instead, it preserves type errors as runtime errors, by transforming them into calls to a specially-defined `__type_error` function that always halts. In Fig. 3b, VarAlyzer preserves the type error with line 10, reflecting the fact that there is no declaration of `f` when macros `M1` and `M2` are both undefined and a conflicting declaration if both macros are defined. The subsequent analysis can then rule out invalid configurations as unreachable code, avoiding the imprecision by analyzing these configurations.

4.1.2 Desugaring C type specifications

While duplicating multiply-declared symbols is sufficient for variables and functions, C also supports user-defined types via typedefs, structs, unions, and enums. The latter three can also appear within declarations. The declaration in Fig. 5a declares `var` to be a new type `struct s`. Structs and unions contain field declarations which themselves may contain struct and union definitions. A naive desugaring could take all combinations of struct/union definitions and emit each one as a separate declaration in the output C program. Real-world SPLs, however, may have highly-configurable structs, where some fields only appear in certain variations.

Struct fields may also be declared using highly-configurable structs, further exploding the possible combinations of declarations.

In addition, C allows forward references to type definitions under certain conditions, which originally made one-pass compilation possible. A struct, for instance, may be referenced in a declaration of a variable before the struct itself is defined, at least in the global scope. Handling forward references would require multiple passes of the AST, making a complete desugaring not possible in a single pass.

To solve these problems, VarAlyzer handles type definitions separately from variable and function declarations. In addition to storing type declarations in the symbol table, as with typical C type checking, we maintain a separate table for struct, union, and enum type definitions. This table collects all possible field variations (or enumerators) for each type definition, regardless of where in the scope they are defined. As with the symbol table, we are tagging each field definition with a logic formula describing which variations contain the particular field. Then, before emitting the desugared contents of each static scope, we emit a single declaration of the struct, union, or enum containing all possible fields or enumerators. When a struct variable accesses its field, we emit runtime checks for type errors.

For instance, in Fig. 5b, the resulting desugared struct definition contains both the `x` and `y` fields, because there is no language construct in pure C for defining conditionally-defined structs. But `y` is only meant to be defined under configurations that have `MACRO` enabled. Since the desugarer's struct symbol table tracks the configurations under which each field is defined, the desugarer accounts for the configuration where fields are *accessed*, rather than where they are defined. For example, in Fig. 5b the desugarer has transformed the access of field `y` to a C conditional (lines 6–10) that covers both possible variations of the struct. The first branch of this conditional covers configurations where `MACRO` is enabled and therefore the field `y` exists (line 7). The else branch accounts for all other configurations, where accesses to `y` are type errors, since the field is not defined those configurations. The desugarer preserves this type error as a run-time error with a call to a specially-defined function on line 9.

Forward references to structs, unions, and enums require further special handling in order to desugar in a single pass. Since VarAlyzer does not know yet what all fields or enumerators of the type will be, it instead emits a fresh type name for the forward reference. Once it has collected all fields for a given type at the end of the static scope, it emits a definition of the fresh forward reference type that contains a field for each definition of the type symbol.

4.1.3 Desugaring function definitions

C function definitions combine a type declaration of the function name with a compound statement for its body, so VarAlyzer needs to both preserve all variations of the function in its symbol table and emit all variations of the function's body. VarAlyzer uses its variation-preserving symbol table to hold function symbols, while the function body is transformed like any other compound statement using C conditionals to preserve variations in statements.

As with declarations, a function with multiple variations of its type is desugared into multiple function definitions to reflect each variation. Any calls to the original function name are replaced by all renamed variations of the function, as long as the function type matches the type at the call site. All top-level declaration and definitions in a C file are global and externally-linked by default, unless specified otherwise with the `static` keyword. Therefore, any renaming at the global scope affects the symbols exported for linking by the compiler. Since C does not provide language constructs for defining modules, it relies on the underlying system's object file format, linker, and build system to coordinate interfaces between C source files. In this work, we focus on desugaring variability encoded by the preprocessor within a C file and leave the support for build system and linker variability as future work.

Instead, we assume a project only exports one type per global symbol, emitting a type warning when a global symbol has multiple, incompatible type declarations. Each C file that uses functions defined externally needs a copy of the external functions' declarations, typically provided in a shared header file that developers copy into the source file using a preprocessor `#include` directive. It is then up to the compiler to produce an object file with a linker table that maps global functions and variables to either their addresses in the object file or to a placeholder. The linker can then automatically match undefined symbols from one object file with its definition in another, as long as the developer has properly defined the build sequence with, for instance, a Makefile.

If a globally-defined symbol's declaration depends on what variation of the program is being compiled, i.e., it is affected by preprocessor conditionals, then preserving all variations of the SPL requires modeling the behavior of the linker across all variations. Such a variation-preserving linker would need to record all renamings of multiply-declared global symbols and resolve these across all C files that comprise the project. This resolution, in turn, depends on knowing what C files are to be linked during the build of the project, information that is only captured in Makefiles or whatever build automation, if any, a project uses. In this work, we focus on desugaring variability encoded by the preprocessor in C files and instead assume a project only exports one type per global symbol, emitting a type warning when a global symbol has multiple, incompatible type declarations.

4.1.4 Limitations of the transformation

VarAlyzer 's transformation part is generally complete and supports the full (mixed) C language. However, we discuss some fundamental challenges that we discovered while pursuing this research in what follows.

While VarAlyzer translates variability encoded in the preprocessor, large-scale projects also encode variability in other parts of the system software toolchain. All top-level declaration and definitions in a C file are global and externally-linked by default, unless specified otherwise with the `static` keyword. Object files act as modules that import and export these external symbols used in other object files. The definitions of these external symbols can vary based on configuration options, which introduces variability in the linking process. VarAlyzer leaves the difficult

Table 2 Preliminary transformation times for transformations that use partially preprocessed system headers

Program	Runtime in seconds	#Source files	#Configuration variables
axTLS	302	28	94
Toybox	586	230	316
BusyBox	484	554	998

problem of supporting variability-aware linking and build automation to future work and focuses on the variability within C files.

Real-world software often includes dozens or hundreds of header files for the C standard and additional libraries. As shown in Sect. 2, such headers may add hundreds or thousands of function declarations and macro definitions to a C file. These function declarations and macro definitions have to be processed over and over again for each C file that includes the respective headers. Since these header files themselves also encode variability to support different operating systems, various compiler versions, and programming languages (C vs. C++), they currently still pose a scalability challenge to VarAlyzer. Tackling this scalability issue for the transformation requires numerous technical details and implementation tricks that are out of scope for this piece of research and require thorough descriptions on their own. One particular compelling idea is to partially preprocess system headers for specific system configurations to counteract unnecessary processing of these headers at each place they are included. In another branch of research, we have started to implement this idea and have since then be able to successfully transform larger programs such as BusyBox, Toybox and axTLS. Table 2 shows some preliminary results for these programs and should give a first impression in which order of magnitude realistic programs can be transformed.

We support variability across application configurations, but assume a single system configuration. System configuration macros provide several challenges for desugaring variability; they require supporting the header file differences between multiple operating systems, multiple (versions of the same) compiler(s), multiple versions of system libraries, etc. These differences not only cause the number of possible configurations to explode, even when the application code's behavior does not depend on them, but they also impose foundational challenges. VarAlyzer cannot leave these system configuration macros unresolved during transformation since the transformed code could then not be compiled to an intermediate representation for analysis. However, resolving these system configuration macros requires information on all possible operating systems, system libraries, etc. which can hardly be obtained, if at all. And even if it could be, a software product line could not be compiled to an intermediate representation since the environment and the compiler used to produce the intermediate representation of the machine on which the transformation takes place are fixed. In addition, SuperC 's underlying parser is based on one particular version of C as implemented by GCC. Supporting multiple versions of compilers would require constructing a superset of all C variations, a daunting and potentially infeasible task.

4.2 Variational data-flow analysis

We next explain how VarAlyzer makes the analysis variability-aware. This allows one to compute, for all configurations at the same time, results that pinpoint under which configurations they are valid.

VarAlyzer accepts as input *any* given distributive data-flow problem encoded within IFDS (Reps et al. 1995) or IDE (Sagiv et al. 1996), and transforms it into a *variational version* of the problem which can then be solved on an SPL that has been desugared according to Sect. 4.1. Because IFDS problems can be encoded within IDE by using edge functions that operate on the binary lattice $V = ^{\top}_{\bot}$ (Sagiv et al. 1996), we continue by presenting how we model general IDE problems in a variability-aware manner.

VarAlyzer builds on SPLlift 's idea to make use of IDE's edge functions to encode all variants of possible data flows a SPL might induce. SPLlift, however, only allowed "lifting" *IFDS*-based analyses. As mentioned earlier, this precludes an efficient encoding of any problem with a large or even infinite abstract domain, e.g., typestate analysis and constant propagation. To efficiently compute on such large (or infinite) domains, we must instead encode the computation within the edge functions of the IDE framework, but it means that the value computation already occupies the edge functions. Therefore, we then cannot use the edge functions (directly) to capture an SPL's variability information. To be able to solve general IDE problems that already use the edge functions for computing while still capturing an SPL's variability, we need to solve two different value computation problems using IDE's edge functions.

VarAlyzer thus lifts edge functions of the user-defined IDE problem by extending their value domain V_u to produce lifted edge functions that operate on the cartesian product domain $V_l = C \times V_u$, where C is the domain of feature constraints used to describe the variability induced by the preprocessor. This enables VarAlyzer to solve both value computation problems at once, relating analysis results to the exact feature constrains under which they hold. A lifted edge function $\hat{e} : C \mapsto V_u$ is thus a mapping from edge functions that describe the feature constraints to the respective user-defined edge functions that specify the value computation problem that is valid under the associated constraint. Whenever a reachability check is performed on the exploded super-graph that has been produced by the lifted analysis problem P_l, the analysis computes the values specified by the user edge functions and the corresponding constraints that are associated with those values. The result for each reachability check of an ESG node (s, d) for a given statement s and data-flow fact d, i.e., the evaluation of a lifted edge function, is a mapping from feature constraints to their corresponding value $\{c_i \mapsto v_i\}$. In the following, we describe this lifting in more detail. Note that our solution is fully transparent: VarAlyzer can automatically lift any IFDS/IDE analysis problem pre-defined for C programs to software product lines without having to change a single line of code.

Fig. 6 Lifting of edge functions for an ordinary user statement s_u (left) and a branching statement s_p^b that originates from a preprocessor directive (right). For the statement s_u, the user edge function is queried and results in $\lambda x.x + 42$. Because the statement has no effects on the preprocessor constraints, the edge function for the constraint domain is modeled as identity. For s_p^b, the user edge function is modeled as identity because it has no effects on the user's value computation. However, it extends the domain with edge functions that add the preprocessor feature-constraints F and $!F$, respectively

4.2.1 Automated lifting of edge functions

The IDE algorithm is guided through the program using its inter-procedural control-flow graph (ICFG). VarAlyzer operates on a variability-aware version of the control-flow graph $ICFG_v$. The $ICFG_v$ respects the encoding of preprocessor directives as presented in Sect. 4.1. Preprocessor symbols are modeled as `extern` global variables that follow a special naming convention. An $ICFG_v$ can be queried for those global variables and their usages. Any statement that directly interacts with one of those global variables through a def-use chain has been artificially introduced by the code transformation. This allows us to distinguish between any ordinary statement s_u that originates from the user program and any statement s_p that is generated by VarAlyzer transformation parts, originating from preprocessor directives (PPDs).

Initially, i.e., at lifting-time, a lifted edge function $\hat{e}$ maps exactly one edge function that describes a feature constraint to an edge function that, in turn, represents a (user-defined) value computation for a given statement under analysis. The lifting process is depicted in Fig. 6. *Ordinary* statements s_u have no effect on the presence of a certain feature. VarAlyzer thus lifts its user-defined edge function e_u to $\hat{e} := (\lambda c.c) \mapsto e_u$. Here, the identity edge function $\lambda c.c$ over constraints expresses that the feature constraint is not altered. The statement's original flow function (opposed to edge function) remains as is.

For statements s_p that are *generated from the preprocessor directives*, the analysis can safely ignore the non-branching statements since they have been artificially introduced by the transformation and must have no effect on the user-defined value computation. For these statements, VarAlyzer applies the identity flow and edge function. For each generated *branching* statement s_p^b that originates from a preprocessor directive, VarAlyzer produces the corresponding edge function $\hat{e}$ by conjoining the feature constraint F specified by the respective preprocessor conditional with the incoming constraint c, i.e., $\hat{e} := (\lambda c.\ c \wedge F) \mapsto (\lambda x.x)$. Here on the right-hand side, we use the identity edge function $\lambda x.x$ because the statement does not influence the user-defined value computation.

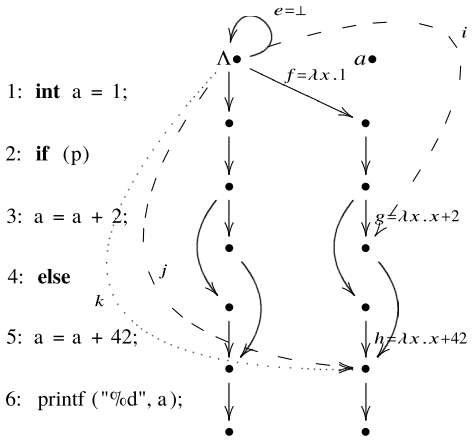

Fig. 7 An exemplary exploded super-graph for a linear constant analysis encoded in IDE (Sagiv et al. 1996). The ESG shows the various operations that edge functions must support. Identity edge functions have been omitted to avoid cluttering. Individual flow functions have been indicated by solid edges and jump functions have been indicated by dotted and dashed edges

4.2.2 Operations on lifted edge functions

To allow for the construction of the exploded super-graph, edge functions need to support the following four operations:

The *composition* operation ($\circ$) composes two edge functions e and f. This operation is used to extend an edge function e and is required to construct the so-called *jump functions* (summaries) that describe the effects of *sequences* of code. An example is shown in Fig. 7. The edge functions e, f, and g can be composed to produce the jump function $i = g \circ (f \circ e) = \lambda x.x + 2 \circ (\lambda x.1 \circ \bot) = \lambda x.3$, which describes the value computation problem for variable a from line 1 to after line 3.

The *join* ($\sqcup$) operation is applied when two paths in the exploded super-graph lead to a common ESG node and the respective edge functions must be combined, for instance, as a result of branching. Consider the example in Fig. 7: the two jump functions i and j are joined to produce the new function $k = i \sqcup j$ that describes the value computation problem for variable a from lines 1 to 5. An *equals* ($=$) operation, comparing two edge functions for equality, is required to update jump functions efficiently within the IDE algorithm, and to ensure termination.

Once an ESG, i.e., all jump functions, is constructed, the value computation problem that is specified by the jump functions can be solved for any given ESG node by simply applying these jump functions. Practical implementations usually do not construct and store the complete ESG but rather only maintain the essential jump functions. To determine the possible value that may be printed in line 6 of Fig. 7, one *evaluates* ($\hookrightarrow$) the respective jump function k. The analysis finds that *any* value may be printed as a result of $\hookrightarrow k = i \sqcup j = \top$.

We next show how to define these four operations for the *lifted* edge functions that operate on the extended user domain $V_l = C \times V_u$ such that a transformed problem P_l can be solved by the IDE algorithm.

Join To join information that is obtained along two (or more) different paths in the ESG, a binary *join* operation is required, see Definition 1. An example of the *join* operation is shown in Fig. 8. When joining, we wish to join also user-defined

Fig. 8 Join of lifted edge functions that have been computed along different control-flow edges. Individual edge functions are denoted by straight arrows (→). Jump functions are denoted by dashed arrows (⇢). The graph on the left depicts the situation when two lifted edge functions must be merged whose constraints are equal. In this case, their user edge functions must be joined. In case the constraints are not equal, they must be left unmerged as two separate pairs of edge functions

edge functions for such constraints that are equal along both branches, as these cases relate to identical feature configurations. Hence the edge functions c_1, c_2 that describe the constraints of the lifted edge functions to be joined are compared pairwise. If $c_1 = c_2$, their corresponding user edge functions u_1 and u_2 are joined. This situation is depicted in the left-hand side graph of Fig. 8. Else if $c_1 \neq c_2$, both results are simply joined by set union, retaining all information about the varying constraints. The latter situation is shown in the right-hand side graph of Fig. 8.

Definition 1 $\sqcup$: Let $\hat{e} = \{c_e^i \mapsto u_e^i\}_{i=0}^n$ and $\hat{f} = \{c_f^j \mapsto u_f^j\}_{j=0}^m$ be two lifted edge functions. We define the *join* operation as:

$$\hat{e} \sqcup \hat{f} := \bigcup_{\substack{(c_e \mapsto u_e) \in \hat{e}, \\ (c_f \mapsto u_f) \in \hat{f}}} \begin{cases} \{c_e \mapsto u_e \sqcup u_f\} & \text{if } c_e = c_f \\ \{c_e \mapsto u_e, c_f \mapsto u_f\} & \text{otherwise} \end{cases}$$

Composition Definition 2 defines the composition operator for lifted edge functions. The program's control can flow only along the two functions' respective program statements when the preprocessor directives that guard these statements are both enabled. Hence, the *compose* operator conjoins the respective feature constraints. The user-defined edge functions meanwhile are composed using their own original composition operator. Whenever the composition operator is applied, one of those edge functions comprises exactly one map entry and the other one may comprise one or more map entries due to potential prior applications of the *join* operation. Those two possible situations for the *composition* operation are shown in Fig. 9. The left-hand side graph of Fig. 9 shows the composition of lifted edge functions for non-branching code. In this case, the edge functions c_1, c_2 that describe the constraints and the user edge functions u_1, u_2 must be composed with each other. As the join of two lifted edge functions at merge points may produce a new edge function that comprises multiple map entries $\{c_1 \mapsto u1, c_2 \mapsto u_2\}$ that need to be composed with the lifted edge function $\{c_3 \mapsto u_3\}$ of the next common successor statement, a pairwise composition must be applied. This situation is depicted in the right-hand side graph of Fig. 9.

Definition 2 $\circ$: Let $\hat{e} = \{c_e^i \mapsto u_e^i\}_{i=0}^n$ and $\hat{f} = \{c_f^j \mapsto u_f^j\}_{j=0}^m$ be two lifted edge functions. We define the *compose* operator as:

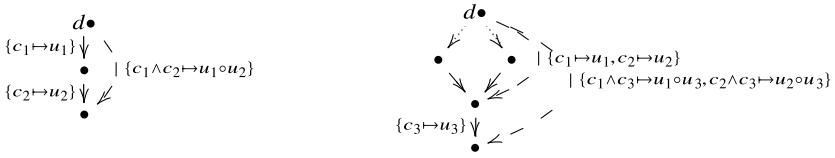

Fig. 9 Composition of lifted edge functions. The left-hand side graph shows the composition of lifted edge functions for non-branching code. The join of two lifted edge functions at merge points may produce a new edge function that comprise multiple edge function pairs that need to be composed with the edge function of the next common successor statement. This situation is depicted in the right-hand side graph

$$\hat{f} \circ \hat{e} := \bigcup_{(c_e \mapsto u_e) \in \hat{e}, (c_f \mapsto u_f) \in \hat{f}} \{c_e \wedge c_f \mapsto u_f \circ u_e\}$$

Equality In addition, the IDE algorithm needs to be able to check for equality of two edge functions. Since we maintain the feature constraints in normalized form, we are able to define two edge functions to be equal if they are equal structurally.

Evaluation Once an exploded super-graph has been constructed, the solver evaluates the value-computation problems described by the jump functions. The value for each ESG node (s, d) that is reachable from the tautological Λ fact is computed by evaluating its associated jump function. We define the unary *evaluate* operation in Definition 3. The evaluation operation of a jump function applies the constraint and user edge-function components of each map entry to the tautological constraint *true* and the bottom element $\perp$ of the user-defined problem, respectively. The result is a map of values that the data-flow fact d can assume, each of which is associated with the feature constraint that encodes the set of configurations under which d holds.

Definition 3 $\hookrightarrow$: Let $\hat{e} = \{c_e \mapsto u_e\}_{i=0}^{n}$ a lifted jump function. We define the unary *evaluate* operator valuate$\hookrightarrow$ as:

$$\hookrightarrow \hat{e} := \{\ c_e^i(true) \mapsto u_e^i(\perp)\ \}_{i=0}^{n}$$

4.2.3 Why IDE is the ideal framework of choice

While VarAlyzer supports IDE, and not only IFDS, IDE still has the restriction that flow functions and edge functions must distribute over the merge operator. The advantage of using such a *distributive* analysis framework to solve data-flow problems on SPLs is that this allows merging variability information directly at each control-flow merge point, *without loss of precision*. This is because for any flow function f and any two abstract domain values x and y of a distributive analysis problem, by definition it holds that $f(x) \sqcup f(y) = f(x \sqcup y)$. As a result, the meet-over-all-paths solution, which is undecidable in general, can be efficiently computed within such frameworks through the maximal-fixed-point solution (Bodden 2018). This solution is the most precise solution possible. The use of IDE thus is guaranteed to

retain full precision w.r.t. a product-based analysis on pure C code, and it guarantees an efficient handling of feature constraints because they are merged and simplified at the earliest opportunity. In result, IDE is the most expressive framework that one can choose without jeopardizing efficiency.

Our idea of capturing variability by using a transparent extension of the user's analysis domain could theoretically also be applied to non-distributive problems. However, this would sacrifice precision and, due to missing summarization capabilities, would likely be prohibitively expensive for any real-world application.

5 Implementation

We implemented VarAlyzer on top of the SuperC (Gazzillo and Grimm 2012) parser and the PhASAR (Schubert et al. 2019) static analysis framework. SuperC supports Bison-style grammars (Bison 2020) for implementing language processors, and automatically parses all variations of a SPL. C constructs with multiple variations due to `#ifdefs` are combined with a static choice tree node that captures each variation and its condition as represented with a logical formula.

VarAlyzer uses SuperC 's existing C grammar and implements the desugarer using semantic actions. A semantic action defines a snippet of code to be executed after each language construct and produces a semantic value for that construct. VarAlyzer records all variations of a construct's desugaring transformation, along with each static condition, as the semantic value of the grammar production. The semantic actions are executed bottom-up, and VarAlyzer gradually constructs the complete, desugared version of the input program by combining the desugared child constructs into larger constructs until reaching the top of the grammar.

VarAlyzer preserves semantic preprocessor information using calls to artificial function headers. Type errors, caused by invalid configurations, are transformed into runtime function calls. VarAlyzer makes the information on symbol renaming available by introducing a symbol table. For each compilation unit, it emits a definition of a static initializer function that specifies the renaming using a function call for each renamed symbol. The static initializer function can be thought of the compilation unit's initializer, because it has no other runtime behavior. The static conditional variables are declared as global boolean variables, since preprocessor macros have no scope and are project-wide. We model preprocessor conditionals using logic formulas and emit a mapping that associates the conditional variables with their respective textual Z3 (de Moura and Bjørner 2008) solver representation using function calls within the initializer function.

VarAlyzer implements the variational analysis presented in Sect. 4.2 on top of PhASAR (Schubert et al. 2019). VarAlyzer provides a wrapper type that can be wrapped around any of PhASAR 's IFDS *and* IDE analyses. The wrapper type wraps the regular user-defined edge functions in a special variability-aware edge function that supports the required operations as described in Sect. 4.2.2. Before VarAlyzer starts the actual analysis at the given entry points on the given target code, it analyzes the aforementioned static initializer function and retrieves the symbol table as well as the preprocessor conditionals. It then decodes the textual

Z3 (de Moura and Bjørner 2008) solver representations of the preprocessor conditionals into their corresponding in-memory `z3::expr` representations, which the analysis uses as part of its lifted edge-function domain. After construction, the variability-aware edge functions are passed to the data-flow solver. The solver follows the control flow of a variability-aware, LLVM-based ICFG implementation that is able to distinguish ordinary instructions from instructions that originate from the preprocessor and have been artificially generated by VarAlyzer 's SPL-transformation part. Once the exploded super-graph is built, the IDE solver solves the value-computation problems, thereby also collecting and computing the feature constraints that are associated with each of the original user-defined edge functions and their respective evaluations.

6 Evaluation

Our empirical evaluation addresses the following research questions:

- *RQ1*. Does VarAlyzer produce results that are identical with these of a product-based analysis?
- *RQ2*. How efficient is VarAlyzer compared to a product-based analysis?
- *RQ3*. To what degree is variational analysis necessary to solve semantic analyses on VarAlyzer-transformed code?

To address *RQ1* and *RQ2*, we compiled each of our 95 benchmark subjects once using VarAlyzer 's conditional compilation approach and once exhaustively using the standard compilation approach for all software products. We then subjected the resulting compiles to VarAlyzer 's variability-aware analysis and a traditional product-based analysis that analyzes each individual software product, respectively. Our benchmark comprises 95 compilation units that make use of OpenSSL's EVP library. For each software product line, we compared the analysis results obtained by VarAlyzer to the results obtained by the product-based approach. We ran each compilation and analysis step five times to account for variance. To address *RQ3* and to answer the question whether variability awareness is necessary, we ran a traditional variability-oblivious inter-procedural typestate analysis encoded in IDE using PhASAR on VarAlyzer-transformed code. We parameterized the typestate analysis for three different APIs of OpenSSL's EVP library. We discuss the precision of the results produced by the traditional variability-oblivious analysis and comment on the reusability of existing static analysis infrastructure on the desugared code.

Unfortunately, comparisons of the VarAlyzer approach to existing tools such as TypeChef (Kenner et al. 2010) or Hercules (Hercules 2020) are either not possible or not very meaningful as the implementations of previous works are not maintained or use different analysis techniques that do not allow one to solve more complex, inter-procedural data-flow analysis problems.

6.1 Experimental setup

We have evaluated VarAlyzer using benchmark subjects consisting of 95 hand-written C compilation units ranging from 8 to 219 lines of source code that comprise correct as well as incorrect usages of OpenSSL's EVP library parts. These compilation units comprise between zero and eleven features and comprise intra- as well as inter-prodcedural usages of the EVP library. We also included compilations units with zero features to assess the potential overhead of VarAlyzer 's conditional compilation and variability-aware data-flow analysis. To obtain correct API uses, we used the code examples presented in OpenSSL's wiki.[2] To ensure that our benchmark programs comprise realistic API usages, we mined 15 SPLs on GitHub using the advanced search and aimed for high-stared and popular projects that make use of OpenSSL's EVP library parts.[3] We then extracted the compilation units that comprise usages of the EVP library and used these to help modeling our benchmark. Surprisingly, several of the real-world API usages completely omit error handling. We thus also omitted error handling code in our benchmark subjects to allow for easier debugging of our transformation and analysis. We then introduced different kinds of protocol breaches, some of them unconditionally and some of them depending on certain (invalid) configurations.

To evaluate VarAlyzer, we used a client typestate analysis $\mathcal{T}$ that had been independently implemented using PhASAR 's implementation of the IDE framework. To allow the analysis to validate useful typestate properties w.r.t. OpenSSL, we parameterized it for the OpenSSL EVP APIs message digests $\mathcal{T}_{MD}$, encryption/decryption (cipher) $\mathcal{T}_{CPR}$, and message authentication codes $\mathcal{T}_{MAC}$. OpenSSL's EVP functionalities provide a high-level interface to OpenSSL's cryptographic functions that are commonly used by projects that require such cryptographic functionalities.

We set up the parameterized typestate analyses to run both in a traditional, variability-oblivious manner using plain PhASAR, which we denote as $\mathcal{T}^{PSR}$, and in a variability-aware manner, which we denote as $\mathcal{T}^{VAR}$. For **RQ1** and **RQ2**, we *exhaustively* sampled and compiled all concrete software products for each SPL of our benchmark to LLVM intermediate representation (LLVM IR) to run the traditional, variability-oblivious typestate analysis $\mathcal{T}^{PSR}$. To be able to run VarAlyzer 's variability-aware analysis $\mathcal{T}^{VAR}$, we desugared each SPL using VarAlyzer 's transformation and then compiled the transformed C code to LLVM IR. We used the standard Clang compiler to produce LLVM IR. For each matching analysis pair, e.g. $\mathcal{T}^{PSR}_{MD}$ (variability-oblivious typestate analysis parameterized for the message digest API) and $\mathcal{T}^{VAR}_{MD}$ (variability-aware typestate analysis parameterized for the message digest API), we automatically checked if the data-flow results produced by $\mathcal{T}^{VAR}$ coincide with all sampled results produced by $\mathcal{T}^{PSR}$, to evaluate the correctness of VarAlyzer 's lifted analysis (**RQ1**). The running times and memory usages of the two approaches are compared in **RQ2**. For **RQ3**, we ran the traditional feature-oblivious

[2] OpenSSL Wiki https://wiki.openssl.org/.

[3] Github advanced search https://github.com/search/advanced.

typestate analysis $\mathcal{T}^{PSR}$ on VarAlyzer-transformed code and compared with its results with the variability-aware analysis $\mathcal{T}^{VAR}$ to assess $\mathcal{T}^{PSR}$'s precision.

We measured the running times and memory usages for the experiments on an Intel i7-5600U CPU@2.60GHz machine running Ubuntu 16.04 with 16GB main memory. We ran each experiment five times, removed the minimum and maximum measuring and computed the average of the remaining three values. We determined the runtimes and peak memory usages of the experiments using the UNIX `time` tool. We measured the lines of code of the compilation units using the UNIX `wc` tool. We formatted the code using the `clang-format` tool and its default settings to allow for a fair comparison of the lines of code measurement. Our benchmark programs, the raw as well as the processed data produced in our evaluation is available in our artifact (Artifacts 2021).

6.2 RQ1: analysis correctness

The PhASAR framework comprises a variety of unit tests for various different parametrizations of the typestate analysis to assess its correctness. It contains tests for parametrizations for C's file API(s) that are concerned with the type `FILE`, OpenSSL's secure heap and secure memory APIs as well as OpenSSL's EVP key derivation API. We developed the typestate parametrizations for OpenSSL's EVP message digest (MD), encryption/decryption (CIPHER), and message authentication codes (MAC) and manually checked their correctness on individually software products that we derived from our benchmark targets. Hence, we can ensure the correctness of the variability-oblivious typestate analysis for the parametrizations $\mathcal{T}^{PSR}_{MD}$, $\mathcal{T}^{PSR}_{CIPHER}$, $\mathcal{T}^{PSR}_{MAC}$ w.r.t. derived programs they have been tested with.

VarAlyzer 's process of lifting IFDS- and IDE-based analysis has been designed to be fully transparent, i.e., it does not modify the semantics of the analysis that is lifted but instead lifts its domain to make it variability-aware—allowing it to distinguish between data-flow facts that have been computed under different feature constraints.

To show that not only theoretically but also in practice VarAlyzer-lifted analyses retain precision compared to their un-lifted, product-based counterpart and also compute the results of all possible software products in a single analysis run, we wrote an automated comparison tool. The comparison tool ran our variational analysis $\mathcal{T}^{VAR}$ on each benchmark subject and then ran its variability-oblivious counterpart $\mathcal{T}^{PSR}$ on all of the exhaustively sampled concrete software products, performing an in-memory comparison of the results. The tool found that the results of $\mathcal{T}^{VAR}$ included the results produced by each analysis run of $\mathcal{T}^{PSR}$. All results, i.e., protocol breaches—on a data-flow fact-level—for each analysis run of $\mathcal{T}^{PSR}$ on a sampled software product can be found in the mapping from feature constraints to data-flow facts produced by $\mathcal{T}^{VAR}$ for the respective feature constraints that describes the software product. Besides that, $\mathcal{T}^{VAR}$ does not introduce spurious data-flow facts that cannot be found in the results of $\mathcal{T}^{PSR}$ run on any concrete software product and hence, VarAlyzer 's results in fact coincide with the results produced by a product-based analysis.

Our variability-aware analysis approach produces results that coincide with the results computed using a corresponding variability-oblivious product-based analysis.

6.3 RQ2: analysis efficiency

Figure 10 presents the results concerning VarAlyzer 's efficiency. Due to space restrictions we can only include the data for the analysis of the 36 benchmark programs that use the OpenSSL EVP encryption/decryption (CIPHER) API and report on the accumulated data for the remaining ones. We have made the complete results available (Artifacts 2021).

Our experiments show that the standard Clang compiler requires between 0.07 and 188.1 seconds to exhaustively compile all concrete software products of a software product line in our benchmark set. VarAlyzer 's two-step desugaring compilation (comprising desugaring and compilation of the desugared code) is, on average, 7.1 times faster, ranging between 0.6 and 1.6 seconds. Running the variability-aware analysis $\mathcal{T}^{VAR}$ on a complete SPL is, on average, 8.0 times faster than analyzing the target software product line using the product-based approach $\mathcal{T}^{PSR}$ that needs to analyze each software product in separate. In total, the complete variability-aware $\mathcal{T}^{VAR}$ pipeline that includes variational compilation and variability-aware analysis is, on average, 7.5 times faster than compiling and analyzing each concrete software product derived from a SPL, while the analysis's memory usage increases by a factor of 1.17.

Figure 10 shows the number of features on a linear scale (the y axis at the bottom), and the accumulated compilation and analysis times of a product-based approach using plain PhASAR for analyzing all sampled concrete software products and VarAlyzer per target program concerning OpenSSL's CIPHER API on a logarithmic scale (the y axis at the top). For the benchmark targets that comprise no variability, the running times of VarAlyzer are generally higher than those of the product-based approach. For most of the compilation units that do comprise variability, the variability-aware approach runs faster than the products-based one as soon as the target comprises more than four features with only one exception. This trend is particularly clear on programs with more features (e.g., *ciis9.c* and *ciise7.c* in Fig. 10). On two occasions (cise4, cise7), however, VarAlyzer 's variability-aware data-flow analysis runs significantly slower than expected. We manually checked the SPLs' source code and found that they use preprocessor integer arithmetic which in our current implementation translates to a relatively long and complex Z3 constraint that slows down the analysis during constraint simplification. The running times of VarAlyzer, apart from the aforementioned two exceptions, generally remain in the same order of magnitude while those of the product-based approach clearly grow exponentially in the number of features reflecting the fact that a software product line may comprise up to $2^{\#features}$ individual software products.

In terms of code size, VarAlyzer 's desugarer causes an increase in lines of code by a factor of 9.2, on average. This is mainly because the desugarer emits artificial function declarations and definitions to preserve the preprocessor's semantics. The definition of the static initializer function, which encodes

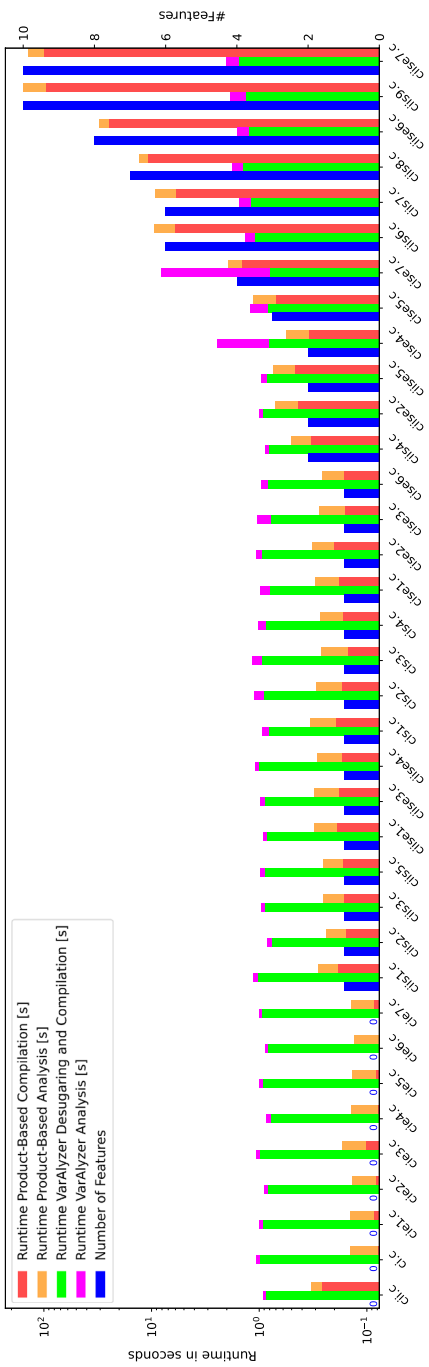

Fig. 10 Analysis efficiency on the benchmark programs that use OpenSSL's EVP CIPHER API. Naming scheme of the benchmark targets: c{'i' - intra-, 'ii' inter-procedural}{'s' - software product line, ' ' - no software product line}{'e' - erroneous API usage, ' ' - correct API usage}

the symbol table for the renamings applied by VarAlyzer and the preprocessor conditionals in Z3's textual representation as described in Sect. 5, is the main cause for the increase in code size. Each renamed symbol leads to an additional line of code, and each preprocessor conditional leads to at least one more line of code. Nevertheless, because the static initializer function is not called within the program but is only used to describe semantics, it does not affect *traditional* variability-oblivious static analyzers.

VarAlyzer outperforms the product-based approach in all cases that comprise more than four features except for one which the current implementation cannot yet handle efficiently. The results would favor the variability-aware analysis even more with an increasing number of features. While the product-based approach requires one to compile and analyze each product, VarAlyzer only requires a single desugaring-compilation and analysis pass.

6.4 RQ3: analysis precision

By comparing the results obtained from running the traditional variability-oblivious analysis $\mathcal{T}^{PSR}$ on VarAlyzer-transformed code, we can assess the precision gained by making the analysis variability-aware. We manually checked the results produced by these analysis runs for the 95 benchmark subjects and observed that the over-approximation leads to great imprecision that renders the results practically unusable. This is because the analysis incorrectly introduces interaction between data flows computed within different features that—according to the preprocessor's semantics—cannot happen in any concrete product. We find that whenever an API's respective context variable is modified across multiple features that cannot actually be enabled together, the typestate analysis $\mathcal{T}^{PSR}$ directly associates those variables with an error state. We can observe that $\mathcal{T}^{PSR}$ returns the most coarse-grain analysis element for the context variables for all 68 compilation units that do comprise variability and whose features differ in the modifications made to the context variable.

Existing analysis approaches presented in literature such as the one by Iosif-Lazar et al. (2017) only employ code transformations to enable the (re)use of existing, unmodified feature-oblivious static analyzers for software product lines. However, information on variability is not preserved and even if it would be, cannot be understood by existing feature-oblivious analyzers. While this generally allows one to apply existing analyzers to entire software product lines, their results are unusable for semantic program analysis as our manual inspection of the results produced by $\mathcal{T}^{PSR}$ on VarAlyzer-transformed code shows. And while simpler, syntax-based analyses may report bugs, it is hard to impossible to account them to a specific feature combination in order to validate and action on them.

For more complex semantic analyses, variability awareness is essential to allow one to distinguish information that is obtained along different mutually exclusive features. To produce useful analysis results on software product lines, one not only requires variability awareness for the transformation, *but* also the analysis parts.

7 Related work

Several previous approaches address, in part, the difficult problem of statically analyzing real-world software product lines (Kästner et al. 2011; Gazzillo and Grimm 2012; Kästner et al. 2012; Chen et al. 2012; Brabrand et al. 2012; Bodden et al. 2013; Midtgaard et al. 2015; Classen et al. 2013; Dimovski 2016). Prior work either created new analyses that had to account for the semantics of the static conditions (Kästner et al. 2011; Gazzillo and Grimm 2012; Garrido and Johnson 2005; Kästner et al. 2012; Kenner et al. 2010) or performed limited syntactic transformations from the preprocessor into C and used off-the-shelf tools (Iosif-Lazar et al. 2017). The works that lift the analysis (Kästner et al. 2011; Gazzillo and Grimm 2012; Garrido and Johnson 2005; Kästner et al. 2012; Kenner et al. 2010) must work on the combined preprocessor/C languages which makes those harder to implement. This causes these approaches to resort to simpler analyses. The approach presented by Iosif-Lazar et al. (2017) misses preprocessor semantics which passes the problem to downstream analyses. The only available data-flow analysis for software product lines written in C (Liebig et al. 2013; Brabrand et al. 2012) is intra-procedural only. To employ precise, inter-procedural static analysis the transformation of the preprocessor directives into ordinary C must be able to handle all of the preprocessor's constructs and, in addition, preserve full information on static preprocessor conditionals. The latter requirement is necessary to make this information available to downstream analysis to avoid a loss in precision.

SPLlift (Bodden et al. 2013) avoids generating all potential software products by analyzing the entire SPL as a whole. This so-called family-based approach encodes feature constraints in distributive flow functions. SPLlift solves IFDS (Reps et al. 1995) problems on SPLs using IFDS's generalization IDE (Sagiv et al. 1996). However, SPLlift can only solve data-flow problems with the small and finite domains, limited by IFDS. SPLlift is a prototype for a seldom-used product-line dialect of Java (Kästner et al. 2009) and thus cannot be applied to real-world product lines, especially not those that use the C preprocessor.

SuperC (Gazzillo and Grimm 2012) presents a configuration-preserving lexer, preprocessor, and parser. Its preprocessor resolves includes and macros while leaving static conditionals intact to preserve its variability. A configuration-preserving parser then generates an abstract syntax tree (AST) that is additionally amended with static choice nodes to represent the static conditionals. SuperC uses a performant fork-merge parsing: it forks subparsers whenever a choice node is encountered and merges after the conditionals. The approach explores how to perform syntactic analysis of C code while preserving its variability. SuperC provides detailed insights on preprocessor usages and interactions of preprocessor usages of software product lines.

TypeChef (Kenner et al. 2010) is another variability-aware parser and typechecker for product lines written in C and allows for detecting variability-induced bugs in configurable systems. It avoids combinatorial explosion by parsing the entire source code in a variability-aware fashion *without* preprocessing. Similar to SuperC, it produces an AST that captures the variability using static choice

nodes. Based on TypeChef 's AST, a variability-aware type system has been developed that type-checks C code with compile-time configurations. While it is possible to implement static program analyses that operate on variability-aware ASTs, those analyses would still only be syntax based and, in addition, would still need to encode the variability themselves (e.g., Liebig et al. 2013; Brabrand et al. 2012). Variability-aware control-flow and syntax-based data-flow analysis can also be implemented on top of TypeChef. However, this requires the development of syntactic AST-based analyses from scratch for the preprocessor/C language. Instead, our approach does not need to capture the static behavior. This allows us to build on existing, sophisticated program analyses; we use PhASAR and our variability-aware extension VarAlyzer.

Hercules (Hercules 2020), a rewriting and refactoring engine built on top of TypeChef, is a source-code transformation tool similar to the goal of SuperD. It transforms compile-time variability into runtime variability. It no longer relies solely on syntactic analysis only but also allows for more difficult semantic analyses as well. Hercules, however, relies on TypeChef 's variability-aware parsing and analysis infrastructure which limits the application to code that is type-error-free, a requirement that real-world code does not hold. Our approach is able to pass *all* information of static preprocessor conditionals to our downstream analysis. This allows for more precise subsequent analyses. For instance, it expresses type errors as ordinary function calls, which allows its subsequent analysis to collect type errors while analyzing the program without the need to exit immediately.

Iosif-Lazar et al. (2017) created C Reconfigurator that translates product lines into single programs by replacing compile-time variability with run-time variability. The resulting programs can be analyzed using traditional off-the-shelf analysis tools such as clang-tidy (clang tidy 2018) or Frama-C (Cuoq et al. 2012). However, C Reconfigurator does not preserve information on the origin of a static conditional, making the results produced by the off-the-shelf tools on the transformed code variability-unaware. Instead, our approach preserves full information on the preprocessor's semantics and can compute the analysis results and their respective variants on-the-fly in a single analysis run. C Reconfigurator also does not include feasible but invalid configurations in the transformed program, making the bugs caused by these configuration impossible to detect.

Le and Pattison (2014) presented the Hydrogen framework that introduced multiversion inter-procedural control-flow graphs (MVICFGs). MVICFGs represent the control flows of multiple versions of a program in a single graph whose edges are annotated with the version(s) under which a control flow is feasible. MVICFGs can be used for incremental update analysis and for determining the bug/patch impact for multiple program releases. The ICFGs of VarAlyzer-transformed programs can be viewed as MVICFGs with the difference that VarAlyzer 's ICFGs represent all possible variants of a software product line instead of (potentially) all versions of an individual software product. While Hydrogen employs a demand-driven symbolic analysis whose queries must be parameterized with a specific version for which to compute results, VarAlyzer 's lifted distributive data-flow analyzes compute the results for all possible software products in a single analysis run and accounts them to the constraints under which they are valid.

8 Conclusions

We have presented the design and implementation of VarAlyzer. VarAlyzer allows one to produce a configuration-preserving encoding off all variability in regular C code which it then subjects to a variability-aware, context- and flow-sensitive data-flow analysis. It enables computing precise results on entire software product lines, annotated with feature constraints that encode in which product configurations each result is valid. Our empirical study using 95 compilation units that make use of OpenSSL shows that this approach outperforms a traditional product-by-product analysis as soon as more than four products need to be analyzed. As a result, for the first time VarAlyzer allows one to conduct an effective static data-flow analysis of software product lines on real-world C code. This has the great potential to allow developers to find bugs and vulnerabilities much earlier in the development process. For instance, whereas previously developers using OpenSSL had to identify vulnerabilities separately for each concretely preprocessed variant of OpenSSL, using VarAlyzer now has the potential to allow the OpenSSL maintainers to detect such vulnerabilities for all relevant configurations ahead of time.

Acknowledgements Open Access funding enabled and organized by Projekt DEAL. This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre 901 "On-The-Fly Computing" under the project number 160364472-SFB901/3, the Heinz Nixdorf Foundation, and NSF grants CCF-1840934 and CCF-1816951.

Data availability and material (data transparency) All accompanying artifacts of this paper, including processed analysis targets and result data, are available as supplemental material (Artifacts 2021).

Code availability (software application or custom code) We will make the implementation of VarAlyzer available as open source and make it available under the permissive MIT license.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

Artifacts: supplementary material (2021). https://drive.google.com/drive/folders/1ESiSu5iKsFTrM2XqN3Oj4fhIqVfdQ93W?usp=sharing

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, pp. 259–269. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2594291.2594299

Bison: bison. https://www.gnu.org/software/bison/ (2020)

Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M.: Spllift: Statically analyzing software product lines in minutes instead of years. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, pp. 355–364. ACM, New York, NY, USA (2013). https://doi.org/10.1145/2491956.2491976

Bodden, E.: The secret sauce in efficient and precise static analysis: The beauty of distributive, summary-based static analyses (and how to master them). In: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA '18, pp. 85–93. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3236454.3236500

Brabrand, C., Ribeiro, M., Tolêdo, T., Borba, P.: Intraprocedural dataflow analysis for software product lines. In: Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development, AOSD '12, pp. 13–24. Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2162049.2162052

Chen, S., Erwig, M., Walkingshaw, E.: An error-tolerant type system for variational lambda calculus. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming, ICFP '12, pp. 29–40. Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2364527.2364535

Clang tidy: clang-tidy (2018). http://clang.llvm.org/extra/clang-tidy/

Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., Raskin, J.F.: Featured transition systems: foundations for verifying variability-intensive systems and their application to LTL model checking. IEEE Trans. Softw. Eng. **39**(8), 1069–1089 (2013). https://doi.org/10.1109/TSE.2012.86

CodeSonar, G.: Grammatech codesonar (2018). https://www.grammatech.com/products/codesonar

Coverity-(SAST): Coverity static application security testing (SAST) (2018). https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html

Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. In: Proceedings of the 10th International Conference on Software Engineering and Formal Methods, SEFM'12, pp. 233–247. Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16

de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: C.R. Ramakrishnan, J. Rehof (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

Dimovski, A.S.: Symbolic game semantics for model checking program families. In: Bošnački, D., Wijs, A. (eds.) Model Checking Software, pp. 19–37. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-32582-8_2

Ernst, M.D., Badros, G.J., Notkin, D.: An empirical analysis of C preprocessor use. IEEE Trans. Softw. Eng. **28**(12), 1146–1170 (2002). https://doi.org/10.1109/TSE.2002.1158288

FileVaultBug: Apple security blunder exposes lion login passwords in clear text. https://www.zdnet.com/article/apple-security-blunder-exposes-lion-login-passwords-in-clear-text/ (2012)

Garrido, A., Johnson, R.: Analyzing multiple configurations of a C program. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, ICSM '05, pp. 379–388. IEEE Computer Society, USA (2005). https://doi.org/10.1109/ICSM.2005.23

Gazzillo, P., Grimm, R.: Superc: parsing all of C by taming the preprocessor. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11–16, 2012, pp. 323–334. ACM (2012). https://doi.org/10.1145/2254064.2254103

GCC-Optimize-Options: GCC optimize options (2018). https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Hercules: Hercules. https://github.com/joliebig/Hercules (2020)

Hermann, B., Reif, M., Eichberg, M., Mezini, M.: Getting to know you: Towards a capability model for java. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 758–769. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2786805.2786829

Holzinger, P., Hermann, B., Lerch, J., Bodden, E., Mezini, M.: Hardening java's access control by abolishing implicit privilege elevation. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 1027–1040 (2017). https://doi.org/10.1109/SP.2017.16

ICCOptimizeOptions: Intel®c++ compiler 19.0 developer guide and reference: Interprocedural optimization (IPO) (2018). https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-interprocedural-optimization-ipo

Iosif-Lazar, A.F., Melo, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Effective analysis of C programs by rewriting variability. CoRR (2017). arxiv:1701.08114

Kästner, C., Giarrusso, P.G., Rendel, T., Erdweg, S., Ostermann, K., Berger, T.: Variability-aware parsing in the presence of lexical macros and conditional compilation. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11, pp. 805–824. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2048066.2048128

Kästner, C., Ostermann, K., Erdweg, S.: A variability-aware module system. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, pp. 773–792. Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2384616.2384673

Kästner, C., Thum, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., Apel, S.: Featureide: A tool framework for feature-oriented software development. In: 2009 IEEE 31st International Conference on Software Engineering, pp. 611–614. IEEE (2009). https://doi.org/10.1109/ICSE.2009.5070568

Kästner, C.: Virtual separation of concerns: toward preprocessors 2.0. Ph.D. thesis, Otto von Guericke University Magdeburg (2010). https://doi.org/10.1524/itit.2012.0662. http://edoc.bibliothek.uni-halle.de/servlets/DocumentServlet?id=8044

Kästner, C., Apel, S., Thüm, T., Saake, G.: Type checking annotation-based product lines. ACM Trans. Softw. Eng. Methodol. (2012). https://doi.org/10.1145/2211616.2211617

Kenner, A., Kästner, C., Haase, S., Leich, T.: Typechef: Toward type checking #ifdef variability in C. In: Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development, FOSD '10, pp. 25–32. ACM, New York, NY, USA (2010). https://doi.org/10.1145/1868688.1868693

Krüger, S., Nadi, S., Reif, M., Ali, K., Mezini, M., Bodden, E., Göpfert, F., Günther, F., Weinert, C., Demmler, D., Kamath, R.: Cognicrypt: Supporting developers in using cryptography. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, pp. 931–936. IEEE Press, Piscataway, NJ, USA (2017). http://dl.acm.org/citation.cfm?id=3155562.3155681

Le, W., Pattison, S.D.: Patch verification via multiversion interprocedural control flow graphs. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 1047–1058. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2568225.2568304

Liebig, J., Kästner, C., Apel, S.: Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In: Proceedings of the Tenth International Conference on Aspect-Oriented Software Development, AOSD '11, pp. 191–202. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1960275.1960299

Liebig, J., von Rhein, A., Kästner, C., Apel, S., Dörre, J., Lengauer, C.: Scalable analysis of variable software. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, pp. 81–91. ACM (2013). https://doi.org/10.1145/2491411.2491437

Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14, SSYM'05, pp. 18–18. USENIX Association, Berkeley, CA, USA (2005). http://dl.acm.org/citation.cfm?id=1251398.1251416

McCloskey, B., Brewer, E.: Astec: A new approach to refactoring c. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, pp. 21–30. Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1081706.1081712

Midtgaard, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Systematic derivation of correct variability-aware program analyses. Sci. Comput. Program. **105**, 145–170 (2015). https://doi.org/10.1016/j.scico.2015.04.005

Onlinedocs, G.: Gcc onlinedocs – cpp 3.4 stringizing (2021). https://gcc.gnu.org/onlinedocs/gcc-11.2.0/cpp/Stringizing.html#Stringizing

Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '95, pp. 49–61. ACM, New York, NY, USA (1995). https://doi.org/10.1145/199448.199462

Reps, T., Schwoon, S., Jha, S.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: Proceedings of the 10th International Conference on Static Analysis, SAS'03, pp. 189–213. Springer-Verlag, Berlin, Heidelberg (2003). http://dl.acm.org/citation.cfm?id=1760267.1760283

Rhein, A.V., Liebig, J., Janker, A., Kästner, C., Apel, S.: Variability-aware static analysis at scale: an empirical study. ACM Trans. Softw. Eng. Methodol. (2018). https://doi.org/10.1145/3280986

Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theor. Comput. Sci. **167**(1–2), 131–170 (1996). https://doi.org/10.1016/0304-3975(96)00072-2

Schubert, P.D., Hermann, B., Bodden, E.: Phasar: An inter-procedural static analysis framework for c/c++. In: T. Vojnar, L. Zhang (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 393–410. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_22

Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. New York Univ. Comput. Sci. Dept., New York, NY (1978). https://cds.cern.ch/record/120118

Strom, R.E.: Mechanisms for compile-time enforcement of security. In: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '83, pp. 276–284. ACM, New York, NY, USA (1983). https://doi.org/10.1145/567067.567093

Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. **12**(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929

Thüm, T., Apel, S.: Analysis strategies for software product lines. none (2012). https://www.cs.cmu.edu/~ckaestne/pdf/tr_analysis12.pdf

Walkingshaw, E., Kästner, C., Erwig, M., Apel, S., Bodden, E.: Variational data structures: Exploring tradeoffs in computing with variability. In: Black, A.P., Krishnamurthi, S., Bruegge, B., Ruskiewicz, J.N. (eds.) Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014, pp. 213–226. ACM (2014). https://doi.org/10.1145/2661136.2661143

Authors and Affiliations

Philipp Dominik Schubert[1] · Paul Gazzillo[2] · Zach Patterson[3] · Julian Braha[2] · Fabian Schiebel[4] · Ben Hermann[5] · Shiyi Wei[3] · Eric Bodden[1,4]

Paul Gazzillo
paul.gazzillo@ucf.edu

Zach Patterson
zach.patterson@utdallas.edu

Julian Braha
julianbraha@gmail.com

Fabian Schiebel
fabian.schiebel@iem.fraunhofer.de

Ben Hermann
ben.hermann@cs.tu-dortmund.de

Shiyi Wei
swei@utdallas.edu

Eric Bodden
eric.bodden@upb.de

[1] Paderborn University, Paderborn, Germany

[2] University of Central Florida, Florida, USA

[3] University of Texas at Dallas, Dallas, USA

[4] Fraunhofer IEM, Paderborn, Germany

[5] Technische Universität Dortmund, Dortmund, Germany