







Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis

Philipp Dominik Schubert   

Heinz Nixdorf Institute, Paderborn, Germany

Ben Hermann   

Technische Universität Dortmund, Germany

Eric Bodden   

Heinz Nixdorf Institute, Paderborn, Germany

Fraunhofer IEM, Paderborn, Germany

Abstract

Static analysis is used to automatically detect bugs and security breaches, and aids compiler optimization. Whole-program analysis (WPA) can yield high precision, however causes long analysis times and thus does not match common software-development workflows, making it often impractical to use for large, real-world applications.

This paper thus presents the design and implementation of MODALYZER, a novel static-analysis approach that aims at accelerating whole-program analysis by making the analysis modular and compositional. It shows how to compute *lossless*, persisted summaries for callgraph, points-to and data-flow information, and it reports under which circumstances this function-level compositional analysis outperforms WPA.

We implemented MODALYZER as an extension to LLVM and PhASAR, and applied it to 12 real-world C and C++ applications. At analysis time, MODALYZER modularly and losslessly summarizes the analysis effect of the library code those applications share, hence avoiding its repeated re-analysis. The experimental results show that the reuse of these summaries can save, on average, 72% of analysis time over WPA. Moreover, because it is lossless, the module-wise analysis fully retains precision and recall. Surprisingly, as our results show, it sometimes even yields precision superior to WPA. The initial summary generation, on average, takes about 3.67 times as long as WPA.

2012 ACM Subject Classification Software and its engineering → Automated static analysis

Keywords and phrases Inter-procedural static analysis, compositional analysis, LLVM, C/C++

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.2

Funding 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 and the Heinz Nixdorf Foundation.

1 Introduction

Static analysis plays an important role in modern software development. While intra-procedural static data-flow analysis might only be useful in a limited number of use-cases, inter-procedural analysis is a powerful building block for bug finding [4, 7, 34], compiler optimization [6, 8] and software hardening [22, 39, 40, 44, 47].

Static analysis is known to be an undecidable problem [57], which challenges static-analysis designers to define analyses that are both precise (yielding little to no approximate information) and efficient. To obtain good precision, static program analyses need to be inter-procedural, i.e., cross procedure boundaries, and also must be context sensitive [68]. Moreover, they must be based on precise points-to analyses [26].



© Philipp Dominik Schubert, Ben Hermann, and Eric Bodden;
licensed under Creative Commons License CC-BY 4.0

35th European Conference on Object-Oriented Programming (ECOOP 2021).

Editors: Manu Sridharan and Anders Möller; Article No. 2; pp. 2:1–2:31

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Such inter-procedural analysis, however, especially if implemented as a whole-program analysis (WPA), is notorious for causing problems with scalability in both runtime and memory consumption. The memory consumption required for larger programs to keep the complete program representation as well as all of the data structures required to perform the analyses and optimizations in memory can easily grow to a large two-digit GB figure [13,80]. Analysis times can amount to several hours, impeding development processes even in cases where the analysis is deployed as nightly build [23,46,69].

There are application scenarios for which one can yield useful results with *intra-procedural* analyses that are simple enough to scale. The clang-tidy tool [2] and Cppcheck [5] use *syntactic* analyses that are able to analyze software comprising a million lines of code within minutes. Many *semantic* program analyses, however, such as data-flow [42], tpestate [74,75] or shape analyses [82], for instance, require detailed program representations that incorporate the effects of procedure calls, yet are virtually impossible to scale if computed for the whole program at once. This precludes important application scenarios, for instance, IDE integration or the automated scanning of frequently changing software. Facebook, for instance, reports that its code base changes so frequently that it has become a real challenge to design analysis tools such that they can report errors quickly enough so that they are still relevant and actionable when reported [37].

In this work, we aim to scale static context-, flow-, and field-sensitive inter-procedural program analysis using a compositional computation of analysis information. The effectivity of this compositional program analysis depends on the number of reusable parts of an application, e.g., program parts that constitute frameworks or libraries, or for parts that simply do not change from one analysis run to the next. A recent study by Black Duck (Synopsis) has shown that more than 96% percent of the applications they scan contain open-source components and that those components now make up, on average, 57% of the code [70]. As those dependencies are updated much less frequently than application code, compositional analysis can potentially accelerate the analysis of applications by reusing analysis results from previous runs.

Previous work on compositional program analysis has been restricted to certain types of data-flow analysis only. Reviser [20], for instance, allows for the ahead-of-time computation of reusable taint-analysis summaries for Java libraries. Reviser builds on concepts by Rountev et al. [62], who showed how to obtain reusable libraries for general distributive data-flow problems. Both those previous approaches, however, have two significant limitations: First, they only apply to Java, making it unclear which concepts carry over to other languages, particularly C/C++, which allow more liberal pointer accesses to the stack and heap. Second, they only apply to data-flow analysis and leave out the composition of points-to and callgraph information. Especially the latter is a serious practical limitation: when composing a library summary with application code, these approaches again perform an expensive whole-program points-to and callgraph analysis, which in itself can take several minutes if not hours to complete. In result, these approaches incrementalize only the tip of the proverbial iceberg. Addressing this limitation is complex as callgraph, points-to, and data-flow information are inter-dependent. A core conceptual contribution of this paper is therefore also *a mechanism for analysis dependency management for a fully compositional analysis*. This mechanism automatically triggers updates whenever novel information becomes available that affects existing information.

An important practical factor impacting the scalability of compositional analysis is the mechanism to persist summaries. While the approach by Rountev et al. [62] *computes* summaries, they are *not persisted at all* [58] but rather discarded at analysis shutdown, which

completely defeats their purpose. Reviser [20] does persist summaries, but its summary format is only applicable to taint analysis that uses a binary lattice \mathbb{T} . Finding an efficient summary format that is able to persist general data-flow information is challenging due to arbitrarily complex lattices used by more advanced analyses. However, efficient and generalized persistence of summaries is key to effective compositional analysis.

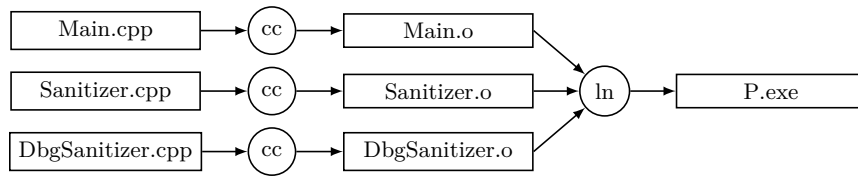
This paper presents MODALYZER, a novel approach to compositional analysis that in contrast to earlier approaches performs an *integrated compositional analysis* for callgraph, points-to and context-sensitive data-flow information in a module-wise fashion. MODALYZER allows the compositional pre-computation of all three pieces of information for individual C/C++ modules, such as libraries and frameworks. Information precomputed this way is then efficiently persisted, and later-on *merged* into larger analysis scopes. Merging analysis information efficiently is an integral part of any compositional analysis approach as combining analysis information computed on individual pieces of code is required to produce overall analysis results.

As our experiments show, this frequently helps to achieve a more efficient analysis of entire applications (compared to WPA) while retaining the same level of precision and recall of a matching WPA.

Interestingly, as this paper shows, merge operations on different types of analysis information can be modelled in a common way by defining merge operations on their respective graph representations. MODALYZER thus conducts its compositional computation of callgraph, points-to, and data-flow information using those graph operations. While MODALYZER compositionally computes all these kinds of information, it also manages the dependencies among them, and updates dependent information as required. MODALYZER creates summaries for callgraph and points-to analysis, and for data-flow analyses expressed in the IFDS [55] and IDE [63] frameworks. Those frameworks support data-flow analyses whose flow functions distribute over the meet operator, which in turn allows for an efficient and – as we also show empirically – *lossless* summarization. MODALYZER does not lose any information and also does not have to overapproximate missing information. Instead, it leaves gaps that will be eventually filled-in during summary application resulting in the same information that would have been obtained by a matching whole program analysis. Many useful data-flow analyses, among others taint analysis as well as all Gen/Kill problems, can be encoded in those *distributive frameworks*. MODALYZER also allows for the computation of more expressive analyses in the monotone framework [41]. While one generally cannot create *data-flow* summaries for such analyses (an undecidable problem), these analyses nonetheless can benefit from summaries for points-to and callgraph information. This still allows to greatly accelerate analysis computations even for *non-distributive* analysis problems.

We have implemented MODALYZER on top of PhASAR [64] and LLVM [45]. We show the improvements of MODALYZER’s compositional analysis over traditional whole-program analysis by analyzing 12 real-world C/C++ applications of various sizes, reaching from 129,000 to 1,400,000 lines of code. For each application, we perform two client analyses (uninitialized-variables analysis and taint analysis), once in whole-program mode and once using library summaries pre-computed by MODALYZER. We compare the resulting running times and client reports to validate the equivalence in precision and recall, and to assess analysis time. Our experiments show that MODALYZER can decrease the analyses’ runtimes between 28% and 91% while keeping the initial one-time runtime overhead for summarization of library parts at 3.67 times as long as the cost of a whole-program analysis.

We will make the implementation of MODALYZER available as open source under the permissive MIT license. We subject it to artifact evaluation. All accompanying artifacts of this paper, including the processed target applications, their modularizations, and result data are available online under the MIT license [16].



■ **Figure 1** C/C++'s compilation model. `cc` is the C/C++ compiler. `ln` is the linker.

In summary, this paper makes the following contributions: it presents

- the first *integrated compositional analysis* for callgraph, points-to and context-sensitive data-flow information with appropriate summarization techniques and summary formats,
- MODALYZER, an open-source C++ implementation within the PhASAR [64] framework, allowing the full module-wise computation of arbitrary distributive static analysis problems (and module-wise computation of points-to and callgraph information for non-distributive analysis problems),
- and an experimental evaluation of MODALYZER, which shows that not just in theory but also in practice precision and recall are retained, and which assesses under which circumstances the reuse of summaries can decrease the overall analysis time.

2 Motivating Example and Intuition

C/C++ programs are usually organized in several files that provide some limited form of modularity. An implementation and its corresponding header file are often referred to as a *compilation unit* or *module*. The compiler translates each module separately and thus, has only knowledge about the information contained within the module that is currently compiled. The resulting object file contains executable program code, which may, however, contain unresolved references. The linker resolves these references across two or more object files and may add links to external libraries. The result after the linkage step is an executable program. Figure 1 depicts the corresponding mechanism.

The vast majority of modern software is not written from scratch, but rather uses libraries, which enable code reuse, faster development and is less error prone [12, 80]. Thus, only a small amount of a program is actual application code and large parts are library code. Once a library has been introduced as a dependency it is rarely changed compared to the application code that uses it.

Our example program is comprised of three compilation units (CUs) – often called *modules* in the C/C++ context – `Main`, `Sanitizer`, and `DbgSanitizer` shown in Listing 1, 2, and 3. We omit the header files for brevity of presentation. The example program is built according to the compilation model presented in Figure 1.

Let us assume that `Sanitizer` and `DbgSanitizer` form a library for sanitization tasks called `libsan`. In C/C++, a library is a collection of one or more object files that have been compiled in form of an archive or shared object file. We further assume that `Main` represents the user application that makes use of the `libsan` library. We use the example program shown in Figure 2 as a running example to detail on our module-wise analysis (MWA) approach.

As a client analysis we use a *taint* analysis which is able to detect potential SQL injections in programs. A taint analysis tracks values that have been tainted by one or more *sources* through the program and reports a leak, if a tainted value reaches a *sink*. The analysis considers all user inputs of a program which potentially contain malicious data as tainted,

```

1 int main(int argc, char **argv) {
2     auto *con = driver->connect(/* connection properties */);
3     auto *stmt = con->createStatement();
4     string q = "SELECT name FROM students where id=";
5     string input = argv[1];
6     string sanin = applySanitizer(input);
7     auto *res = stmt->executeQuery(q + sanin);
8     res->beforeFirst();
9     if (!res->rowCount()) { cout << "no record found\n"; }
10    while (res->next()) { cout << res->getString("name") << '\n'; }
11    delete stmt; delete res; delete con; return 0;
12 }

```

Listing (1) Main – Contains the main application code.

```

13 struct Sanitizer {
14     virtual ~Sanitizer() = default;
15     virtual string sanitize(string &in) {
16         if (isMalicious(in)) { in = /*actual sanitization*/; }
17         return in;
18     }
19     bool isMalicious(string &in) { return /*check if malicious*/; }
20 };
21 string applySanitizer(string &in) {
22     Sanitizer *s = getGlobalSan();
23     string out = s->sanitize(in);
24     return out;
25 }

```

Listing (2) Sanitizer – A module of the sanitization library.

```

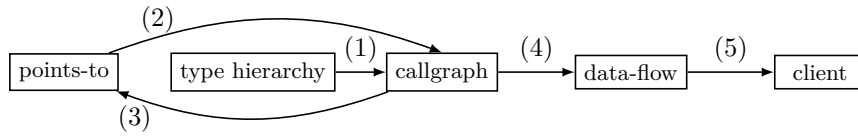
26 struct DbgSanitizer : Sanitizer {
27     bool disable = true;
28     ~DbgSanitizer() override = default;
29     string sanitize(string &in) override {
30         if (!disable && isMalicious(in)) { throw malicious_input(":'("); }
31         return in;
32     }
33 };
34 Sanitizer *getGlobalSan() {
35     static Sanitizer *s = new DbgSanitizer;
36     return s;
37 }

```

Listing (3) DbgSanitizer – A module of the sanitization library.

■ **Figure 2** Modular example program.

e.g. the parameters `argc` and `argv` that are passed into the `main()` function in our example program presented in Listing 1. The function `Statement::executeQuery()` serves as a sink in this scenario. Without sanitization, a malicious user of the program could carefully craft the string “1 OR TRUE;” and pass it as the program’s second command-line argument. As the input string is just concatenated the database server will return the names of all students not just the one where the id matches. By crafting such malicious inputs, a user can leak or alter the data stored in the database. A tainted value may be *sanitized* in our scenario by using the `Sanitizer::sanitize()` function (Listing 2) that clears malicious contents, and therefore *un-taints* a value. The client analysis \mathbb{T} aims to find flows of (unsanitized) tainted values to sinks and reports a potential SQL injection vulnerability whenever it finds such an illegal flow.



■ **Figure 3** Dependencies of a client analysis involving type hierarchy, points-to information, inter-procedural control-flow and data-flow information. Numbered edges determine computation order.

3 Framework Architecture

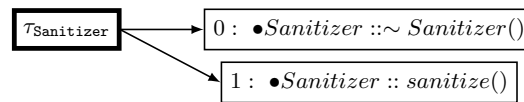
In this section, we elaborate on our compositional, *module-wise analysis*. We first present the idea of the algorithm in a nutshell and continue with our concept of summary generation. We then explain the steps we take for result merging and optimizations. As summaries are always depending on assumptions made, we discuss them at the end of this section.

3.1 Idea of the Algorithm

We have built our module-wise analysis approach following C/C++’s compilation model. To determine a program property of interest, a concrete data-flow analysis, the *client*, may require information from other analyses as shown in the dependency graph in Figure 3. To be able to determine the inter-procedural data flow that a concrete client analysis is interested in, a precise callgraph is needed. A precise callgraph, in turn, requires points-to information [26] and the type hierarchy of the program, but points-to analysis requires a callgraph as well. The data-flow information depends on the callgraph and the client analysis transitively depends on all of these pieces of information. Note that a points-to analysis does require information on subtyping. Information such as the declared and allocated pointer types can be queried ad-hoc. Many useful static analyses can be encoded using the dependencies show in Figure 3 and thus, we will assume such a scenario in this paper.

To achieve fully compositional analysis information for all levels of information as shown in Figure 3, we must be able to (i) compute all information required for a client analysis on a function level (except the type hierarchy, which is always computed on a module level) and *summarize* them, (ii) *merge* the information and (iii) perform an *update* if a merge reveals new information that affect the current results. The merge operation combines static analysis summaries computed on two individual modules into a novel summary such that it reflects the information that would have been obtained by linking those modules first and then computing the static analysis information afterwards. In such an MWA-style analysis library modules would be analyzed separately. Their computed summaries would be merged whenever necessary while analyzing a program which uses those library modules.

As mentioned in Section 1, the compositional approaches to static data-flow analysis presented by Rountev et al. [62] and Reviser [20] only apply to Java. In that regard, MODALYZER can take advantage of C’s and C++’s language characteristics, which are quite different from Java. The MODALYZER approach merges summaries for each function per compilation unit. The intuition is that related source code often resides within the same compilation unit. Because C and C++ are often used to implement performance-critical applications [1, 11], developers have a great interest in making as much information available to the compiler as possible within an individual compilation unit. Otherwise, the compiler would not be able to perform inlining and other important optimizations in an ordinary (i.e., non-WPA) compilation setup [49, 50]. Additionally, whereas all function members (or methods) in Java are virtual, function members are non-virtual by default in C++. It



■ **Figure 4** Type hierarchy and respective virtual function table(s) of the `Sanitizer` module.

generally seems that C++ programs make use of dynamic dispatch less frequently to avoid performance penalties [17, 29, 32], a property that MODALYZER, again, uses to our advantage. Summaries computed for C/C++ code are thus more expressive and less likely to contain *gaps* due to missing information. While MODALYZER, in general, is applicable to other languages as well it might work better for C and C++ programs than for programs written in Java or C#, for instance, which use virtual calls all over the place. For those languages, the portion of partial summaries will increase and the overall performance of MODALYZER will degrade as more gaps need to be closed while analyzing the “main application”. Previous works by Rountev show that summarization techniques nonetheless can greatly improve running times for large Java applications, even when restricted to data-flow analysis only. We elaborate on that in detail in Section 5.

In the following, we show that merge operations on analysis information can be modelled in a common way through merge operations performed on their respective graph representations. However, special care must be taken to update the dependent information accordingly if new information becomes available due to merging two module summaries. This makes it crucial to keep particularly the callgraph up to date, as all other information except the type hierarchy depend on it.

3.2 Summary Generation

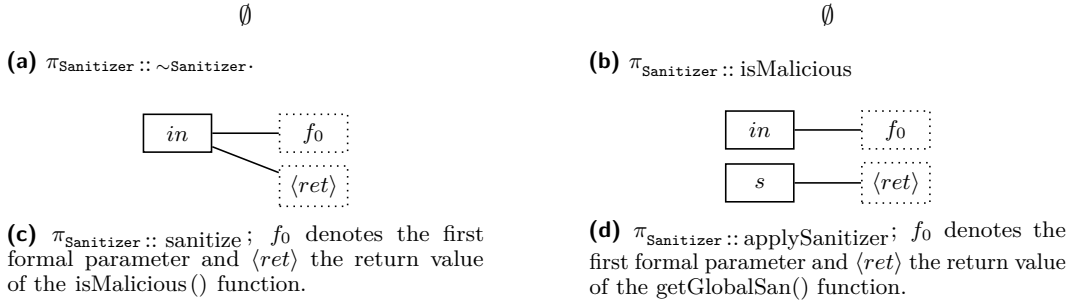
In the following, we will explain the steps of our analysis based on the example presented in Section 2. The assumption is that `Main` changes frequently, and `libsan` only once in a while. For presentation here we start our library pre-analysis by analyzing the `Sanitizer` module, although the analysis algorithm does not make any assumptions about module order.

3.2.1 Type Hierarchies

Our approach first computes the type hierarchy as it is the most robust structure in the sense that the amount of information can only grow monotonically. We use τ_t to denote the type of a class or struct t and we use T_C to denote the type hierarchy for a module C . In addition, the type hierarchy maintains information on the virtual function tables (call targets) for C++’s *struct* or *class* types that declare virtual functions.

► **Example 1.** The analysis will find that the type hierarchy for the `Sanitizer` module consists of a graph containing a single node representing the type $\tau_{\text{Sanitizer}}$. The call target for $\tau_{\text{Sanitizer}}$ contains two entries, $\{\text{Sanitizer}::\sim\text{Sanitizer}(), \text{Sanitizer}::\text{sanitize}()\}$.¹ The (partial) type hierarchy for the `Sanitizer` module is shown in Figure 4

¹ If a C++ type is meant to be used polymorphically, its destructor has to be declared virtual. Otherwise, if the static type of an object to be deleted differs from its dynamic type, the behavior is undefined.



■ **Figure 5** $\Pi_{\text{Sanitizer}}$ containing all pointer-assignment graphs of `Sanitizer`.

3.2.2 Intra-Procedural Points-To Information

In the next step, the analysis computes function-wise, intra-procedural, never-invalidating² points-to information using an *Andersen* [19] or *Steensgaard* [73]-style algorithm. The points-to information computed is flow-insensitive, and we store it as graphs. These function-wise pointer-assignment graphs (PAGs) are used to resolve potential call targets at dynamic call sites. We merge those intra-procedural PAGs later to obtain inter-procedural pointer information while constructing the callgraph. We use $\pi_{C::f}$ to denote a pointer-assignment graph for function f in module C . We use Π_C to denote a pointer-assignment graph containing all pointer-assignment graphs for module C .

► **Example 2.** For each function definition contained in the `Sanitizer` module a PAG is computed and added to the graph $\Pi_{\text{Sanitizer}}$. The $\Pi_{\text{Sanitizer}}$ graph containing $\pi_{\text{Sanitizer}} :: \sim\text{Sanitizer}$, $\pi_{\text{Sanitizer}} :: \text{sanitize}$, $\pi_{\text{Sanitizer}} :: \text{isMalicious}$, and $\pi_{\text{Sanitizer}} :: \text{applySanitizer}$ is shown in Figure 5. Inter-procedural points-to relations are not followed and thus, formal pointer-typed parameters and calls to functions that return a pointer value remain unresolved and represent boundaries to the respective PAG. For instance, the pointer s in the `applySanitizer()` function points to the return value of `getGlobalSan()` which is indicated by a special node in the respective PAG (cf. Figure 5d).

3.2.3 Callgraphs and Inter-Procedural Points-To Information

After having computed the function-wise pointer assignment graphs, the callgraph is constructed according to Algorithm 1, Algorithm 2 and its resolver routine shown in Algorithm 3. The same algorithm also computes points-to information across procedure boundaries. Since one cannot know upfront what library functions a user is going to call, the callgraph algorithm has to consider every externally visible function definition as a possible entry point [54] (cf. line 58 of Algorithm 1). We use CG_C to denote a (partial) callgraph of a module C . The algorithm starts at an arbitrary externally visible function f of module C . It then iterates through all call sites cs of f (cf. line 44). We denote a call site as cs_i where i represents the line number at which the call site is found. In the following, we write \overline{cs}_i for a static call site and \widetilde{cs}_i for a dynamic call site at which a function pointer or virtual function member is called. In case a static call site has been detected, the algorithm adds a new callgraph edge (line 46). In addition, for the pointer analysis, the algorithm connects the caller’s actual

² Intra-procedural points-to information is, by definition, never invalidated by additional program information from other procedures.

pointer parameters and pointer return value with their corresponding formal parameters and return value of the callee target using a *stitch* operation (line 69), thus promoting (intra-procedural) pointer information to inter-procedural information. We formally define the stitch operation in Definition 3 and then discuss its use. In the latter case (line 48), the algorithm uses points-to information provided by Π_C to resolve potential call targets of \widetilde{cs}_i according to Algorithm 3. Starting from the function pointer that is invoked or the pointer variable of the receiver that the virtual member function is being called on at \widetilde{cs}_i , we search in Π_C for reachable functions in case of function pointer calls (line 83) or allocation sites in case of virtual member function calls (line 96), respectively.

In this process, two situation may occur along with different levels of completeness of points-to information which dictate what (missing) dependencies must be tracked: *Incomplete or partially complete information*: If no functions or allocation sites are reachable yet, the reachable pointers at the function boundaries (i.e., formal pointer parameters or pointer return value of a function whose definition is missing) are marked as dependencies of \widetilde{cs}_i (line 86 and 94). The dependencies are maintained in a bidirectional map from dependent pointer parameters to the respective unresolved call site and vice versa. If only some functions or allocation sites are reachable but also there are some reachable pointers at function boundaries as well, then pointers at function boundaries are added to the dependencies of \widetilde{cs}_i and reachable functions are added as potential call targets to the callgraph (line 109 and 50). The edges of the callgraph are annotated with \widetilde{cs}_i . For virtual member function calls, the call targets of the allocated types at reachable allocation sites are inspected to find the potential targets (line 104) which are then added to the callgraph. *Complete information*: If no boundary pointers but only functions or allocation sites are reachable starting from the pointer at \widetilde{cs}_i , then no dependencies must be tracked.

During the construction of the callgraph we can have situations where a pointer-assignment graph will be amended with new information. To this end, we define a first graph operation which we call *stitch* and which we use to combine pointer information at call sites.

► **Definition 3.** *Stitch*: Let $G = (V, E)$ be a (directed) graph containing vertices $\{u, v\} \subseteq V$ with $u \neq v$ and $e = (u, v) \notin E$. The *stitch* of u and v is a new graph $G' = (V', E')$, where $V' = V$ and $E' = E \cup (u, v)$. For convenience, we additionally define the function *stitch* : $G \times G' \times P \rightarrow G''$ that maps the (directed) graphs $G = (V, E)$ and $G' = (V', E')$, and P a set of pairs of vertices (u, v) with $u \in G$ and $v \in G'$ that shall be stitched together to a new graph G'' . The *stitch* function $\text{stitch}(G, G', P)$ produces G'' such that $G'' = (V \cup V', E \cup E' \cup P)$.

For each target function $C::g$ that could be successfully resolved, the algorithm stitches \widetilde{cs}_i to $\pi_{C::g}$ (cf. line 69): Actual pointer parameters are connected with the corresponding formal parameters of the callee function $C::g$. If $C::g$ returns a pointer parameter, it is connected as well. All edges are annotated with the corresponding call site.

If this graph stitch affects a pointer that is listed in the dependency map, the algorithm recursively continues resolving the affected call sites. Otherwise, the algorithm recursively continues resolving call sites in the resolved target functions. The algorithms for the interwoven points-to, callgraph computation are shown in Algorithm 1, Algorithm 2, and Algorithm 3. We use the symbol cs in a call to the function $\text{stitch}(G, G', cs)$ as shorthand for $\{(a_i, f_i)\}$, the set of pairs of left-hand-site pointer variable/actual pointer parameters and pointer return value/formal pointer parameters of the callee at cs that are stitched together.

► **Example 4.** The callgraph algorithm starts analyzing the function `Sanitizer :: sanitize ()`. At the call site \overline{cs}_{16} , the actual parameter is stitched to the formal parameter of `Sanitizer :: isMalicious ()` and the algorithm proceeds in `Sanitizer :: isMalicious ()`. Since `Sanitizer :: isMalicious ()` has now already been visited, the next function to be analyzed is `applySanitizer ()`.

■ **Algorithm 1** Callgraph construction algorithm.

```

38 directed graph:  $CG_C = \emptyset$ ,  $T = \text{computeTypeHierarchy}()$ ; undirected graph:  $\Pi_C = \emptyset$ ;
   bidirectional map:  $D = \emptyset$ ; set:  $V = \emptyset$ ;
39 Function constructionWalk( $f$ ):
40   if  $f \in V \parallel \text{isDeclaration}(f)$  then
41     return;
42   end
43    $V \cup = f$ ;
44   foreach callsite  $cs \in f$  do
45     if  $cs$  is static then
46        $CG_C \cup = \langle cs, \text{getCallee}(cs) \rangle$ ;
47        $\text{updatePointerInfo}(f, \text{getCallee}(cs))$ ;
48     else
49        $\text{callees} = \text{resolveIndirectCallSite}(cs)$ ;
50       foreach  $callee \in \text{callees}$  do
51          $CG_C \cup = \langle cs, callee \rangle$ ;
52          $\text{updatePointerInfo}(f, callee)$ ;
53       end
54     end
55   end
56   return;
57 Function constructCallGraph():
58   foreach  $f \in C$  do
59     if  $\text{!isDeclaration}(f)$  then
60        $\Pi_C \cup = \text{computePointsToGraph}(f)$ ;
61     end
62   foreach  $f \in C \setminus \{\text{internal functions}\}$  do
63     if  $f \notin V \wedge \text{!isDeclaration}(f)$  then
64        $CG_C \cup = f$ ;
65        $\text{constructionWalk}(f)$ ;
66     end
67   return;

```

■ **Algorithm 2** Procedure for updating the pointer information.

```

68 Function updatePointerInfo( $f, callee$ ):
69    $\Pi_C = \text{stitch}(\Pi_C[f], \Pi_C[callee], cs)$ ;
70    $\text{modptrs} = \text{getVerticesInvolvedInGraphOp}(\text{stitch}, \Pi_C[f], \Pi_C[callee], cs)$ ;
71   foreach  $ptr \in \text{modptrs}$  do
72     if  $ptr \in D$  then
73        $fmod = \text{getFunctionContaining}(D[ptr])$ ;
74        $V = V \setminus fmod$ ;
75        $\text{constructionWalk}(fmod)$ ;
76     end
77    $\text{constructionWalk}(callee)$ ;
78   return;

```

$\text{applySanitizer}()$ contains two interesting call sites. \overline{cs}_{22} is a static call to $\text{getGlobalSan}()$. However, its definition is currently not available and thus, a callgraph node which is marked as a declaration is added to the callgraph. Note that the function causes incomplete points-to information as it returns a pointer value that is stored in variable s (cf. Figure 5d).

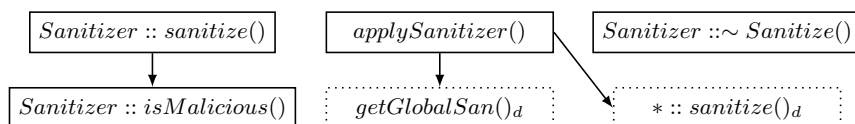
Furthermore, a virtual function member is called at \tilde{cs}_{23} on the receiver pointer variable s of type Sanitizer^* . Due to dynamic dispatch we have incomplete information on the possibly called functions and are not able to resolve this call, because we cannot yet determine the allocation sites that are reachable through s . The algorithm marks this call site as incomplete and keeps track of the dependent pointer variable s . The call site has to be updated as further information might be discovered later on. For instance, if the definition of $\text{getGlobalSan}()$ becomes available that provides the required additional points-to information. The partial callgraph that can be computed individually on the `Sanitizer` module is shown in Figure 6.

■ **Algorithm 3** Procedure for resolving dynamic call sites.

```

79 Function resolveIndirectCallSite(cs):
80   callees =  $\emptyset$ ;
81   if isFunctionPtrCall(cs) then
82     fptr = getCALLEDPtr(cs);
83     rfptrs = getReachablePtrs(fptr);
84     foreach fptr'  $\in$  rfptrs do
85       if isBoundaryPtr(fptr') then
86          $D[cs] \cup = fptr'$ ;
87       end
88     callees  $\cup =$  getReachableFunctions(fptr);
89   else
90     aptr = getAllocationPtr(cs);
91     raptrs = getReachablePtrs(aptr);
92     foreach aptr'  $\in$  raptrs do
93       if isBoundary(aptr') then
94          $D[cs] \cup = aptr'$ ;
95       end
96     allocs = getReachableAllocSites(aptr);
97     foreach alloc  $\in$  allocs do
98        $\tau =$  getAllocatedType(alloc);
99        $v_\tau =$  getVTable( $T, \tau$ );
100      if !  $v_\tau$  then
101         $D[\tau] \cup = cs$ ;
102      else
103         $i =$  getVCallIndex(cs);
104        callee = getVTableEntry( $v_\tau, i$ );
105        callees  $\cup =$  callee;
106      end
107    end
108  end
109  return callees;

```



■ **Figure 6** Callgraph for Sanitizer: $CG_{\text{Sanitizer}}$. f_d denotes the declaration of a function f .

3.2.4 Background on IFDS/IDE

To illustrate how MODALYZER summarizes data-flow information, as foundational background we first present the inherently compositional *Interprocedural Finite Distributive Subset (IFDS)* [55] and *Interprocedural Distributive Environments (IDE)* [63] frameworks that MODALYZER utilizes to solve data-flow problems.

The IFDS and IDE frameworks both follow the functional approach [66] to inter-procedural data-flow analysis. We use IFDS/IDE to encode our data-flow analyses as they allow the generation of graph-based, precise, reusable data-flow summaries of regions of code, even for incomplete code. Additionally, IFDS/IDE allow for the composition of data-flow summaries. This is required when implementing a compositional approach throughout all pieces of information where parts of the program are missing while the data-flow analysis is performed.

Reps et al. showed that distributive data-flow problems can be solved elegantly and efficiently by transforming them into graph reachability problems. The IFDS framework and its generalization IDE construct an exploded super-graph (ESG) in which each node represents a data-flow fact. If a data-flow fact d holds at a statement s the node (s, d) is

reachable in the ESG from a special tautological fact Λ (that always holds) and vice versa. The ESG is constructed by replacing each node in the inter-procedural control-flow graph (ICFG) with a bipartite graph that represents the equivalent flow function and thus, describes the effects of the statement on the data-flow facts. Standard functions for generating (Gen) or removing (Kill) data-flow facts can be encoded in bipartite graphs. Therefore, all Gen/Kill problems such as live variables, available expressions, etc. can be encoded within IFDS/IDE.

The runtime complexity of IFDS is $\mathcal{O}(|N| \cdot |D|^3)$, where $|N|$ is the number of nodes in the ICFG and $|D|$ is the size of the data-flow domain D . Thus, the efficiency highly depends on the size of the underlying data-flow domain.

In IDE, however, the data-flow domain D is decomposed into the data-flow domain D and a separate value domain V . The value domain V can be infinite. Because IDE has the same complexity as IFDS, the size of V does not affect the complexity of the algorithm. IDE annotates the edges of the ESG with lambda-functions that describe a value computation over the domain of V . When a reachability check is performed in IDE to decide whether an ESG node (s, n) is reachable and, therefore, the fact d holds at statement s , the value computation problem that is specified along those edges leading to (s, d) is solved. Figure 7 shows some exploded super-graphs for a taint analysis conducted on the code in Listing 2.

IFDS and IDE follow the functional, summary-based approach to achieve fully context-sensitive, inter-procedural analysis. The effect of statements of sections of code can be summarized by composing the flow functions of subsequent 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 each subsequent context the procedure p is called. Importantly, because the flow functions are assumed to distribute over the merge operator, this summarization is known to be *lossless* [55]. IFDS/IDE problems can thus be solved with full precision, without the need for approximation.

3.2.5 Data-Flow Information

In the next step, the analysis computes the possibly partial data-flow information using IFDS/IDE [51, 55, 63] according to the description of the data-flow problem to be solved for the available function definitions. In contrast to the information computed before, the data-flow information depends on the configuration of the client analysis because data-flow information is never general and always depending on a specific definition.

We use the flow and edge functions of the client analysis to construct the partial exploded super-graph of the library to be summarized. The partial callgraph is traversed in a depth-first bottom-up manner to maximize the number of functions that can be summarized completely. For a library function f that does not make any calls, the summary information is computed by applying the client's flow and edge functions to each node n of the control-flow graph. The resulting exploded super-graph edges are then combined using composition and meet to construct the *jump functions* $\psi(f)$ that summarize the complete function. For each incoming data-flow fact d_i , its respective jump function $\phi_i(f)$ describes the effect of the analyzed function on d_i .

In case a function f contains call sites cs_i , the IFDS/IDE algorithm computes a partial data-flow summary from f 's entry node to the first call-site node cs_1 , $\psi_{cs_1}^{entry}(f)$. It then computes the summary of the called function f' , $\psi(f')$, (if not already computed) and composes it with the partial summary $\psi_{cs_1}^{entry}(f)$ to obtain $\psi_{rs_1}^{entry}(f)$. The algorithm proceeds successively until the complete summary $\psi(f) = \psi_{exit}^{entry}(f)$ has been constructed.

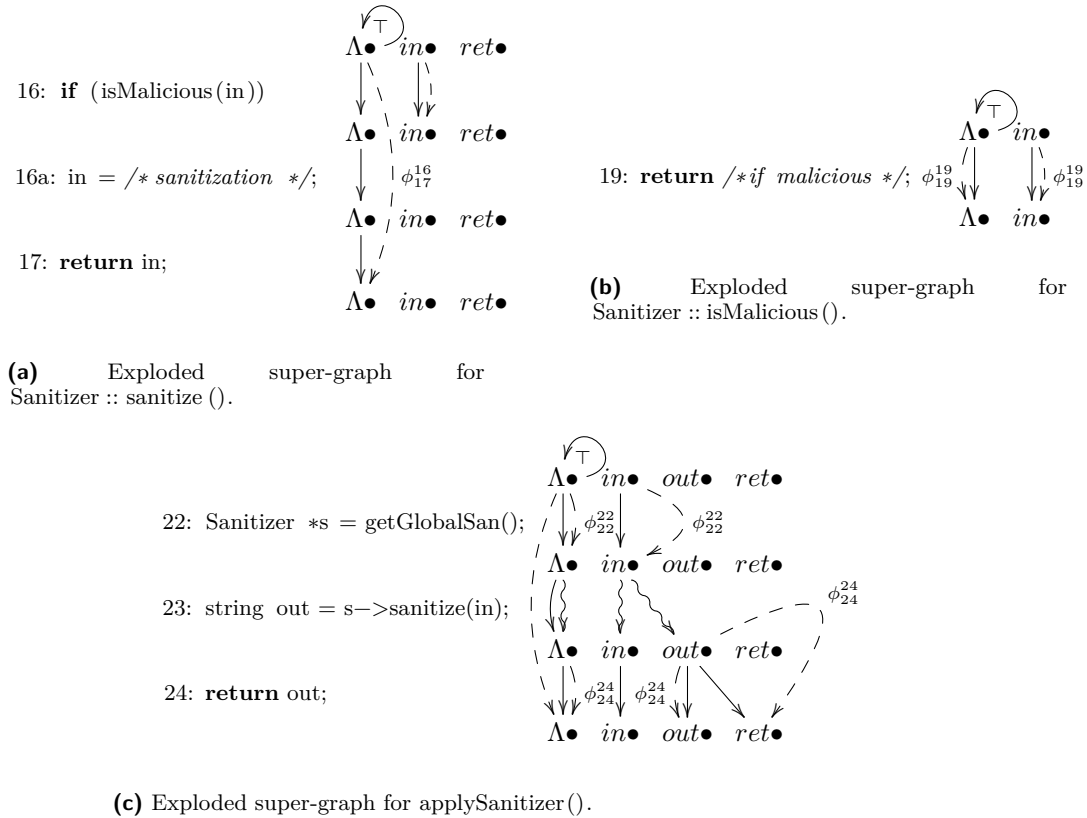


Figure 7 Exploded super-graphs for the Sanitizer module.

However, in case a library function f contains call sites that are depending on user code, for instance, because of callbacks or incomplete points-to information, a complete summary $\psi(f)$ cannot be computed. In this case, MODALYZER computes a set of partial summary functions ψ_m^n , where n is a function’s entry point or some return site (rs) and m is a function’s exit statement or some call site (cs) whose call targets are not or only partially known. This results in *gaps* in the exploded super-graph that represent the unresolved effects of the missing call targets.

► **Example 5.** The data-flow information computed for the Sanitizer module is shown in Figure 7. Individual flow/edge functions are denoted by solid (\rightarrow) and jump functions by dashed (\dashrightarrow) arrows. Analyzing applySanitizer() leads to an incomplete ESG, because the callgraph for the Sanitizer module is only partially complete. The definition of getGlobalSan() is not yet available and the dynamic call site at line 23 cannot be resolved with the information available within the Sanitizer module.

The call to the unresolved function getGlobalSan() does not interact *directly* with the data-flow information as it receives no arguments, its return type differs from the type of the data-flow domain (strings), and the string which the variable in refers to is not global as no global declarations are present. Therefore it cannot be modified by the call and one can safely use the identity function here. We will further elaborate on that in Subsection 3.4.

The call to *: sanitize () results in a *gap* in the ESG. In Figure 7c gaps in the ESG are indicated with squiggled arrows (\rightsquigarrow). We pass *in* and *out* as identity after the gap and also generate other variables, such as the implicit return variable, that depend on *out*. Later

on, after the merging process, the missing targets of the call site at line 23 will have been determined and their data-flow summaries can be inserted. Then, the analysis will check whether *in*, *out*, and *ret* are reachable from Λ , and determine if those variables are tainted.

The ESGs for the `Sanitizer::sanitize()` and `Sanitizer::isMalicious()` functions are shown in Figure 7a and 7b, respectively. For our example analysis we assume that `Sanitizer::isMalicious()` checks whether the variable in contains malicious data and the function does not modify the data-flow facts. `Sanitizer::sanitize()` checks if the string referred to by variable in contains malicious data – is tainted – and, if so, replaces it with a sanitized version. Again, to keep our example analysis simple, we assume that the analysis is aware of the special semantics of `Sanitizer::isMalicious()` and thus, kills the variable in in both branches.

After having computed the data-flow summaries for the `Sanitizer` module, we have determined any information we need on `Sanitizer` as an individual module. We denote the combination of the partial type-hierarchy graph (and call targets) in Figure 4, partial points-to in Figure 5 and callgraph in Figure 6 and the partial data-flow summaries for `Sanitizer` in Figure 7 as $\Xi_{\text{Sanitizer}}$.

3.3 Merging Analysis Summaries

To complete the picture, we next combine the information obtained by analyzing `Sanitizer` and `DbgSanitizer` with an analysis of the client application `Main`.

For this we need to define a new operation on graphs which we call *contraction*. We use the contraction operation when new information becomes available during a merge, to replace placeholder nodes (that indicate missing information) of a graph by their counterparts that represent the actual information. We apply this operation to combine partial type hierarchy- and callgraphs. For instance, we combine callgraphs by *contracting away* function declaration nodes with their respective definition counterpart nodes: the nodes representing function declarations are removed and all former incoming edges now lead to the corresponding definition nodes. We formally define the contraction operation as follows:

► **Definition 6.** *Contraction:* Let $G = (V, E)$ be a (directed) graph containing vertices $\{u, v\} \subseteq V$ with $u \neq v$. Let f be a function that maps every vertex in $V \setminus \{u, v\}$ to itself, and otherwise, maps it to a new vertex w . The contraction of u and v is a new graph $G' = (V', E')$, where $V' = (V \setminus \{u, v\}) \cup \{w\}$, $E' = E \setminus \{e = (u, v)\}$, and for every $x \in V$, the vertex $x' = f(x) \in V'$ is incident to an edge $e' \in E'$, iff the corresponding edge $e \in E$ is incident to x in G (reproduced from [53]). For convenience, we additionally define the function $\text{contract} : G \times G' \times P \rightarrow G''$ that maps the (directed) graphs $G = (V, E)$ and $G' = (V', E')$, and P a set of pairs of vertices $u \in V$ and $v \in V'$ that shall be contracted to a new graph G'' . The contraction function $\text{contract}(G, G', P)$ contracts the pairs of vertices u_i and v_i and produces a new (directed) graph $G'' = ((V \cup V') \setminus \{u_i\}, ((E \cup E') \setminus \{(t_j, u_i)\} \setminus \{(u_i, v_i)\}) \cup \{(t_j, v_i)\})$, where all edges incident to u_i with their origin in some vertex t_j are replaced by edges from t_j to v_i contracting away u_i . We use f in $\text{contract}(G, G', f)$ as shorthand for $\{(f_{\text{decl}}, f)\}$, the set of function declaration/definition pairs and τ in $\text{contract}(G, G', \tau)$ as shorthand for $\{(\tau_{\text{decl}}, \tau)\}$, the set of type declaration/definition pairs.

Our merge procedure for two module summaries Ξ_i and Ξ_j is shown in Algorithm 4. In the following, we present all involved steps for each piece of analysis information.

3.3.1 Type Hierarchies

The analysis first merges the type-hierarchy graphs using vertex contraction (cf. line 114), to remove redundant definitions of the same type. The redundancy is caused by including a type's definition (which usually resides in a corresponding header file) in multiple modules that require a type's exact data layout (e.g. for allocation or subtyping).

► **Example 7.** While performing the contraction, the analysis finds that `Sanitizer`'s type $\tau_{\text{Sanitizer}}$ is sub-typed by $\tau_{\text{DbgSanitizer}}$. The contraction has no immediate effect on the callgraph analysis: As the callgraph uses points-to information to resolve indirect calls, no immediate update is required at this point, because the new type-hierarchy information is not used before new pointer information becomes available. The type hierarchy needs to be queried if a new allocation site has been found. For each newly discovered allocation site, the type hierarchy is used to retrieve the entry of the allocated type's virtual function table.

3.3.2 Callgraphs and Points-To Information

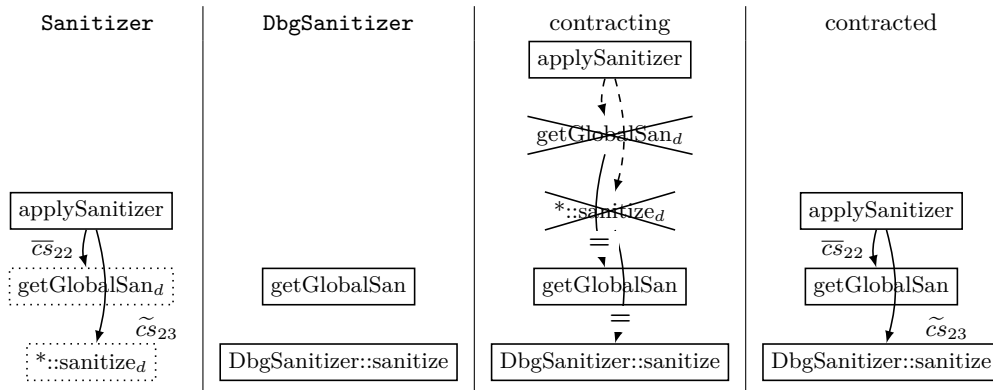
The analysis merges the callgraphs by using the vertex contraction operation introduced before (line 122). A contraction is used to remove function-declaration nodes and replace them with their corresponding definition nodes, now linking calls to callees. While performing the contraction on the callgraphs, the corresponding partial pointer-assignment graphs are not contracted but stitched together (cf. Definition 3); through the stitch (line 126) no nodes of the pointer-assignment graph are replaced to keep information on the parameter mapping. Actual pointer parameters at a call site as well as pointer return values at the respective return site are connected with the corresponding formal parameters of the called function and the left-hand side variables, respectively. The information on the contracted callgraph nodes is used in the next step when *repropagating* data-flows.

■ **Algorithm 4** Merge procedure for callgraphs.

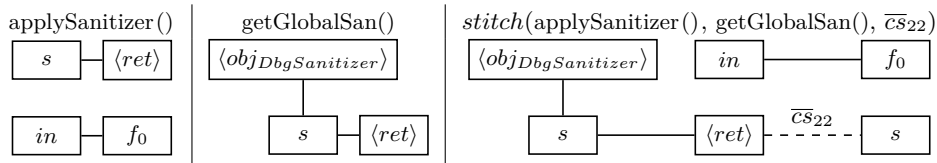
```

110 Function merge( $CG_C, T_C, \Pi_C, D_C, V_C, CG_{C'}, T_{C'}, \Pi_{C'}, D_{C'}, V_{C'},$ ):
111    $D_C \cup = D_{C'}$ ;
112    $V_C \cup = V_{C'}$ ;
113    $\Pi_C \cup = \Pi_{C'}$ ;
114    $T_C = \text{contract}(T_C, T_{C'}, \tau)$ ;
115    $modtypes = \text{getVerticesInvolvedInGraphOp}(\text{contract}, T_C, T_{C'}, \tau)$ ;
116   foreach  $\tau \in modtypes$  do
117     if  $\tau \in D$  then
118        $f = \text{getFunctionContaining}(D[\tau])$ ;
119        $V = V \setminus f$ ;
120        $\text{constructionWalk}(f)$ ;
121     end
122    $CG_C = \text{contract}(CG_C, CG_{C'}, f)$ ;
123    $\{cs, f\} = \text{getVertexPairsInvolvedInGraphOp}(\text{contract}, CG_C, CG_{C'}, f)$ ;
124   foreach  $\langle cs, f \rangle$  do
125      $f' = \text{getFunctionContaining}(cs)$ ;
126      $\Pi_C = \text{stitch}(\Pi_C[f'], \Pi_C[f], cs)$ ;
127      $modptrs = \text{getVerticesInvolvedInGraphOp}(\text{stitch}, \Pi_C[f'], \Pi_C[f], cs)$ ;
128     foreach  $ptr \in modptrs$  do
129       if  $ptr \in D$  then
130          $f = \text{getFunctionContaining}(D[ptr])$ ;
131          $V = V \setminus f$ ;
132          $\text{constructionWalk}(f)$ ;
133     end
134   end

```



■ **Figure 8** Excerpt of the vertex contraction for callgraphs of `Sanitizer` and `DbgSanitizer`. f_d denotes the declaration of a function f .



■ **Figure 9** Excerpt of the vertex stitch of the PAG's for `applySanitizer()` and `getGlobalSan()`.

► **Example 8.** The callgraph contraction of the modules `Sanitizer` and `DbgSanitizer` is indicated in Figure 8. The callgraph contraction triggers the corresponding stitching of PAGs. For instance, the points-to graphs $\pi_{\text{Sanitizer::applySanitizer}}$ and $\pi_{\text{getGlobalSan}}$ are stitched together at $\bar{c}s_{22}$ as indicated in Figure 9. Through the stitch, the analysis recognizes that the previously marked pointer variable s gets new inputs from the resolved callee function `getGlobalSan()`. As s is now able to reach `getGlobalSan()`'s variable s of allocated type $\tau_{\text{DbgSanitizer}}$ and the receiver object s in `applySanitizer()` has no other unresolved dependencies, the possible call targets are updated in the callgraph such that `DbgSanitizer::sanitize()` is now the only possible target for the dynamic call site at line 23. The pointer-assignment graph of the newly discovered callee at line 23 is stitched to the call site $\tilde{c}s_{23}$.

3.3.3 Fixed-Point Iteration for Callgraph and Points-To Graph

Note that there are cases in which the stitch (of two PAGs) of a resolved callee function changes the points-to information in such a way that previously partially resolved indirect call sites must be revised again (cf. line 69 for summarization, and line 126 for merges). In these cases, the analysis loops in updating callgraph and points-to information until the callgraph and points-to information stabilize. A constructed yet expressive example of the

```

void (*f)();
void bar() {}
void foo() { f = &bar; }
void init(void (*f)()) { f = &foo; }
int main() { init(f); f(); /* ← indirect call site */ return 0; }
    
```

■ **Listing 4** Example in which the update of points-to- invalidates callgraph information.

mentioned for function pointers is shown in Listing 4. When the callgraph algorithm resolves the indirect call to the function pointer `f` using points-to information, it determines `foo()` as the callee target. However, `foo()` manipulates the points-to information such that `bar()` becomes a feasible target as well. Thus, the indirect call site has to be revisited and `bar()` has to be added as a possible target as well. When analyzing `bar()` the callgraph and points-to information stabilize and the algorithm terminates.

3.3.4 Data-Flow Information

Once a callgraph has been updated by a merge, the data-flow information has to be repopulated in order to reflect the changes. Whenever two callgraphs are merged, new function definitions and their respective data-flow summaries become available which have been previously unknown to the other module's data-flow information. The merge procedure for the callgraphs shown in Algorithm 4 issues the contracted nodes (function declarations) and their respective call sites. This information and the newly available function definitions and accompanying data-flow summaries are used to close potential gaps in the ESG. The analysis visits all sub-graphs that have undergone the callgraph contraction procedure in a depth-first bottom-up manner, filling in the newly available data-flow summaries.

Suppose a function f contains a previously unresolved or only partially resolved call site cs and therefore, a pair of partial summaries $\psi_{cs}^{entry}(f)$ and $\psi_{exit}^{rs}(f)$. If the callgraph contraction reveals the call target f' and its respective data-flow summary, $\psi_{cs}^{entry}(f)$ and $\psi_{exit}^{rs}(f)$ are composed with $\psi(f')$ to produce a complete summary of f , $\psi_{exit}^{entry}(f) = \psi_{exit}^{rs}(f) \circ \psi(f') \circ \psi_{cs}^{entry}(f)$. The summary $\psi_{exit}^{entry}(f)$ may need to be merged with any existing jump functions that have been obtained along other paths, for instance, call-free-paths (cf. flows for Λ in Figure 7c) or paths for other call targets of cs that have been available for analysis already. The complete summary $\psi(f)$ is used to successively fill in potential other gaps in the ESG.

In case a target library to be summarized is depending on code of its user(s) because it uses features such as callbacks, for instance, the static analysis summaries Ξ even for the complete library code will contain gaps. Those gaps are eventually closed once the main application is available, analyzed and merged with the precomputed library summaries to produce the final analysis results.

► **Example 9.** As the function definition of `DbgSanitizer::sanitize()` becomes now accessible to `applySanitizer()`, its respective data-flow summary can now be plugged into the current gap of `applySanitizer()` to obtain a complete IFDS/IDE summary for it. The sub-graphs that undergo the contraction procedure are visited in a depth-first, bottom-up manner and the data-flow summary for `DbgSanitizer::sanitize()` is inserted into `applySanitizer()`. The analysis therefore finds that the values passed as a reference parameter into `DbgSanitizer::sanitize()` and the value returned by it are indeed tainted. Therefore, the return value of `applySanitizer()` is tainted as well. The pre-analysis of the library is now complete and the obtained results can be used by any potential client to the library.

3.3.5 Analyzing the Main Application

When analyzing the application program `Main` the analysis first constructs `Main`'s type hierarchy, function-wise pointer-assignment and callgraph (cf. Algorithm 1). The type hierarchy-, call- and pointer-assignment graphs for `Main` are merged with the library's respective graphs (cf. Algorithm 4). The data-flow analysis can then start at the entry point

main(). As the data-flow analysis recognizes the call to `applySanitizer()` it can directly use the (complete) pre-computed summary and thus keeps the return value as well as the actual reference parameter input marked as tainted. Finally, the client analysis is able to query the results and finds that the tainted variable `sanin` leaks at the call to `Statement::executeQuery()`.

3.4 Removing Dependencies Ahead of Time

While computing the data-flow information for an individual module, information at dynamic call sites or static call sites, where the callee definitions are not available, will be incomplete. However, by using the following shortcuts, MODALYZER is able to compute a complete and precise data-flow summary nonetheless. We already observed such a situation while computing the data-flow information for `applySanitizer()` in the `Sanitizer` module. Because the call to `getGlobalSan()` at line 22 does not have a *direct* impact on the data-flow information (as described in Example 5), we can model it using the identity flow function. Note, however, that the call still has an *indirect* impact since the function is able to change what function is being called in the next line. When our analysis recognizes a function f that misses information on potential callees, but where we can ensure that the missing information has no direct or indirect impact on the *data-flow* information, we can nevertheless compute a complete and precise summary for f using the *identity* shortcut denoted as $\overset{id}{\hookrightarrow}$ and thus fully remove any dependencies on the missing callees. To determine if $\overset{id}{\hookrightarrow}$ can be applied, different predicates may be applied, depending on the client analysis, e.g. *pass and return by value*. For instance, if a function receives its arguments *by value* they are copied into the callee. Thus, we can be sure that it cannot modify its arguments even if information on the callee's definition is missing.

```
string foo(bool p){string in = userInput(); return p ? sanitize(in) : in;}
```

■ **Listing 5** Code allowing the $\overset{\top}{\hookrightarrow}$ shortcut.

Another example of a situation in which a data-flow analysis can perform such an optimization is shown in Listing 5. Such a treatment for summarization of incomplete data-flow analysis has also been presented in [43]. While analyzing `foo()` we assume the information \top for the variable `in`, i.e., `in` is tainted. `foo()` sanitizes `in` only in one of the branches (depending on an unknown predicate). Hence, if we assume that we are conducting a may-taint analysis, then it holds that `in` may be tainted at the end of `foo()` *no matter what* the call to `sanitize()` does. It follows that \top will always be associated with `in`. In this case, we can compute a complete summary even with incomplete information by using the $\overset{\top}{\hookrightarrow}$ shortcut. This is always true for *may*-analyses that use set union as the merge operator, which for instance in IFDS is always the case.

In the presence of global variables, MODALYZER applies shortcuts *only* if they can be proven sound, which MODALYZER manages easily if only module-internal global variables are involved. Global variables are often declared as static (in case of C) or within anonymous namespaces (in case of C++) making them internal to the module that declares them. MODALYZER's shortcuts are not applied if externally visible global variables are involved in the situation, i.e., variables that are used across multiple modules.

Due to C/C++'s modular compilation model, an analysis frequently encounters situations as presented above, in which it can use these shortcuts to compute data-flow information. Functions where these shortcut summaries are used do not need to be revisited, thus, the analysis is able to work more efficiently. Therefore, when summarizing a module, it is desirable to remove as many data-flow dependencies as possible using the $\overset{id}{\hookrightarrow}$ and $\overset{\top}{\hookrightarrow}$ shortcuts.

4 Implementation

We have implemented the strategy described in Section 3 in a tool called MODALYZER, as an extension to PhASAR [64], a static-analysis framework that has been implemented on top of LLVM [45]. PhASAR allows to solve arbitrary monotone data-flow problems on the LLVM intermediate representation (LLVM IR) and also provides IFDS/IDE solver implementations.

We extended the existing IDE solver as well as the other infrastructure for type hierarchy, points-to, and callgraph computation and added the necessary summarize, merge, and update functionalities respectively.

MODALYZER persists the summary results by using a document-oriented store in which it saves the graphs along with the code the analysis is conducted on with help of LLVM’s metadata capabilities. LLVM allows for a key-based introduction of custom metadata. Each function that is defined in a module is annotated with its function-wise summaries for the different pieces of static analysis information, i.e., its points-to and exploded super-graph. A module carries the module-wise information that is obtained by merging all information of its enclosed functions as well as type hierarchy and callgraph information. Those module-wise summaries are referred to using the module flags section of the LLVM IR.

For the persistence, we created a bidirectional mapping from LLVM’s in-memory representation to a textual representation allowing us to store the graphs comprising pointer values to LLVM IR records as graphs that use the text-encoded version. Additionally, we implemented *import* and *export* functionalities for each graph type that enable us to manage loads and stores of encoded graphs along with the LLVM IR.

LLVM’s metadata mechanism does not restrict the type of data for annotations. Thus, arbitrary data structures and encodings may be used to persist the analysis information. We use the capabilities of the Boost Graph Library (BGL) [67] to manage type hierarchy, points-to, and callgraph information. The BGL offers of-the-shelf textual import and export functionalities and allows for implementing custom reader/writer concepts. We use the default Graphviz [15] format to store the graphs in metadata records. As PhASAR’s IFDS/IDE solver implementation works by incrementally constructing two tables to represent flow functions/jump functions of ever longer sequences of code (c.f. [51,64]), we use the following sets of quintuples for the data-flow summary representation of a function $\psi(f) := \{ \langle n_i, d_x, n_j, d_y, l \rangle \}$, where a quintuple represents a jump function (or an edge in the ESG) from data-flow fact d_x to d_y with the corresponding edge function l that summarizes parts of the effects of the region of code that is enclosed by the statements n_i and n_j . The concrete (partial) data-flow summary for the `applySanitizer()` function (cf. Figure 7c) looks as follows: $\{ \langle 22, \Lambda, 24, \Lambda, \top \rangle, \langle 22, in, 22, in, \top \rangle, \langle 24, in, 24, in, \top \rangle, \langle 24, out, 24, out, \top \rangle, \langle 24, out, 24, ret, \top \rangle \}$. Note that for IFDS we can use the simple encoding of the binary lattice and the edge functions. We handle the persistence of the difficult-to-handle, general IDE edge functions by creating a record to keep track which edge functions are composed and meet for each jump function while constructing them. We finally persist the record using the extensive Boost Serialization library [14]. On load, the record can be *replayed* to (re)construct the actual jump functions.

5 Experiments

Our empirical evaluation aims to answer the following research questions:

- **RQ1:** Does the use of a module-wise static analysis incur a precision loss when compared to a whole program analysis? If so, what causes this loss in precision?

- **RQ2:** Compared to conducting a whole-program analysis, what speed-up can one achieve when applying MWA using pre-computed summaries for type-hierarchy, callgraph, points-to and data-flow information?
- **RQ3:** How frequently can the data-flow shortcuts $\xrightarrow{\text{id}}$ and $\xrightarrow{\top}$ be applied in MWA?

To address **RQ1**, we compare the analysis results of a whole program analysis with the results obtained by a module-wise analysis. Ideally, the results of both analyses should be identical. To address **RQ2**, we measure and compare the runtimes of a client analysis using pre-computed summaries and a version that computes everything on-the-fly. To address **RQ3**, we extend PhASAR’s IFDS/IDE solver implementation and measure how frequently it makes use of both shortcuts for different client analyses.

5.1 Experimental Setup

We have evaluated MODALYZER using as benchmark subjects the C coreutils (version 8.28) [3] and the PhASAR framework itself.

The GNU core utilities are a collection of C programs that share a common core, providing a library that consists of 251 files. Each coreutil program itself only consists of a small number of C source files that provides the program’s entry point, manages the command-line, and makes suitable calls into the common core in order to achieve the desired task. For our evaluation we prepared and analyzed 97 of the coreutils and chose 10 of them at random which to present in this paper in more detail. (However, the figures for the remaining 87 coreutils can be found online [16].)

PhASAR is written in C++ and is similarly structured. To provide flexible, reusable software components, the main functionalities of the different components are implemented as libraries. The front-ends (or drivers) themselves represent only a relatively small amount of “glue code” and large amounts of their runtime is spent in library code. Using PhASAR we defined two benchmark subjects: First PhASAR’s own command-line client and the PhASAR-based tool *MPT*, a exemplary client that uses PhASAR as a library, both of which can be found alongside PhASAR’s examples [10].

We chose those subjects because they have a relatively high amount of virtual calls. This stresses MODALYZER’s points-to based callgraph algorithm. We observed that C++ developers generally try to minimize the amount of indirect calls to avoid indirect jumps, which degrade performance, especially when implementing performance critical software systems [17]. The chosen subjects hence set a relatively high bar when it comes to evaluating analysis performance. The raw as well as the processed data produced in our evaluation is available online [16].

All programs and their characteristics are shown in Table 1. We prepared all programs presented for analysis with the PhASAR framework by compiling them into LLVM IR with production flags using the Clang compiler. The numbers in Table 1 are based on LLVM IR.

We used an uninitialized-variables analysis \mathbb{U} and a taint analysis \mathbb{T} as two concrete client analyses that both impose the information dependencies as shown in Figure 3. \mathbb{U} and \mathbb{T} are both implemented in IFDS within PhASAR.

Uninitialized-variables analysis \mathbb{U} : \mathbb{U} is an analysis that finds potentially uninitialized variables and tracks them through the program. If the analysis finds an uninitialized variable to be read from, it reports an *illegal* use of that variable. Uninitialized variables propagate through computations and thus, the analysis tracks those as well. \mathbb{U} also tracks the variables across function boundaries making it an inter-procedural analysis.

■ **Table 1** Number of compilation units, library/application code ratio, number of statements, pointer variables and allocation sites of the analyzed (completely linked) programs.

Program	Compilation Units	$\frac{IR\ LOC\ lib}{IR\ LOC\ app}$	Statements	Pointers	Allocation Sites
wc	252	41.2	63,166	10,644	396
ls	253	5.9	71,712	13,200	438
cat	252	66.3	62,588	10,584	391
cp	256	10.5	67,097	11,722	443
whoami	252	335.7	61,860	10,433	389
dd	252	16.8	65,287	11,150	408
fold	252	105.8	62,201	10,509	390
join	252	24.9	64,196	11,042	402
kill	253	88.2	62,304	10,527	394
uniq	252	60.1	62,663	10,650	396
MPT	156	13.8	1,351,735	755,567	176,540
PhASAR (driver)	156	56.4	1,368,297	763,796	178,486

Taint analysis \mathbb{T} : \mathbb{T} is a parameterizable taint analysis that tracks tainted values through the program and reports potential leaks whenever it finds a tainted value that may flow into a *sink* function (or operation). *Sources* and *sinks* are parameterizable. We used PhASAR’s default parametrization that treats the command-line arguments passed into `main` as tainted. All standard *input* functions (e.g., `fread()`, `fgets()`) are treated as *sources* as well. All *output* functions (e.g., `fwrite()`, `printf()`) are treated as *sinks*.

For each target program shown in Table 1 we computed the library and application code ratio based on lines of LLVM IR code. If a module is used by more than one application, we consider it to be part of the library, whereas modules that are only used by one application are considered as application code. We also measured runtimes and number of leaks/uninitialized variables that each of the analyses reported in a WPA setup as well as an MWA setup. The measurements for MWA are split into a summarization and an actual analysis step. The PhASAR framework implements a reporting system which we use to compare the actual reports to make sure that the findings are identical. We also recorded the number of callgraph updates $\#CG \odot$ that had to be performed in the MWA setup, i.e., we counted the number of callgraph edges that have been introduced during the merge process. This is a good indicator of the expense of a merge, as the introduction of a new callgraph edge causes the points-to and data-flow information to be updated as well. In addition, we measured the number of shortcuts that a data-flow analysis was able to use. We measured the runtimes by performing 5 runs for each analysis in each setup on a virtual machine running on an Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz machine with 128GB memory. We removed the minimum and maximum values and computed the average of the remaining 3 values. Table 2 shows the results. The first column comprises the programs under analysis, the second column contains the WPA runtimes, column three contains the required runtime for summarization, column four the actual analysis time of MWA. The differences of the runtimes and reports of WPA and MWA are shown in column five. Column six, seven, and eight contain the respective number of callgraph updates, identity shortcuts, and \mathbb{T} shortcuts, respectively. The number of callgraph updates are equal for both analysis as the callgraph information is not affected by the concrete client analyses.

■ **Table 2** Runtimes and findings WPA vs. MWA for the taint analysis \mathbb{T} (first half) and uninitialized variables \mathbb{U} (second half).

\mathbb{T} : Program	WPA [s]	$\Sigma_{m \in lib}$ [s]	MWA [s]	Δ runtimes / (Δ reports)	$\#\overset{CG}{\circ}$	$\#\overset{id}{\leftrightarrow}$	$\#\overset{\mathbb{T}}{\leftrightarrow}$
wc	2.3	5.7	0.5	-1.8 / (0)	47	8,052	78
ls	4.8	5.7	1.3	-3.5 / (0)	166	13,470	11
cat	1.9	5.7	0.2	-1.7 / (0)	21	2,117	269
cp	4.4	5.7	1.8	-2.6 / (0)	197	19,712	1077
whoami	2.0	5.7	0.4	-1.6 / (0)	4	6,065	11
dd	8.1	5.7	5.5	-2.6 / (-3)	58	48,747	90
fold	2.1	5.8	0.4	-1.7 / (0)	12	6,695	11
join	2.4	5.7	0.6	-1.8 / (0)	58	8,979	11
kill	1.9	5.7	0.2	-1.7 / (0)	14	2,079	11
uniq	2.2	5.7	0.4	-1.8 / (0)	29	7,281	11
MPT	2,306	42,847	1,516	-809 / (0)	41	29,061	0
PhASAR	7,176	42,876	598	-6578 / (0)	3	47,736	0
\mathbb{U} : Program	WPA [s]	$\Sigma_{m \in lib}$ [s]	MWA [s]	Δ runtimes / (Δ reports)	$\#\overset{CG}{\circ}$	$\#\overset{id}{\leftrightarrow}$	$\#\overset{\mathbb{T}}{\leftrightarrow}$
wc	2.6	5.9	0.6	-2.0 / (0)	47	2,413	162
ls	8.4	6.0	3.3	-5.1 / (0)	166	7,173	184
cat	2.0	6.0	0.3	-1.7 / (0)	21	845	12
cp	5.2	5.9	2.2	-3.0 / (0)	197	6,684	1122
whoami	2.0	5.9	0.3	-1.7 / (0)	4	535	0
dd	3.1	5.9	0.9	-2.2 / (0)	58	2,522	16
fold	2.1	6.0	0.4	-1.7 / (0)	12	895	0
join	2.8	6.0	0.5	-2.3 / (0)	58	2,582	171
kill	2.2	6.0	0.4	-1.8 / (0)	14	793	12
uniq	2.5	5.9	0.5	-2.0 / (0)	29	1,433	17
MPT	3,811	53,703	2,958	-826 / (0)	41	137,722	8,136
PhASAR	10,160	53,348	968	-9,192 / (0)	3	210,032	24,446

5.2 RQ1: Precision

As the points-to and therefore, call- and control-flow graphs guide an analysis through a program, they may heavily influence the reported results. Therefore, we compared the callgraph obtained in an MWA setting with the one obtained in a WPA setting. We found that the callgraphs only differ at call-sites at which a static function pointer is called. In those cases, our MWA callgraph implementation turns out to be *more* precise as it does not consider every function of the complete program that matches the pointer’s signature as a possible target, but only the ones reachable within the module whose address can actually be taken.³ This reduces the number of infeasible call targets while retaining soundness.

We compared the client analyses precision and recall of WPA and MWA using PhASAR’s reporting capabilities. Column Δ in Table 2 shows how many result entries differ from a WPA to an MWA setup for each client analysis. We only observed a difference in the reports for the “dd” program while performing the taint analysis. In this case, the analysis in WPA mode reports three leaks in a library function f_L , whereas the analysis in MWA reports none. We investigated the cause of this difference and found that this is actually a false positive in the WPA. The leaking function f_L is not called within the “dd” program. However, “dd” defines a static global function pointer p in the application code and the WPA analysis safely

³ Reducing the set of feasible function pointer targets in WPA mode can be easily implemented.

assumes that f_L , which matches the function pointers signature, might be called. When the application code that defines the static function pointer is analyzed in MWA mode, the analysis does not find a declaration of f_L within the application code and therefore, its address cannot possibly be taken, preventing it to be a callee target of p . While one could adapt the WPA to be equally precise, the MWA obtains this precision automatically.

Since MODALYZER does not need to overapproximate information it does indeed also preserve recall. The MODALYZER approach has been designed to obtain this property by construction. Besides the differing result entries that are caused by the differences in the callgraph, both the results of MODALYZER and WPA coincide.

The module-wise analysis generally yields the same precision as the whole-program analysis, in some cases even exceeds it.

5.3 RQ2: Performance

Table 1 shows that the library/application ratio ranges from 5.9 to 5675.6 and therefore, that the actual application code only comprises a small fraction of the complete program. One expects the MWA runtime to pay off better with increasing code ratios, since more pre-computed summaries can be (re)used for a program’s library parts. The runtimes of both analyses measured in the WPA and MWA setup live up to that expectation. Looking at the programs with an especially advantageous library/application ratio such as whoami, fold, kill, cat, PhASAR, the use of pre-computed summaries saves between 81% and 91% of the analysis time. On average, MWA saves 72% of analysis time compared to WPA while MWA’s initial one-time summarization step is, on average, 3.67 times as expensive than the corresponding run in a WPA setup. Thus, computing the initial summarization of the library (or infrequently changing) parts of a program is more expensive than performing a whole program analysis. Computing summaries will always be more expensive compared to computing plain WPA due to the additional overhead required for organizing and maintaining the summaries. In addition, many of PhASAR’s critical analysis parts have undergone tremendous amounts of manual optimization while MODALYZER’s implementation for summary generation has not yet been optimized manually. As a concrete example, analyzing PhASAR in an MWA setup outperforms WPA with the seventh run using the taint analysis and after the sixth run for the uninitialized variables analysis – assuming an initial summary must be computed and no changes in PhASAR’s library occur after summarization. For the MPT program, that has a larger number of callgraph updates to be performed, MWA pays off with the 54th run for the taint analysis and 64th run for the uninitialized variables analysis, respectively.

In case of PhASAR, runtime savings of 92% can be achieved as the application merely consists of few calls into the library code. This is underlined by the three callgraph updates that are necessary. We manually inspected the program and confirmed that, although the amount of front-end code is certainly large, it performs only very few calls into the corresponding library. A controller class, which is part of the library, is used to dispatch the different tasks to solve into calls to the adequate library functionalities. This shifts large parts of the computation to the offline MWA summarization phase.

The size of the persisted summaries that are stored along with the library code increase a library, on average, by a factor of five in size. The code and summaries for PhASAR require approx. 2.8 GB of memory for persistence and 30 MB for the core utils.

Summaries for static callgraph, points-to and data-flow analysis can be used to capture the analysis effects of libraries. After a one-time pre-computation effort, this allows a runtime reduction of 72%, on average, compared to the runtimes in whole-program mode.

5.4 RQ3: Shortcuts

The number of $\overset{\text{id}}{\hookrightarrow}$ shortcuts taken by an analysis is parameterized by a predicate as described in Subsection 3.4. For the analyses \mathbb{U} and \mathbb{T} we used the predicate *return type is void and uses pass-by-value*. However, different predicates might be useful for other analyses, depending on the specific assumptions that can be made on an analysis’s domain. The results in Table 2 show that both shortcuts can be frequently applied during analysis. The $\overset{\text{id}}{\hookrightarrow}$ shortcut can be applied between 535 and 210,032 times depending on the client data-flow analysis that is performed. The $\overset{\mathbb{T}}{\hookrightarrow}$ shortcut can be applied between 0 and 24,446 times. We are confident that the number of $\overset{\mathbb{T}}{\hookrightarrow}$ shortcuts could be further increased, if one adjusts PhASAR’s data-flow solvers to favour analyzing branches first that contains fewer (or no) function calls.

Shortcuts can be frequently applied. Hence, to decrease the number of data-flow dependencies and to increase the amount of complete summaries that can be pre-computed offline, it is advisable to make use of shortcuts whenever possible.

6 Limitations of the Approach

In this section, we briefly discuss the limitations of MODALYZER. MODALYZER needs to summarize the different pieces of information presented in Figure 3 to be able to construct effective module-wise summaries for a given concrete client analysis. Hence, MODALYZER requires analysis algorithms that produce summarizable results such as IFDS [55], IDE [63] or *Weighted Pushdown Systems* (WPDS) [56].

For problems that are distributive, hence fit into these frameworks, the summarization is lossless. It is generally also possible to use MODALYZER to solve non-distributive client analysis problems. As mentioned in Section 1, one cannot generally compute summaries for non-distributive data-flow problems. In that case, the approach can only make use of the summaries for type-hierarchy, points-to, and callgraph information, which may still lead to large performance increases as we present in Section 5.

We use never-invalidating points-to information computed using an *Andersen* [19] or *Steensgaard*-style [73] algorithm to be able to produce effective summaries. Again, computing more precise inter-procedural, context-, and flow-sensitive points-to information is a non-distributive problem for which no effective summaries can be computed. However, Späth et al. showed how flow- and context-sensitive pointer analysis can be decomposed into multiple analysis problems each of which, in turn, can be expressed within a distributive framework [72] – making the overall problem distributive. MODALYZER’s current points-to algorithm could therefore also be replaced by an adjusted version the distributive BOMMERANG approach proposed by Späth et al. The BOOMERANG approach – as is – operates in an on-demand manner and does not compute reusable summaries nor does it persist results. It is interesting to see the performance of MODALYZER with an improved BOOMERANG-style points-to algorithm, that reuses summaries, presented in [72], but we consider it as future work.

As described in Section 5, MODALYZER’s overall effectiveness degrades with the number of updates that must be performed while merging summaries with the application code. Therefore, MODALYZER’s performance increase may not apply to programs that make excessive use of callbacks.

7 Related Work

Several previous approaches address, in part, the difficult problem of compositional static analysis [30, 31, 33, 36, 38, 52, 59–61, 78, 83]. However, existing techniques for compositional static analysis typically focus on data-flow or points-to analysis only. As advocated in this paper, a concrete compositional data-flow analysis client requires at the very least a combination of compositional callgraph, points-to and data-flow analysis.

Compositional data-flow techniques rely on the functional approach [66] allowing to solve distributive data-flow problems by using summary-based, inherently compositional frameworks such as IFDS [55], IDE [63], or WPDS [56]. Rountev et al. used IDE data-flow summaries to summarize large object-oriented libraries [62] and showed that a significant amount of time can be saved when using pre-computed summaries. The approach presented by Rountev et al., however, omits to tackle the challenging task of persisting general IDE summaries but rather discards the summaries at analysis shutdown. STUBDROID [21] is a fully automated approach to generate precise library models for taint-analysis problems for the Android Framework, effectively preventing the re-analysis of the Android Framework for the analysis of different Android apps. Both Rountev’s approach and StubDroid assume the existence of whole-program points-to and callgraph information.

Several works use partial points-to information in from of function-local summaries computed using context-free language (CFL-)reachability [48, 65, 81]. The summaries can be used in various scenarios allowing, among others, for on-demand points-to analysis, pre-analysis, and pointer analysis of partial programs using different sensitivities. These works present individual solutions to individual problems, while this paper presents the first integrated approach and shows its effectiveness on real-world C/C++ applications.

The IDEal [71] approach developed by Späth et al. is an alias-aware extension to the framework IDE [63] framework. IDEal embeds the alias analysis BOOMERANG [72] into the IDE solver implementation HEROS [25] to automatically resolve alias queries on-demand at analysis time while solving a given distributive data-flow analysis problem. However, it does not compute (persisted,) reusable summaries but rather computes analysis queries on-demand and still requires external callgraph information.

AVERROES [18] uses the *separate compilation assumption* and Java’s constant pool [9] to generate sound and precise callgraphs without actually analyzing library code in order to generate a placeholder library. Existing whole-program callgraph construction algorithms can use the replacement to obtain a sound and precise application callgraph. AVERROES supports callgraph construction only. Its summaries cannot be used for precise pointer analysis, nor for precise data-flow analysis.

Other techniques try to improve the scalability of inter-procedural static analysis by using sparse propagation of data-flow facts along def-use chains [77] or demand-driven analysis that only analyze parts of a program that a user is currently interested in [72, 76]. Sparseness is a concept orthogonal to the ones proposed here. Both could be used in combination.

Some tools, including clang-tidy [2] and CppCheck [5], trade off scalability for reduced complexity. Thus, they only apply syntactic analysis to retrieve information on the property of interest. Precise, fully-fledged static analysis is replaced by much simpler checks that are capable of analyzing even million lines of code in minutes. However, these checks are often too imprecise to check for interesting properties.

Klohs et al. described the situation for *may*-analysis in which \top , representing all information, is obtained along one path in the control-flow graph, and thus, the other path does not have to be analyzed. This allows to remove data-flow dependencies ahead of time [43]. The approach presented here adopts this insight.

MODALYZER computes the module-level summaries in a completely unrestricted way and does not make any assumptions about missing code. Yet, it may be advisable to compute summaries based on various sensible assumptions in scenarios where the summarization step can be performed ahead of time, e.g. for library pre-analysis. Tree-adjointing languages [79] and Dyck context-free language reachability [30, 78] can be used to increase the effective library summarization by computing reasonable conditional summaries that enable greater summary reuse under certain premises checked at analysis time of the application code. Such a strategy allows for more computations to be performed on a module-level. During the merge, the analysis can check whether an assumption that has been made holds and, if so, directly use the corresponding summary that may be much more expressive than one that has been computed without any assumptions about missing code, effectively reducing the amount of work that needs to be done while merging summaries with the application code. MODALYZER currently does not use such a conditional summarization, however, it provides all required infrastructure to easily integrate the approach. Unfortunately, one cannot rely on programmers specifying pointer or reference parameters as constants using the `const` keyword because C/C++’s typesystem provides several mechanisms to circumvent constant declarations (e.g. `const_cast` and `mutable` in case of C++). Although writes through `const` are possible, they are used sparingly in real-world software as shown by Eyolfson and Lam [35]. Therefore, one reasonable assumption may be *const means const*. Especially `const`-qualified pointer parameters then represent hard inter-procedural boundaries and a data-flow analysis is not concerned with those parameters.

Early versions of Facebook’s Infer [27] used separation logic to allow for the compositional analysis of heap-based programs. The approach computed bottom-up summaries using bi-abductive inference [24, 28], which could then be used in different calling contexts. Using Infer, one could thus formulate compositional static analyses that are evaluated using abstract interpretation. These analyses, however, were largely restricted to finding cases of memory corruption. Since about 2019 – reportedly due to a lack of general applicability and extensibility – Infer thus does not use abductive inference for most of its analyses any longer, and now instead bases its implementation on data-flow analysis using abstract interpretation. This analysis is no longer compositional.

8 Conclusion

In this paper, we presented MODALYZER, a compositional approach to speeding up static analysis using persisted summaries for callgraph, points-to and data-flow information. We have presented an integrated strategy based on the dependencies as shown in Figure 3 that manages all those information and their dependencies, which many useful, concrete client analyses impose to provide precise results. MODALYZER allows one to compute static analysis summaries on individual parts of a program without the need to make any assumptions on the missing code. These pre-computed summaries can then be (re)used later on, effectively shifting large parts of the computational effort to an offline phase.

Our experiments confirm the finding by previous works that actual application code often only constitutes only a small fraction of the complete program. Thus, MODALYZER outperforms traditional whole program analysis in both runtime and flexibility.

References

- 1 C++ applications, December 2018. URL: <http://www.stroustrup.com/applications.html/>.
- 2 clang-tidy, August 2018. URL: <http://clang.llvm.org/extra/clang-tidy/>.

- 3 coreutils, July 2018. URL: <https://www.gnu.org/software/coreutils/coreutils.html>.
- 4 Coverity static application security testing (sast), December 2018. URL: <https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html>.
- 5 Cppcheck, August 2018. URL: <http://cppcheck.sourceforge.net/>.
- 6 Gcc optimize options, December 2018. URL: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>.
- 7 Grammatech codesonar, 2018. URL: <https://www.grammatech.com/products/codesonar>.
- 8 Intel® c++ compiler 19.0 developer guide and reference: Interprocedural optimization (ipo), December 2018. URL: <https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-interprocedural-optimization-ipo>.
- 9 Java virtual machine specification: The constant pool, December 2018. URL: <https://docs.oracle.com/javase/specs/jvms/se10/html/jvms-4.html#jvms-4.4>.
- 10 Phasar, July 2018. URL: <https://phasar.org>.
- 11 The programming languages beacon, December 2018. URL: <http://www.lextrait.com/vincent/implementations.html/>.
- 12 The state of open source security, December 2018. URL: <https://snyk.io/stateofossecurity/>.
- 13 Thinlto: Scalable and incremental lto, July 2018. URL: <http://blog.llvm.org/2016/06/thinlto-scalable-and-incremental-lto.html>.
- 14 Boost.serialization, August 2019. URL: https://www.boost.org/doc/libs/1_70_0/libs/serialization/doc/.
- 15 Graphviz, August 2019. URL: <https://www.graphviz.org/>.
- 16 Supplementary material, 2019. URL: <https://drive.google.com/drive/folders/1uLHDkmdWdjQ-aeZjyRizhy9zwrX4VWVo?usp=sharing>.
- 17 Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in c++ programs. In Pierre Cointe, editor, *ECOOP '96 — Object-Oriented Programming*, pages 142–166, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
- 18 Karim Ali and Ondřej Lhoták. Averroes: Whole-program analysis without the whole program. In *Proceedings of the 27th European Conference on Object-Oriented Programming, ECOOP'13*, pages 378–400, Berlin, Heidelberg, 2013. Springer-Verlag. doi:10.1007/978-3-642-39038-8_16.
- 19 Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. Datalogisk Institut, Københavns Universitet, 1994.
- 20 Steven Arzt and Eric Bodden. Reviser: Efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pages 288–298, New York, NY, USA, 2014. ACM. doi:10.1145/2568225.2568243.
- 21 Steven Arzt and Eric Bodden. Stubdroid: Automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 725–735, New York, NY, USA, 2016. ACM. doi:10.1145/2884781.2884816.
- 22 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 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*, pages 259–269, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594299.
- 23 Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010. doi:10.1145/1646353.1646374.
- 24 Dirk Beyer, Sumit Gulwani, and David A Schmidt. Combining model checking and data-flow analysis. In *Handbook of Model Checking*, pages 493–540. Springer, 2018.

- 25 Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 3–8, New York, NY, USA, 2012. ACM. doi:10.1145/2259051.2259052.
- 26 Eric Bodden. The secret sauce in efficient and precise static analysis. In *Proceedings of the 7th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2018, 2018. To appear.
- 27 Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.
- 28 Cristiano Calcagno, Dino Distefano, Peter O’Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 289–300, New York, NY, USA, 2009. ACM. doi:10.1145/1480881.1480917.
- 29 Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in c++ programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 397–408, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/174675.177973.
- 30 Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. Optimal dyck reachability for data-dependence and alias analysis. *Proc. ACM Program. Lang.*, 2(POPL):30:1–30:30, 2017. doi:10.1145/3158118.
- 31 Michael Codish, Saumya K. Debray, and Roberto Giacobazzi. Compositional analysis of modular logic programs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 451–464, New York, NY, USA, 1993. ACM. doi:10.1145/158511.158703.
- 32 Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in c++. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, page 306–323, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/236337.236369.
- 33 M. B. Dwyer. Modular flow analysis for concurrent software. In *Proceedings of the 12th International Conference on Automated Software Engineering (Formerly: KBSE)*, ASE '97, pages 264–, Washington, DC, USA, 1997. IEEE Computer Society. URL: <http://dl.acm.org/citation.cfm?id=786767.786816>.
- 34 Michael Eichberg, Ben Hermann, Mira Mezini, and Leonid Glanz. Hidden truths in dead software paths. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 474–484, New York, NY, USA, 2015. ACM. doi:10.1145/2786805.2786865.
- 35 Jon Eyolfson and Patrick Lam. C++ const and Immutability: An Empirical Study of Writes-Through-const. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2016.8.
- 36 Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. Field analysis: Getting useful and low-cost interprocedural information. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 334–344, New York, NY, USA, 2000. ACM. doi:10.1145/349299.349343.
- 37 Mark Harman and Peter O’Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23. IEEE, 2018.
- 38 Mary Jean Harrold and Gregg Rothermel. Separate computation of alias information for reuse. *IEEE Trans. Softw. Eng.*, 22(7):442–460, July 1996. doi:10.1109/32.538603.
- 39 Ben Hermann, Michael Reif, Michael Eichberg, and Mira Mezini. 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, pages 758–769, New York, NY, USA, 2015. ACM. doi:10.1145/2786805.2786829.
- 40 P. Holzinger, B. Hermann, J. Lerch, E. Bodden, and M. Mezini. Hardening java’s access control by abolishing implicit privilege elevation. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1027–1040, May 2017. doi:10.1109/SP.2017.16.
 - 41 John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.
 - 42 Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL ’73, pages 194–206, New York, NY, USA, 1973. ACM. doi:10.1145/512927.512945.
 - 43 Karsten Klohs. A summary function model for the validation of interprocedural analysis results. In *Proceedings of the 7th International Workshop on Compiler Optimization meets Compiler Verification, COCV’08*, 2008.
 - 44 Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. Cognicrypt: Supporting developers in using cryptography. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, pages 931–936, Piscataway, NJ, USA, 2017. IEEE Press. URL: <http://dl.acm.org/citation.cfm?id=3155562.3155681>.
 - 45 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO ’04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. URL: <http://dl.acm.org/citation.cfm?id=977395.977673>.
 - 46 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 129–140, New York, NY, USA, 2018. ACM. doi:10.1145/3236024.3236041.
 - 47 V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM’05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1251398.1251416>.
 - 48 Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with cfl-reachability. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC’13, pages 61–81, Berlin, Heidelberg, 2013. Springer-Verlag. doi:10.1007/978-3-642-37051-9_4.
 - 49 Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005.
 - 50 Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O’Reilly Media, Inc., 1st edition, 2014.
 - 51 Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the ifds algorithm. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC’10/ETAPS’10, pages 124–144, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-11970-5_8.
 - 52 Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In Ole Lehrmann Madsen, editor, *ECOOP ’92 European Conference on Object-Oriented Programming*, pages 329–349, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.
 - 53 Santanu Saha Ray. *Graph Theory with Algorithms and Its Applications: In Applied Science and Technology*. Springer Publishing Company, Incorporated, 2014.
 - 54 Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International*

- Symposium on Foundations of Software Engineering*, FSE 2016, pages 474–486, New York, NY, USA, 2016. ACM. doi:10.1145/2950290.2950312.
- 55 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM. doi:10.1145/199448.199462.
 - 56 Thomas Reps, Stefan Schwoon, and Somesh Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 189–213, Berlin, Heidelberg, 2003. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=1760267.1760283>.
 - 57 H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 1953, 74, 2, 358, 1953.
 - 58 Personal communication with atanas (nasko) rountev, 2014.
 - 59 A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, June 2004. doi:10.1109/TSE.2004.20.
 - 60 Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In Reinhard Wilhelm, editor, *Compiler Construction*, pages 20–36, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
 - 61 Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, pages 235–252, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
 - 62 Atanas Rountev, Mariana Sharp, and Guoqing Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 53–68, Berlin, Heidelberg, 2008. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=1788374.1788380>.
 - 63 Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1-2):131–170, 1996. doi:10.1016/0304-3975(96)00072-2.
 - 64 Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for c/c++. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–410, Cham, 2019. Springer International Publishing.
 - 65 Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 264–274, New York, NY, USA, 2012. ACM. doi:10.1145/2259016.2259050.
 - 66 M Sharir and A Pnueli. *Two approaches to interprocedural data flow analysis*. New York Univ. Comput. Sci. Dept., New York, NY, 1978. URL: <https://cds.cern.ch/record/120118>.
 - 67 Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library - User Guide and Reference Manual*. C++ in-depth series. Pearson / Prentice Hall, 2002.
 - 68 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM. doi:10.1145/1926385.1926390.
 - 69 Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594320.
 - 70 Black Duck Software. 2018 open source security and risk analysis. <https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2018-ossra.pdf>, 2018.

- 71 Johannes Späth, Karim Ali, and Eric Bodden. Ideal: Efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133923.
- 72 Johannes Späth, Lisa Nguyen, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In *European Conference on Object-Oriented Programming (ECOOP)*, 17 - 22 July 2016.
- 73 Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM. doi:10.1145/237721.237727.
- 74 R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986. doi:10.1109/TSE.1986.6312929.
- 75 Robert E. Strom. Mechanisms for compile-time enforcement of security. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 276–284, New York, NY, USA, 1983. ACM. doi:10.1145/567067.567093.
- 76 Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 460–473, New York, NY, USA, 2016. ACM. doi:10.1145/2950290.2950296.
- 77 Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 265–266, New York, NY, USA, 2016. ACM. doi:10.1145/2892208.2892235.
- 78 Hao Tang, Di Wang, Yingfei Xiong, Lingming Zhang, Xiaoyin Wang, and Lu Zhang. Conditional dyck-cfl reachability analysis for complete and efficient library summarization. In Hongseok Yang, editor, *Programming Languages and Systems*, pages 880–908, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
- 79 Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 83–95, New York, NY, USA, 2015. ACM. doi:10.1145/2676726.2676997.
- 80 John Toman and Dan Grossman. Taming the Static Analysis Beast. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.SNAPL.2017.18.
- 81 John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, page 187–206, New York, NY, USA, 1999. Association for Computing Machinery. doi:10.1145/320384.320400.
- 82 Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 1–17, London, UK, UK, 2000. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=647476.760384>.
- 83 Jingling Xue and Phung Hua Nguyen. Completeness analysis for incomplete object-oriented programs. In *International Conference on Compiler Construction*, volume 3443, pages 271–286, April 2005. doi:10.1007/978-3-540-31985-6_21.