Position Paper: Static Flow-Sensitive & Context-Sensitive Information-flow Analysis for Software Product Lines [*]

Eric Bodden

Secure Software Engineering Group
European Center for Security and Privacy by Design (EC SPRIDE)
Technische Universität Darmstadt
Darmstadt, Germany
eric.bodden@ec-spride.de

Abstract

A software product line encodes a potentially large variety of software products as variants of some common code base, e.g., through the use of `#ifdef` statements or other forms of conditional compilation. Traditional information-flow analyses cannot cope with such constructs. Hence, to check for possibly insecure information flow in a product line, one currently has to analyze each resulting product separately, of which there may be thousands, making this task intractable.

We report about ongoing work that will instead enable users to check the security of information flows in entire software product lines in one single pass, without having to generate individual products from the product line. Executing the analysis on the product line promises to be orders of magnitude more faster than analyzing products individually.

We discuss the design of our information-flow analysis and our ongoing implementation using the IFDS/IDE framework by Reps, Horwitz and Sagiv.

Categories and Subject Descriptors D.2.0 [*Software Engineering*]: Protection Mechanisms

General Terms Design, Languages, Security, Verification

Keywords Information flow control, static analysis, software product lines, IFDS, IDE

[*] This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLAS'12 June 15, Beijing, China.
Copyright © 2012 ACM ISBN 978-1-4503-1441-1/12/06...$10.00

1. Introduction

A Software Product Line (SPL) describes a set of software products as variations of a common code base. Variations, so-called features, are typically expressed through compiler directives such as the well-known `#ifdef` from the C preprocessor or other means of conditional compilation. SPLs have become quite popular in certain application domains, for instance the development of games and other applications for mobile devices: depending on the hardware capabilities of a certain mobile device, it may be advisable or not to include certain features in a software product for that device, or to include a variety of a given feature.

In the area of software security, a more common problem is probably that programmers use `#ifdef` directives when programming security primitives in languages such as C. While not being product lines in the strict sense, such programs with conditional compilation can also be seen as product lines for the purpose of this paper. Our approach applies to such programs in the same way as it applies to software that was specifically designed as an SPL.

The major problem with product lines is that due to their built-in variety, SPLs can induce a large number of resulting products: a product line with n optional features can generally induce up to $p = 2^n$ different products, as each feature may be enabled or not.[1] When performing static analyses such as an information-flow analysis on SPLs, this is problematic. Existing information-flow analyses cannot cope with the conditional-compilation constructs in the SPL definition, and hence can only be applied to the resulting product after the pre-processor has been applied. For p different products this would imply p different analysis runs, even though typically all products have much in common and do not vary significantly.

In this paper we present ongoing work on designing an information-flow analysis that will allow users to instead

[1] Some combinations may be illegal, and hence SPLs usually come with a so-called feature model that describes the allowed combinations.

analyze an entire software product line at once, including all the possible products. In all cases where a traditional information-flow analysis would report that for some product p a property violation exists at a given statement s, our SPL-based analysis will not only report that such a violation may exists but also for which feature combinations. A result $(F \wedge G) \vee \neg H$, for instance, signifies that the violation may exists if feature F is enabled as well as feature G or if feature H is disabled. Since our approach analyzes the entire product line at once, it promises cost savings of up to several orders of magnitude compared to the traditional approach, which comes at a cost exponential in the number of features.

Our approach is based on the IFDS framework for inter-procedural, finite, distributive subset problems by Reps, Horwitz and Sagiv [9]. In IFDS, inter-procedural data-flow analysis problems are reduced to a graph reachability problem. Many program analyses can be expressed using IFDS, and information-flow analysis is particularly amenable to the IFDS framework. In this work we show how an information-flow analysis expressed in the IFDS framework can be lifted to entire software product lines by treating the analysis as an instance of the more general IDE framework [10]. IDE is an extension of IFDS that allows distributive functions to be computed along the graph edges that IFDS operates on. In our setting, those functions compute feature constraints.

We are currently implementing our information-flow analysis on top of an IDE solver that we wrote ourselves, as an extension to the Soot program analysis toolkit [8]. Our implementation can treat all software product lines developed in the Java-based Color IDE (CIDE) [5].

To summarize, this paper presents the following original contributions:

- A mechanism for static information-flow analysis on software product lines and other applications that use conditional compilation.
- The sketch of an implementation based on Soot and CIDE.

The remainder of this paper is structured as follows. Section 2 introduces a small running example. In Section 3, we discuss the design of our information-flow analysis for software product lines. Section 4 discusses our implementation and Section 5 related work. We discuss our conclusions in Section 6.

2. Example

We show a simplified running example in Figure 1. In this example there are multiple different value assignments, depending on whether or not the features F, G or H are enabled when a product is generated from the product line. We assume that the task of our information-flow analysis is to report a property violation whenever a value returned from `secret()` is passed to the function `print`. As the observant reader will notice, in the example product line from

```
1  void main() {
2    int x = secret();
3    int y = 0;
4    #ifdef F
5      x = 0;
6    #endif
7    #ifdef G
8      y = foo(x);
9    #endif
10   print(y);
11 }
12
13 int foo(int p) {
14   #ifdef H
15     p = 0;
16   #endif
17   return p;
18 }
```

(a) Example SPL

```
1  void main() {
2    int x = secret();
3    int y = 0;
4    y = foo(x);
5    print(y);
6  }
7
8  int foo(int p) {
9    return p;
10 }
```

(b) Product for $\neg F \wedge G \wedge \neg H$

Figure 1: Example: secret is printed if `F` and `H` are disabled but `G` is enabled

Figure 1a this is the case only if features F and H are disabled but feature G is enabled.

Traditionally, one would perform information-flow analysis on a software product line by first generating all possible products from this product line, typically through the use of a pre-processor. In the next step, one would then apply a traditional information-flow analysis to the resulting (regular) program code. In Figure 1b we show the product for the violating feature configuration $\neg F \wedge G \wedge \neg H$. Any existing inter-procedural information-flow analysis would determine that a violating information flow exists in this product.

The problem is, however, that this traditional approach becomes prohibitively slow in the case where many features are present, and thus many different resulting products exists. Product lines with several hundreds or thousands of possible products are not uncommon [6]. Even the small example from Figure 1a induces $2^3 = 8$ possible products, only one of which actually violates our information-flow policy.

In this work we discuss a feature-sensitive information-flow analysis that instead allows users to analyze entire product lines, such as the one in Figure 1a, at once. When applied to this example, our analysis reports an information-flow violation at line 10, and that this violation can happen for the single configuration satisfying the feature constraint $\neg F \wedge G \wedge \neg H$. This not only obviates the analysis of individual products, it also yields the useful information which features are responsible for a violation. Depending on the concrete configuration, the violation may actually be benign, for instance if the resulting products are guaranteed to be run in a trusted environment. In this case the violation could be ignored.

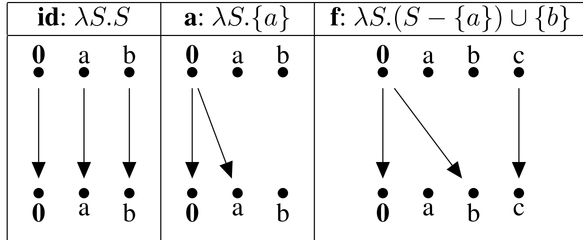


Figure 2: Function representation in IFDS, reproduced from [9]

3. Information-flow Analysis for Software Product Lines

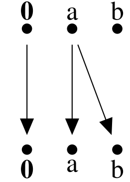
In the following section, we explain our approach to information-flow analysis of software product lines. We first explain the IFDS framework, and how it can be used to formulate traditional information-flow analyses. In Section 3.3, we then explain how such analyses can be lifted to Software Product Lines.

3.1 The IFDS framework

Our approach is limited to information-flow analyses formulated in the so-called *IFDS framework* by Reps, Horwitz and Sagiv [9]. This framework defines a general solution strategy for inter-procedural, flow-sensitive, context-sensitive analysis of finite distributive subset problems. The algorithm has complexity $O(ED^3)$, where E is the number of control-flow edges (or statements) of the analyzed program and D is the size of the analysis domain. For simple analyses that only consider two possible values, *high* for secret and *low* for public values, D will typically be quite low, as it is bounded by the maximal number of variables (or access paths) within any given method.

The major idea of the IFDS framework is to reduce any program-analysis problem formulated in this framework to a simple graph-reachability problem. The IFDS algorithm builds, based on the program's inter-procedural control-flow graph, a so-called "exploded super graph" in which a node (s, d) is reachable from a selected start node $(s_0, \mathbf{0})$ if and only if the data-flow fact d holds at s. (By "fact" we mean any logical statement, such as "variable v carries a high value.") To achieve this goal, the framework has to encode data-flow functions as nodes and edges. Figure 2, reproduced from [9], shows how to represent compositions of typical *gen* and *kill* functions, as they are used in information-flow analysis. The function **id** is the identity function, mapping each data-flow fact before a statement onto itself. In IFDS, the value **0** represents an empty fact that is always valid, i.e., two nodes representing **0** will always be connected. This **0** value is used to generate data-flow facts unconditionally. The flow function **a** generates the data-flow fact a, and at the same time kills the data-flow fact b. Function **f**, on the other hand kills a, generates b and leaves all other values (such as c) untouched. This could denote, for example, summary information for the following code snippet `a = 3; b = secret()`. Here, b would be generated, e.g., to denote that `b` is assigned a high value, and a would be killed to denote that `a` is assigned a low value.

In addition, information-flow analysis must often consider functions whose output actually depends on the input value, so-called *non-separable data-flow functions*. For instance, the function representation to the right could be chosen to model an assignment `b=a`. Here, `a` has the same value as before the assignment, modeled by the arrow from a to a, and `b` obtains `a`'s value, modeled by the arrow from a to b. If `b` was previously high, then it will only remain high if `a` was high as well. This is modeled by a missing arrow from b to b.



3.2 Information-flow analysis in IFDS

Figure 3 shows the exploded super graph for our information-flow analysis applied to the product from the running example (Figure 1b). We assume an information-flow policy stating that the return value of the method `secret()` is a source of high (secret) values and that the parameter to the method `print(..)` is a sink, i.e., must not be reached by high values. Our analysis generates high values at sources, as can be seen by the edge from **0** to `x` at the statement `x = secret()`. The analysis then tracks simple assignments. For the purpose of this position paper we ignore field assignments and aliasing. Those would be modeled by using a larger set of data-flow facts, incorporating not only local variable but alias sets and/or access paths. Our idea of lifting information-flow analysis to product lines is orthogonal to the question of how to model and resolve aliasing, which is why we omit those details here. We currently also ignore sanitizers and other forms of safe declassification of high values, but plan to add support by using appropriate kill functions.

In our example, the violation of the information-flow policy is detected by observing the data flow marked in red: the node for `y` just before the `print` statement is reachable in the graph, hence an information-flow violation is detected.

3.3 Information-flow analysis for Software Product Lines

In the following we explain how we can lift our IFDS-based information-flow analysis to software product lines. IFDS is actually a specific instance of a more generalized framework called *inter-procedural distributive environment transformers* (IDE) [10], also developed by Sagiv, Reps and Horwitz. As in IFDS, the IDE frameworks models data flow through edges in an exploded super graph. In addition to IFDS, however, IDE allows for the computation of distributive functions along those edges. We use this functionality to compute, along the edges representing the violating information

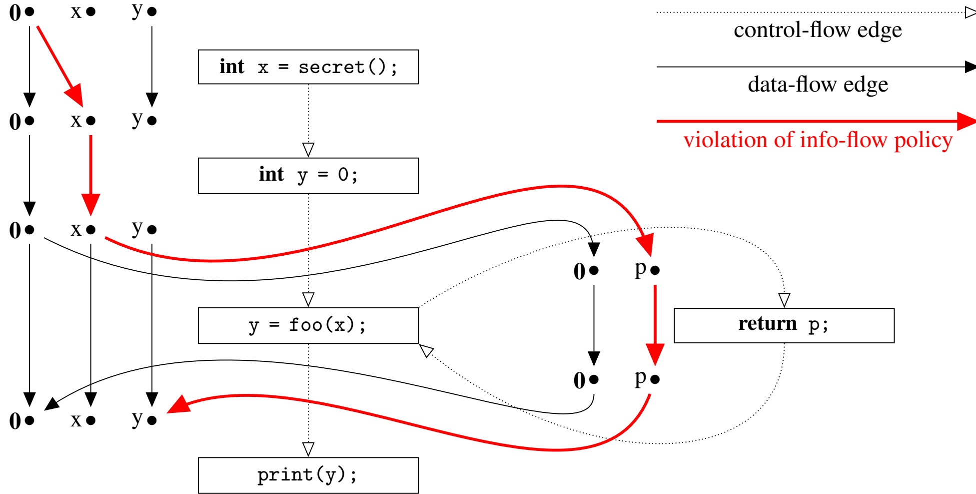
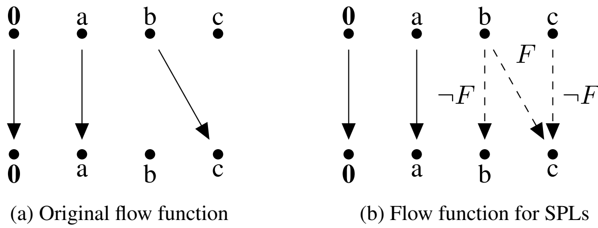


Figure 3: Exploded super graph for our running example; main method shown on left-hand side, method `foo` shown to the right



(a) Original flow function (b) Flow function for SPLs

Figure 4: Flow function before and after conversion

flow, a feature constraint that tells us which features must or must not be enabled for the potential flow to occur. Effectively, we convert IFDS function representations into IDE function representations, as shown in Figure 4.

In Figure 4a, we show an example IFDS flow function. In Figure 4b, we show the converted IDE version, applicable to software product lines. We assume that by parsing the product line's code we are able to see which conditional-compilation directives guard each statement. (In Section 4 we discuss how our implementation achieves this.) In Figure 4b, we consider a statement that is annotated with the feature F. Because the statement is guarded by F, if F is disabled it will *not* cause any data-flow changes, effectively degrading to the identity function. We therefore guard any data-flow edges that generate data-flow facts with F: the facts are only generated when the feature is active. In the figure, this is shown by the edge from b to c. We also generate edges for the inverse case: values that are killed by the original data-flow function (by missing an edge) are not killed if the feature is enabled. Hence, we introduce new "identity edges", labeled with $\neg F$, in the figure from b to b and from c to c.

Although in Figure 4b we labeled the edges only with feature constraints, in actuality the edges are associated with functions that compute a conjunction with this feature constraint. For instance, when moving from b to c in the figure, along the edge labeled with F, this will transform the original constraint, say c at the b-node, to the constraint $c \wedge F$. In other words, function composition is modeled by the "and" operator. This is because a violating information flow is only possible if all the feature constraints along this flow hold for the product in question.

At control-flow merge points we merge constraints using "or". In Figure 4b, when merging the constraints F and $\neg F$ for the two edges entering the c node, those constraints are merged to $F \vee \neg F$ = **true**. As the neutral element for this operation we naturally use **false**.

In Figure 5, we show how our inter-procedural information-flow analysis operates on our example product line. The initial constraint at the start node $(s_0, \mathbf{0})$ is **true**, the neutral element of our function composition operator "and". As the possible information flows are computed, constraints are conjoined, and disjoined at merge points. In the figure, the violating information flow leads to the following constraint:

$$(\textbf{true} \wedge \neg F \wedge G \wedge \neg H \wedge G) \vee \textbf{false} = \neg F \wedge G \wedge \neg H$$

In our actual implementation, we first stored feature constraints in a canonical minimized Disjunctive Normal Form, but found that this representation was quite inefficient for frequent operations such as conjunction. We now uses Binary Decision Diagrams which can be conjoined much more efficiently in the most common cases. We wish to emphasize that the idea of using IDE to extend IFDS-based analyses to product lines is a novel, original contribution of this paper.

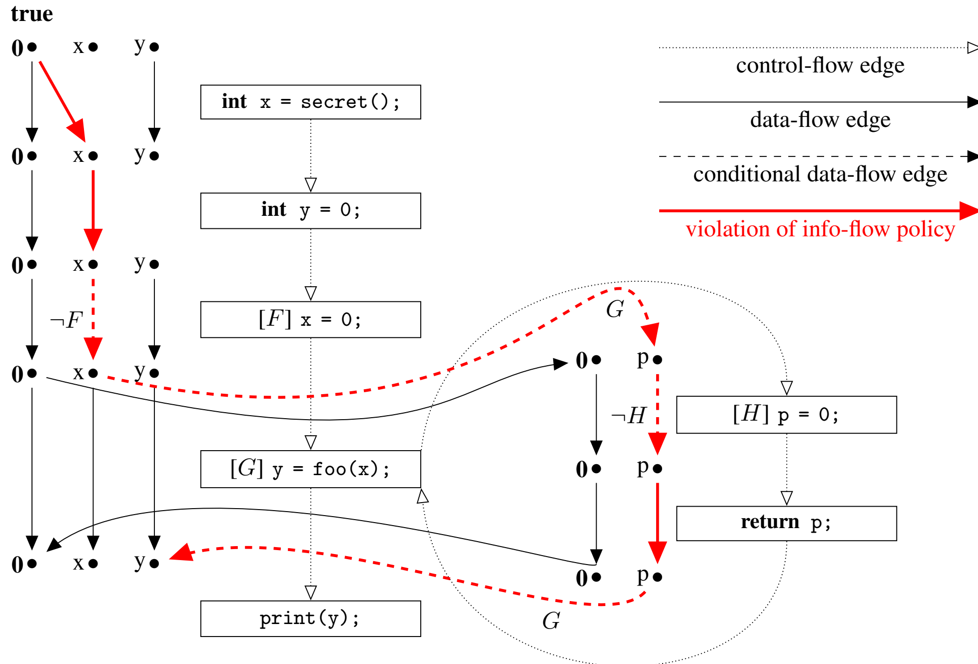


Figure 5: Example from Figure 3, lifted to SPLs; an edge labeled with feature constraint C represents the function $\lambda x.\, x \wedge C$

4. Implementation

We have almost finished an implementation based on the Soot program analysis and transformation framework [8] and CIDE, the Colored Integrated Development Environment [5]. We have implemented an IDE solver [10] in Soot that works directly on Soot's intermediate Java-code representation "Jimple". Jimple is a three-address code representation of Java programs that is particularly simple to analyze. Jimple statements are never nested, and all control-flow constructs are reduced to simple conditional and unconditional branches. Soot can produce Jimple code from Java source code or bytecode, and compile Jimple back into bytecode or into other intermediate representations.

To be able to actually parse software product lines, we decided to use CIDE, an extension of the Eclipse IDE [1]. In CIDE, software produce lines are expressed as plain Java programs. This makes them comparatively easy to parse: there are no explicit compiler directives such as `#ifdef` that would need to be handled. Instead, code varieties are expressed by marking code fragments with different colors. Each color is associated with a feature name. CIDE forbids so-called "undisciplined" annotations, i.e., forbids users from marking code regions that do not span entire nodes in the abstract syntax tree. As a result, this means that users can only color entire expressions, statements, methods, classes, etc. but cannot color arbitrary code fragments. Previous research has shown that this is typically not a practical limitation [6]. Figure 6 shows our running example program with the appropriately marked features in CIDE.

```
void main() {
    int x = secret();
    int y = 0;
    x = 0;
    y = foo(x);
    print(y);
}

int foo(int p) {
    p = 0;
    return p;
}
```

Figure 6: Example program in the Colored IDE (CIDE)

Current Limitations

Due to our tool chain, our implementation is currently limited to software product lines expressed with CIDE, which therefore have to use Java as the underlying base language. Having said that, there is nothing that precludes our approach from being applied to other programming languages. In particular, it could be applied to C, C++ and C# if similar tool support were present.

Another limitation of our implementation is that, due to a bug in CIDE, we are currently unable to make use of the

product line's feature model. A feature model defines, usually in a graphical way, a Boolean constraint that describes the set of all valid feature combinations. Feature models are typically defined manually by a user. Exploiting the feature model is useful both to improve analysis efficiency, and to avoid false positives. For example, assume a feature model inducing the constraint $F \leftrightarrow G$ for our example product line. This constraint would allow us to deduce that the configuration that we determined as violating, $\neg F \wedge G \wedge \neg H$ is not compatible with the feature-model constraint:

$$(\neg F \wedge G \wedge \neg H) \wedge (F \leftrightarrow G) = \textbf{false}$$

Hence, by making use of the model we could identify the detected violation as a false positive and refrain from reporting it to the user. But even better: by considering the feature model in every step of the computation of the feature constraint, we could identify and abort unnecessary computations. In Figure 5, when reaching the node labeled with p, we would recognize that the constraint computed so far, $\neg F \wedge G$, is already incompatible with the feature model. Thereby, one could abort computation of this information flow at this point, potentially saving a significant amount of analysis time. At the time of writing we are in contact with the developers of CIDE to find a fix or workaround for our problems with accessing the feature model.

Last but not least, our current presentation as well as prototype implementation ignores the problem of aliasing. Aliasing is important to resolve to be able to track information flow through heap-allocated variables such as fields or arrays. Guarnieri et al. have recently shown how to treat aliasing in an IFDS-based information-flow analysis for JavaScript [4]. Effectively, variable names are replaces by sets of access paths. We plan to integrate the handling of aliasing in the near future.

5. Related Work

Brabrand et al. present a mechanism to lift intra-procedural data-flow analyses to software product lines by extending the analysis abstraction with feature constraints [2]. Our approach, on the other hand, lifts information-flow analyses to SPLs on a whole-program level. In addition, our approach requires no extension of the analysis abstraction. The implementation of the IFDS flow functions can literally remain unchanged. Classen et al. describe how to apply model-checking techniques for temporal properties to Software Product Lines [3].

6. Conclusion

We have presented an inter-procedural, context-sensitive data-flow analysis for applications that are assembled using conditional-compilation constructs, particularly in the context of software product lines. Our current implementation, based on Soot and the Colored IDE (CIDE), is applicable to product lines based on Java. In future work, we plan to integrate the handling of aliasing. We are also interesting in conditional-compilation scenarios that go beyond product lines, such as code generated by web services.

Further, our current implementation does not capture implicit information flow through control-flow dependencies. Capturing such dependencies is generally possible by extending the program's super graph [7]. We plan to investigate whether such extensions can be lifted to product lines in a similar manner.

Acknowledgements. We thank Paulo Borba, Claus Brabrand, Márcio Ribeiro, Társis Tolêdo for providing their CIDE/Soot connector implementation and for giving valuable comments on this draft.

References

[1] The Eclipse IDE. http://eclipse.org/.

[2] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. In *AOSD 2012*, March 2012. To appear.

[3] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE 2010*, pages 335–344, 2010.

[4] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable javascript. In *ISSTA 2011*, pages 177–187, New York, NY, USA, 2011. ACM.

[5] C. Kästner. *Virtual Separation of Concerns*. PhD thesis, Universität Magdeburg, 2010.

[6] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *AOSD 2011*, pages 191–202, New York, NY, USA, 2011. ACM.

[7] Yin Liu and Ana Milanova. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 146–155, Washington, DC, USA, 2010. IEEE Computer Society.

[8] Vijay Sundaresan Patrick Lam Etienne Gagnon Raja Vallée-Rai, Laurie Hendren and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[9] 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.

[10] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT '95*, pages 131–170, Amsterdam, The Netherlands, The Netherlands, 1996. Elsevier Science Publishers B. V.