Precise Data Flow Analysis in the Presence of Correlated Method Calls*

Marianna Rapoport[1], Ondřej Lhoták[1], and Frank Tip[2]

[1] University of Waterloo
{mrapoport, olhotak}@uwaterloo.ca
[2] Samsung Research America
ftip@samsung.com

Abstract. When two methods are invoked on the same object, the dispatch behaviours of these method calls will be correlated. If two correlated method calls are polymorphic (i.e., they dispatch to different method definitions depending on the type of the receiver object), a program's interprocedural control flow graph will contain infeasible paths. Existing algorithms for data-flow analysis are unable to ignore such infeasible paths, giving rise to loss of precision.
We show how infeasible paths due to correlated calls can be eliminated for *Interprocedural Finite Distributive Subset* (IFDS) problems, a large class of data-flow analysis problems with broad applications. Our approach is to transform an IFDS problem into an *Interprocedural Distributive Environment* (IDE) problem, in which edge functions filter out data flow along infeasible paths. A solution to this IDE problem can be mapped back to the solution space of the original IFDS problem. We formalize the approach, prove it correct, and report on an implementation in the WALA analysis framework.

1 Introduction

A control-flow graph (CFG) is an over-approximation of the possible flows of control in concrete executions of a program. It may contain *infeasible* paths that cannot occur at runtime. The precision of a data-flow analysis algorithm depends on its ability to detect and disregard such infeasible paths. The *Interprocedural Finite Distributive Subset* (IFDS) algorithm [16] is a general data-flow analysis algorithm that avoids infeasible interprocedural paths in which calls and returns to/from functions are not properly matched. The *Interprocedural Distributive Environment* (IDE) algorithm [18] has the same property, but supports a broader range of data-flow problems.

This paper presents an approach to data-flow analysis that avoids a type of infeasible path that arises in object-oriented programs when two or more methods are dynamically dispatched on the same receiver object. If the method

* This research was supported by the Natural Sciences and Engineering Research Council of Canada and the Ontario Ministry of Research and Innovation.

calls are polymorphic (i.e., the method invoked depends on the run-time type of the receiver), then their dispatch behaviours are correlated, and some of the paths between them are infeasible. A recent paper [21] made this observation but did not present any concrete algorithm to take advantage of it.

Our approach transforms an IFDS problem into an IDE problem that precisely accounts for infeasible paths due to correlated calls. The results of this IDE problem can be mapped back to the data-flow domain of the original IFDS problem, but are more precise than the results of directly applying the IFDS algorithm to the original problem. We present a formalization of the transformation and prove its correctness: specifically, we prove it still soundly considers all paths that are feasible, and that it avoids flow along all paths that are infeasible due to correlated calls.

We implemented the correlated-calls transformation and the IDE algorithm in Scala, on top of the WALA framework for static analysis of JVM bytecode [5]. Our prototype implementation was tested extensively by using it to transform an IFDS-based taint analysis into a more precise IDE-based taint analysis, and applying the latter to small example programs with correlated calls. Our prototype along with all tests will be made available to the artifact evaluation committee.

The remainder of this paper is organized as follows. Section 2 presents a motivating example. Section 3 reviews the IFDS and IDE algorithms. Section 4 presents the correlated-calls transformation, states the correctness properties[3], and discusses our implementation. Related work is discussed in Section 5. Finally, Section 6 presents conclusions and directions for future work.

2 Motivation

We illustrate our approach using a small example that applies our technique to improve the precision of taint analysis. A taint analysis computes how string values may flow from "sources", which are typically statements that read untrusted input, to "sinks", which are typically security-sensitive operations such as calls to a database. In previous research [2, 6], taint analysis algorithms have been formulated as IFDS problems.

Figure 1 shows a small Java program. The program declares a class A with a subclass B, where A defines methods foo() and bar() that are overridden in B. We assume that secret values are created by an unspecified function secret(), which is called in A.foo() on line 2. Any write to standard output is assumed to be a sink (e.g., the call to System.out.println() in B.bar()). Depending on the number of arguments passed to the program, the main() method of the example program creates either an A-object or a B-object. The program then calls foo() on this object on line 18, which is followed by a call to bar() on the same object.

We wish to answer the following question: Is it possible for the untrusted value that is read on line 2 to flow to the print statement? Consider the control-flow supergraph for the example program that is shown in Figure 2. The nodes

[3] Detailed proofs of our lemmas and theorems can be found in the Technical Report [15].

```
 1 class A {
 2   String foo { return secret(); }
 3   void bar(String s) {}
 4 }
 5 class B extends A {
 6   String foo {
 7     return "not␣secret";
 8   }
 9   void bar(String s) {
10     System.out.println(s);
11   }
12 }
13
14 class Main {
15   static void main(String[] args) {
16     A a = (args == null)
17         ? new A() : new B();
18     String v = a.foo();
19     a.bar(v);
20   }
21 }
```

Fig. 1: Example program containing correlated calls

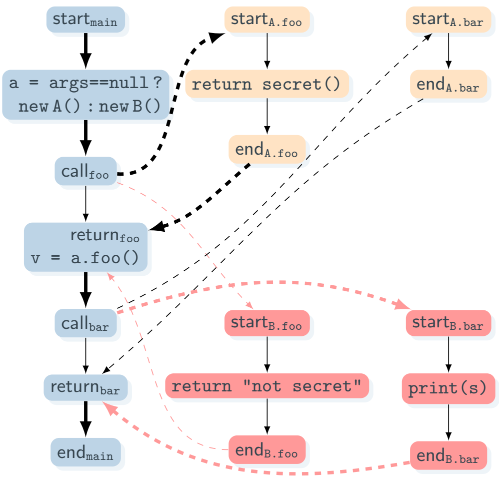

Fig. 2: Control flow supergraph for the example program of Figure 1. Dashed lines depict interprocedural edges. An infeasible path is shown in bold.

in this graph correspond to statements, method entry points (start nodes) and method exit points (end nodes). For each method call, the graph contains a distinct call-node and a return-node. Edges in the graph reflect intraprocedural control flow, flow of control from a caller to a callee (edges from call-nodes to start-nodes), or flow of control from a callee back to a caller (edges from end-nodes to return-nodes).

In our example, the control flow within each method is straightforward and all interesting issues arise from interprocedural control flow. In particular, since a may point to either an A-object or a B-object, the call on line 18 may dispatch to either A.foo() or to B.foo(), as is reflected by edges from the node labeled call$_{\text{foo}}$ to the nodes labeled start$_{\text{A.foo()}}$ and start$_{\text{B.foo()}}$ and by edges from the nodes labeled end$_{\text{A.foo}}$ and end$_{\text{B.foo}}$ to the node labeled return$_{\text{foo}}$. Similarly, there are edges from the node labeled call$_{\text{bar}}$ to the nodes start$_{\text{A.bar()}}$ and start$_{\text{B.bar()}}$, and edges from the nodes labeled end$_{\text{A.bar}}$ and end$_{\text{B.bar}}$ to the node labeled return$_{\text{bar}}$.

An IFDS analysis propagates data-flow facts along the edges of a control flow supergraph such as the one in Figure 2. The IFDS algorithm already avoids flow along infeasible paths from one call site, through a target method, and returning to a different call site of the target method. However, in this example, all methods are called in exactly one place, so IFDS is unable to eliminate data flow along any of the paths shown in the figure. As a result, IFDS-based taint analysis algorithms such as [2, 6] would report that the secret value read on line 2 might flow to the print statement on line 10.

As we discussed previously, the calls to foo() and bar() may dispatch to the implementations in classes A and B, because the receiver variable a may be bound to objects of type A or B at run time. However, the methods foo() and bar() are invoked on *the same object*. Thus the behaviours of the method calls are *correlated*: if the call to foo() dispatches to A.foo(), then the call to bar() must dispatch to A.bar(), and analogously for B.foo() and B.bar(). Consequently, paths such as the one shown in bold in Figure 2 where the calls dispatch to A.foo() and B.bar() are infeasible.

Our main contribution is an algorithm for transforming an IFDS problem into an IDE problem that expresses the feasibility of paths in light of correlated calls. The approach associates with each interprocedural CFG edge a function that records the types of variables that are used as the receiver of correlated method calls. Paths that are composed of edges in which the same receiver expression has different types are infeasible, and the propagation of data-flow facts along such paths is prevented. Applying our technique to an IFDS-based taint analysis would enable the resulting IDE-based taint analysis to determine that no secret value can flow from line 2 to the print statement on line 10.

While the discussion in this section has focused on the specific problem of taint analysis, our technique generally applies to *any* data-flow-analysis problem that can be expressed in the IFDS framework. This includes many common analysis tasks such as reaching definitions, constant propagation, slicing, typestate analysis, pointer analysis, and lightweight shape analysis.

2.1 Occurrences of Correlated Calls

How often do correlated calls occur in practice? To assess the benefit of the correlated-calls analysis, we counted the number of correlated calls that occur in programs of the Dacapo benchmarks [3], using the WALA framework [5]. Our goal was to obtain an upper bound on the number of redundant IFDS-result nodes that could be potentially removed by our analysis. The results are shown in the Technical Report [15].

In these programs, on average, 3% of all call sites C are polymorphic call sites C_P. Out of these polymorphic call sites, a significant fraction (39%) are correlated call sites $C^{\Subset}$. We also see that, on average, each correlated-call receiver is involved in approximately three correlated calls.

2.2 An Example from the Scala Collections Library

The Scala collections library contains the trait TraversableOnce that is shared by both collections and iterators over them. The toArray method of this trait creates an array and copies the contents of the collection or iterator into it:

```
val result = new Array[B](this.size)
this.copyToArray(result, 0)
```

When this refers to an iterator rather than a collection, the call to this.size extracts all elements of the iterator to count them. At the call to copyToArray, the iterator is already empty, so nothing is copied to the newly created array. One could design an IFDS analysis to detect this kind of bug.

However, the implementation of TraversableOnce.toArray is actually correct because the above code is guarded with a test: if (this.isTraversableAgain) ... When the isTraversableAgain method returns false, as it does for an iterator, the toArray method uses a different (less efficient) implementation. The bug report would therefore be a false positive. The isTraversableAgain method is easy to analyze: it returns the constant true in a collection and the constant false in an iterator. However, in order to eliminate the false positive bug report, an analysis would need to rule out infeasible paths using correlated calls. Specifically, the following path triggers the bug, but is infeasible: first, call isTraversableAgain on a collection, returning true, then call size and copyToArray on an iterator. Our correlated calls analysis could determine that this path is infeasible because it calls the collection version of isTraversableAgain but the iterator versions of size and copyToArray. The relevant code from TraversableOnce and other related traits is shown in the Technical Report [15].

3 Background

This section defines terminology and presents the IFDS and IDE algorithms.

3.1 Terminology and Notation

The *control-flow graph* of a procedure is a directed graph whose nodes are instructions, which contains an edge from n_1 to n_2 whenever n_2 may execute immediately after n_1. A CFG has a distinguished *start node* start_p and *end node* end_p. Following the presentation of Reps et al. [16, 18], we follow every call instruction with a no-op instruction, so that every *call node* is immediately followed by a *return node* in the CFG. The *control-flow supergraph* of a program contains the CFGs of all of the procedures as subgraphs. In addition, for each call instruction c, the supergraph contains a *call-to-start* edge to the start node of every procedure that may be called from c, and an *end-to-return* edge from the end node of the procedure back to the call instruction.

A call site is *monomorphic* if it always calls the same procedure. In an object-oriented language, a call site $r.m(\ldots)$ can dynamically dispatch to multiple methods depending on the runtime type of the object pointed to by the receiver r. A call site that calls multiple procedures is called *polymorphic*. We define a function lookup to specify the dynamic dispatch: if s is the signature of m and t is the runtime type of the object pointed to by r, $\mathsf{lookup}(s, t)$ gives the procedure that will be invoked by the call $r.m(\ldots)$. We also define a function τ that may be viewed as the inverse of lookup: given a signature s and a specific invoked procedure f, $\tau(s, f)$ gives the set of all runtime types of r that cause $r.m(\ldots)$ to dispatch to f: $\tau(s, f) = \{t \mid \mathsf{lookup}(s, t) = f\}$.

A path in the control-flow supergraph is *valid* if it follows the usual stack-based calling discipline: every end-to-return edge on the path returns to the site of the most recent call that has not yet been matched by a return. The set of all valid paths from the program entry point to a node n is denoted $\mathsf{VP}(n)$.

A *lattice*[4] is a partially ordered set $(S, \sqsubseteq)$ in which every subset has a least upper bound, called *join* or $\sqcup$, and a greatest lower bound, called *meet* or $\sqcap$. A *meet semilattice* is a partially ordered set in which every subset only has a greatest lower bound. The symbols $\bot$ and $\top$ are used to denote the greatest lower bound of S and of the empty set, respectively.

We denote a map m as a set of pairs of keys and values, with each key appearing at most once. For a map m, $m(k)$ is the value paired with the key k. We denote by $m[x \to y]$ a map that maps x to y and every other key k to $m(k)$.

3.2 IFDS

The IFDS framework [16] is a precise and efficient algorithm for data-flow analysis that has been used to solve a variety of data-flow analysis problems [4, 9, 12, 22]. The IFDS framework is an instance of the *functional approach* to data-flow analysis [19] because it constructs summaries of the effects of called procedures. The IFDS framework is applicable to *interprocedural* data-flow problems whose domain consists of *subsets* of a *finite* set D, and whose data-flow functions are *distributive*. A function f is distributive if $f(x_1 \sqcap x_2) = f(x_1) \sqcap f(x_2)$.

The IFDS algorithm is notable because it computes a meet-over-valid paths solution in polynomial time. Most other interprocedural analysis algorithms are either: (i) imprecise due to invalid paths, (ii) general but do not run in polynomial time [7, 19], or (iii) handle a very specific set of problems [8].

The input to the IFDS algorithm is specified as $(G^*, D, F, M_F, \sqcap)$, where $G^* = (N^*, E^*)$ is the supergraph of the input program with nodes N^* and edges E^*, D is a finite set of *data-flow facts*, F is a set of distributive data-flow functions of type $2^D \to 2^D$, $M_F : E^* \to F$ assigns a data-flow function to each supergraph edge, and $\sqcap$ is the *meet operator* on the powerset 2^D, either union or intersection. In our presentation, the meet operator will always be union, but all of the results apply dually when the meet is intersection.

The output of the IFDS algorithm is, for each node n in the supergraph, the *meet-over-all-valid-paths* solution $\mathsf{MVP}_F(n) = \bigsqcap_{q \in \mathsf{VP}(n)} M_F(q)(\top)$, where M_F is extended from edges to paths by composition.

Overview of the IFDS Algorithm The key idea behind the IFDS algorithm is that it is possible to represent any distributive function f from 2^D to 2^D by a *representation relation* $R_f \subseteq (D \cup \{0\}) \times (D \cup \{0\})$. The representation relation can be visualized as a bipartite graph with edges from one instance of $D \cup \{0\}$ to another instance of $D \cup \{0\}$. The IFDS algorithm uses such graphs to efficiently

[4] The definitions that we give here are of *complete* lattices and semilattices. Since all of the (semi)lattices discussed in this paper are required to be complete, we omit the *complete* qualifier.

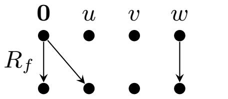

Fig. 3: $R_f = \{(\mathbf{0},\, \mathbf{0}),$ $(\mathbf{0},\, u),\, (w,\, w)\}$

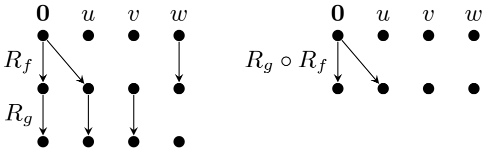

Fig. 4: $R_g \circ R_f$

represent both the input data-flow functions and the summary functions that it computes for called procedures. Specifically, the representation relation R_f of a function f is defined as:

$$R_f = \{(\mathbf{0},\, \mathbf{0})\} \cup \{(\mathbf{0},\, d_j) \,|\, d_j \in f(\varnothing)\} \cup \{(d_i,\, d_j) \,|\, d_j \in f(\{d_i\}) \setminus f(\varnothing)\}.$$

Example 1. Given $D = \{u,\, v,\, w\}$ and $f(S) = S \setminus \{v\} \cup \{u\}$, the representation relation $R_f = \{(\mathbf{0},\, \mathbf{0}),\, (\mathbf{0},\, u),\, (w,\, w)\}$, which is depicted in Figure 3.

The representation relation decomposes a flow function into functions (edges) that operate on each fact individually. This is possible due to distributivity: applying the flow function to a set of facts is equivalent to applying it on each fact individually and then taking the union of the results.

The meet of two functions can be computed as simply the union of their representation functions: $R_{f \sqcap f'} = R_f \cup R_{f'}$. The composition of two functions can be computed by combining their representation graphs, merging the range nodes of the first function with the corresponding domain nodes of the second function, and finding paths in the resulting graph.

Example 2. If $g(S) = S \setminus \{w\}$ and $f(S) = S \setminus \{v\} \cup \{u\}$, then $R_g \circ R_f = \{(\mathbf{0},\, \mathbf{0}),\, (\mathbf{0},\, u)\}$, as illustrated in Figure 4.

Composition of two distributive functions f and f' corresponds to finding reachable nodes in a graph composed from their representation relations R_f and $R_{f'}$. Therefore, evaluating the composed data-flow function for a control flow path corresponds to finding reachable nodes in a graph composed from the representation relations of the data-flow functions for individual instructions.

It is this graph of representation relations that the IFDS algorithm operates on. In this graph, called the *exploded supergraph*, each node is a pair (n, d), where $n \in N^*$ is a node of the control-flow supergraph and d is an element of $D \cup \{0\}$. For each edge $(n \rightarrow n') \in E^*$, the exploded supergraph contains a set of edges $(n, d_i) \rightarrow (n', d_j)$, which form the representation relation of the data-flow function $M_F(n \rightarrow n')$. The IFDS algorithm finds all exploded supergraph edges that are reachable by *realizable* paths in the exploded supergraph. A path is *realizable* if its projection to the (non-exploded) supergraph is a valid path (i.e., if it is of the form $(n_0, d_0) \rightarrow (n_1, d_1) \rightarrow \cdots \rightarrow (n_m, d_m)$ and where $n_0 \rightarrow n_1 \rightarrow \cdots \rightarrow n_m$ is a valid path).

Example 3. The exploded supergraph for Listing 1 is shown in Figure 5. The labels on the edges will be explained in Section 3.3 We can see that there is a

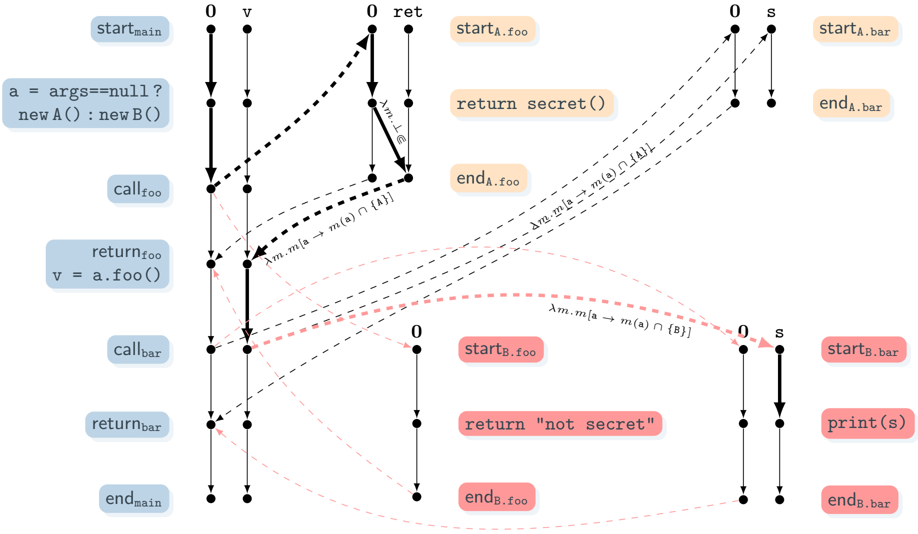

Fig. 5: An example program demonstrating correlated-call edge functions on the **0**-node path for Listing 1. All non-labeled edges are implicitly labeled with identity functions id. The variable `ret` denotes the return value of the `A.foo` method.

realizable path, highlighted in bold, from the start node of the exploded graph to the variable `s` at the node print(s) in the `B.bar` method. This means that `s` is considered secret at that node.

3.3 IDE

The IDE algorithm [18] extends IFDS to *interprocedural distributive environment* problems. An IDE problem is one whose data-flow lattice is the lattice $\mathsf{Env}(D, L)$ of maps from a finite set D to a meet semilattice L of finite height, ordered pointwise. Like IFDS, IDE requires the data-flow functions to be distributive.

The input to the IDE algorithm is $(G^*, D, L, M_{\mathsf{Env}})$ where G^* is a control-flow supergraph, D is a set of data-flow facts, L is a meet semilattice of finite height, and $M_{\mathsf{Env}} : E^* \to (\mathsf{Env}(D, L) \to \mathsf{Env}(D, L))$ assigns a data-flow function to each supergraph edge.

The output of the IDE algorithm is, for each node n in the supergraph, the *meet-over-all-valid-paths* solution $\mathsf{MVP}_{\mathsf{Env}}(n) = \bigsqcap_{q \in \mathsf{VP}(n)} M_{\mathsf{Env}}(q)(\top_{\mathsf{Env}})$, where $\top_{\mathsf{Env}} = \lambda d.\top$ is the top element of the lattice of environments, and M_{Env} is extended from edges to paths by composition.

Overview of the IDE Algorithm Just as any distributive function from 2^D to 2^D can be represented with a representation relation, it is also possible to represent any distributive function from $\mathsf{Env}(D, L)$ to $\mathsf{Env}(D, L)$ with a *pointwise representation.* A pointwise representation is a bipartite graph with the same nodes [5] and edges as a representation relation, except that each edge is labelled with a *micro-function*, which is a function from L to L.

Thanks to distributivity, every environment transformer $t : \mathsf{Env}(D, L) \rightarrow \mathsf{Env}(D, L)$ can be decomposed into its effect on $\top_{\mathsf{Env}}$ and on a set of environments $\top_{\mathsf{Env}}[d_i \rightarrow l]$ that map every element to $\top$ except one (d_i). Formally,

$$t(m)(d_j) = \lambda l.t(\top_{\mathsf{Env}})(d_j) \sqcap \bigsqcap_{d_i \in D} \lambda l.t(\top_{\mathsf{Env}}[d_i \rightarrow l])(d_j).$$

The functions $\lambda l. \cdots$ in this decomposition are the micro-functions that appear on the edges of the pointwise representation edges from $\mathbf{0}$ to each d_j and from each d_i to each d_j.[6] The absence of an edge in the pointwise representation from some d_i to some d_j is equivalent to an edge with micro-function $\lambda l.\top$.

Example 4. In the exploded supergraph in Figure 5, the micro-functions are shown as labels on the graph edges. Every edge without an explicit label has the identity as its micro-function. The micro-functions on the three edges from the node `return secret()` to the node $\mathtt{end}_{\mathtt{A.foo}}$ together represent the environment transformer $\lambda e.e[\mathtt{ret} \rightarrow \lambda m.\bot \sqcap \lambda m.m]$.

To eliminate infeasible paths due to correlated calls, we encode the taint analysis using environments $e \in \mathsf{Env}(D, L)$, where D is the set of variables and L is a map from receiver variables to sets of possible types. The interpretation of such an environment e is that a given variable $v \in D$ may contain a secret value in an execution in which the runtime types of the objects pointed to by the receiver variables are in the sets specified by $e(v)$.

The meet of two environment transformers t_1, t_2 is computed as the union of the edges in their pointwise representations. When the same edge appears in the pointwise representations of both t_1 and t_2, the micro-function for that edge in $t_1 \sqcap t_2$ is the meet of the micro-functions for that same edge in t_1 and in t_2.

The composition of two environment transformers can be computed by combining their pointwise representation graphs in the same fashion as IFDS representation relations, and computing the composition of the micro-functions appearing along each path in the resulting graph.

The IDE algorithm operates on the same exploded supergraph as the IFDS algorithm (but its edges are labelled with micro-functions). For each pair (n, d) of node and fact, IDE computes a micro-function equal to the meet of the micro-functions of all the realizable paths from the program entry point to the pair.

In order to do this efficiently, the IDE algorithm requires a representation of micro-functions that is general enough to express the basic micro-functions of the data-flow functions for individual instructions, and that supports computing the meet and composition of micro-functions.

A practical implementation of the IDE algorithm requires the input data-flow functions to be provided in their pointwise representation as exploded supergraph edges labelled with micro-functions. Specifically, the input is generally

[5] The IDE literature uses the symbol Λ for the node that is denoted $\mathbf{0}$ in the IFDS literature. We use $\mathbf{0}$ throughout this paper for consistency.

[6] The IDE paper defines a more complicated but equivalent set of micro-functions that eliminate some duplication of computation.

provided as a function $\mathsf{EdgeFn} : (N^* \times D) \times (N^* \times D) \to F$, where F is the set of representations of micro-functions from L to L. Given an exploded supergraph edge $e = (n,\, d) \to (n',\, d')$, $\mathsf{EdgeFn}((n,\, d),\, (n',\, d'))$ returns the micro-function that appears on the exploded supergraph edge e. In an implementation, it can be convenient to split the function EdgeFn into separate functions that handle the cases when $n \to n'$ is an intraprocedural edge, a call-to-return edge, a call-to-start edge, or an end-to-return edge.

4 Correlated Calls Analysis

4.1 Transformations from IFDS to IDE

Let $G^{\#}$ be the exploded supergraph of an arbitrary IFDS problem. A *transformation* $\mathcal{T} :\ (G^{\#}) \to (G^{\#},\, L,\, \mathsf{EdgeFn})$ converts the IFDS problem into an IDE problem. We consider two IFDS-to-IDE transformations: an *equivalence transformation* $\mathcal{T}^{\equiv}$ (pronounced "t-equiv") and a *correlated-calls transformation* $\mathcal{T}_S^{\Subset}$ (pronounced "t-c-c") for a set of receivers S. Both transformations keep the exploded supergraph $G^{\#}$ the same, and only generate different edge functions. The solution of the IDE problem can be mapped back to an IFDS solution. If the equivalence transformation was used, then this solution is identical to the solution that would be computed by the IFDS algorithm for the original IFDS problem. If the correlated-calls transformation was used, then this solution is more precise because it excludes flow along infeasible paths due to correlated calls.

Equivalence Transformation The lattice for the equivalence transformation $\mathcal{T}^{\equiv}$ is the two-point lattice $L^{\equiv} = \{\bot,\ \top\}$, where $\bot$ means "reachable", and $\top$ means "not reachable". The edge functions $\mathsf{EdgeFn}^{\equiv}$ are defined as

$$\mathsf{EdgeFn}^{\equiv} = \begin{cases} \lambda e \,.\, \lambda m \,.\, \bot & \text{if } e = (n_1, \mathbf{0}) \to (n_2, d_2)\text{, where } d_2 \neq \mathbf{0}; \\ \lambda e \,.\, \mathsf{id} & \text{otherwise.} \end{cases} \tag{1}$$

At a "diagonal" edge from a $\mathbf{0}$-fact to a non-$\mathbf{0}$-fact d, the micro function returns $\bot$ to make the fact d reachable. All other micro-functions are the identity function.

Correlated-Calls Transformation In the correlated-calls transformation $\mathcal{T}_R^{\Subset}$, the lattice elements are maps from receivers to sets of types: $L^{\Subset} = \left\{\, m :\ R \to 2^T \right\}$, where R is the set of considered receivers and T is the set of all types. For each receiver r, the map gives an overapproximation of the possible runtime types of r. Sets of types are ordered by the superset relation, and this is lifted to maps from receivers to sets of types, so the bottom element $\bot_{\Subset}$ maps every receiver to the set of all types, and the top element $\top_{\Subset}$ maps every receiver to the empty set of types. During an actual execution, every receiver r points to an object of some runtime type. Therefore, a data-flow fact is unreachable along a given path if its corresponding lattice element maps any receiver to the empty set of types.

A micro-function $f \in L^{\Subset} \to L^{\Subset}$ defines how the map from receivers to types should be updated when an instruction is executed. The micro-function for most kinds of instructions is the identity. On a call to and return from a specific method m called on receiver r, the micro-function restricts the receiver-to-type map to map r only to types consistent with the polymorphic dispatch to method m. Finally, when an instruction assigns an object of unknown type to a receiver r, the corresponding micro-function updates the map to map r to the set of all types. This is made precise by the following definition:

Definition 1. *Given a previously fixed set $S \subseteq R$ of receivers, the micro-function $\varepsilon_S(e)$ of a supergraph edge e is defined as:*

$$\varepsilon_S(e) = \lambda m \,. \tag{2}$$

$$\begin{cases} m[r \to m(r) \cap \tau(s,\, f)], & \text{if } e \text{ is a call-start edge } r.c() \to \mathsf{start}_f \text{ that calls procedure } f \text{ with signature } s, \text{ and } r \in S; \\ m[r \to m(r) \cap \tau(s,\, f)] \\ [v_1 \to \bot_T] \dots [v_k \to \bot_T], & \text{if } e \text{ is an end-return edge } \mathsf{end}_f \to \mathsf{return}_{r.c()} \text{ from method } f \text{ with signature } s \text{ to the return node corresponding to the call } r.c(),\ v_1, \dots, v_k \in S \text{ are the local variables in } f, \text{ and } r \in S; \\ m\,[r \to \bot_T]\,, & \text{if } e = n_1 \to n_2 \text{ and } n_1 \text{ contains an assignment to } r \in S; \\ m & \text{otherwise.} \end{cases}$$

In the above definition, the purpose of the set S is to limit the set of considered receivers. We will use S in Section 4.5.

We can now define EdgeFn, which assigns a micro-function to each edge in the exploded supergraph. Along a $\mathbf{0}$-edge, the micro function is the identity. On a "diagonal" edge from $\mathbf{0}$ to a non-$\mathbf{0}$ fact that corresponds to some data-flow fact becoming reachable, $\varepsilon_S(e)$ is applied to $\bot_{\Subset}$ that maps every receiver to an object of every possible type. On all other edges, $\varepsilon_S(e)$ is applied to the existing map before the edge. The is formalized in the following definition.

Definition 2. *For each edge $e = (n_1, d_1) \to (n_2, d_2)$, $\mathsf{EdgeFn}^{\Subset}_S(e)$ is defined as follows:*

$$\mathsf{EdgeFn}^{\Subset}_S(e) = \begin{cases} \mathsf{id} & \text{if } d_1 = d_2 = \mathbf{0}, \\ \lambda m \,.\, \varepsilon_S(e)(\bot_{\Subset}) & \text{if } d_1 = \mathbf{0} \text{ and } d_2 \neq \mathbf{0}, \\ \lambda m \,.\, \varepsilon_S(e)(m) & \text{otherwise.} \end{cases} \tag{3}$$

Example 5. Consider the program from Figure 1, whose exploded supergraph appeared in Figure 5. Returning a secret value in method A.foo creates a "diagonal" edge from the $\mathbf{0}$-fact to the method's return value r. The diagonal edge is labeled with $\lambda m \,.\, \bot_{\Subset}$, so every receiver is mapped to the set of all types $\bot_T$. On the end-return edge from A.foo to main, the set of types of a is restricted by the micro function $\lambda m \,.\, m[\mathsf{a} \to m(\mathsf{a}) \cap \{\mathsf{A}\}]$ corresponding to the assignment of the return value r to v. On the call-start edge from main to B.bar, the possible types of a are further restricted by the micro-function $\lambda m \,.\, m[\mathsf{a} \to m(\mathsf{a}) \cap \{\mathsf{B}\}]$ on the edge that passes the argument v to the parameter s. The composition of these micro functions results in the empty set as the possible types of a, indicating that this path is infeasible.

4.2 Converting IDE Results to IFDS Results

An IFDS solution $\mathcal{R}_{\text{IFDS}}$ has type $N^* \to 2^D$: it maps each program point n to a set of facts d that may be reached at n. An IDE solution $\mathcal{R}_{\text{IDE}}$ pairs each such fact d with a lattice element ℓ, so its type is $N^* \to (D \to L)$.

In the equivalence transformation lattice $L^{\equiv}$, $\bot$ means reachable and $\top$ means unreachable. Therefore, an IDE solution ρ computed using $\mathcal{T}^{\equiv}$ is converted to an IFDS solution as: $\mathcal{U}^{\equiv}(\rho) = \lambda n.\{d \mid \rho(n)(d) \neq \top\}$. In the correlated-calls transformation lattice $L^{\Subset}$, a map that maps any receiver to the empty set of possible types means that the corresponding data-flow path is infeasible. Therefore, an IDE solution ρ computed using $\mathcal{T}^{\Subset}_S$ is converted to an IFDS solution as

$$\mathcal{U}^{\Subset}(\rho) = \lambda n.\{d \mid \forall r \in S\,.\, \rho(n)(d)(r) \neq \top_T\}. \tag{4}$$

4.3 Implementation of Correlated Calls Micro-Functions

Conceptually, micro-functions are functions from L to L, where L is the IDE lattice, either $L^{\equiv}$ or $L^{\Subset}$ in our context. The IDE algorithm requires an efficient representation of micro-functions. The representation must support the basic micro-functions that we presented in Section 4.1, and it must support function application, comparison, and be closed under function composition and meet. We now propose such a representation for the correlated-calls micro-functions.

The representation of a micro-function is a map from receivers to pairs of sets of types $I(r)$ and $U(r)$, where $U(r)$ is required to be a subset of $I(r)$. We use the notation $\langle I, U\rangle$ to represent such a map, and $I(r)$ and $U(r)$ to look up the sets corresponding to a particular receiver r. The micro-function takes the existing set of possible types of the receiver r, intersects it with $I(r)$, then unions it with $U(r)$: $[\![\langle I, U\rangle]\!] = \lambda m\,.\,\lambda r\,.\,(m(r) \cap I(r)) \cup U(r)$.

All of the basic micro-functions defined in Definition 1 can be expressed in this representation. The following lemmas show how function comparison, composition, and meet can be implemented using basic set operations on I and U. The proofs of all of the lemmas and theorems are in the Technical Report [15].

Lemma 1. *For any pair of micro-function representations $\langle I, U\rangle$, $\langle I', U'\rangle$,*

$$\forall r\,.\, I(r) = I'(r) \wedge U(r) = U'(r) \iff [\![\langle I, U\rangle]\!] = [\![\langle I', U'\rangle]\!]\,. \tag{5}$$

Lemma 2. *For any pair of micro-function representations $\langle I, U\rangle$, $\langle I', U'\rangle$,*

$$[\![\langle I, U\rangle \circ \langle I', U'\rangle]\!] = [\![\langle I, U\rangle]\!] \circ [\![\langle I', U'\rangle]\!]\,,$$

where the composition of two micro-function representations is defined as follows:

$$\langle I, U\rangle \circ \langle I', U'\rangle = \langle \lambda r\,.\,(I(r) \cap I'(r)) \cup U(r),\ \lambda r\,.\,(I(r) \cap U'(r)) \cup U(r)\rangle\,.$$

Lemma 3. *Let $[\![\langle I, U\rangle]\!] \sqcap [\![\langle I', U'\rangle]\!] = \lambda m.\lambda r.\, [\![\langle I, U\rangle]\!]\,(m)(r) \cup [\![\langle I', U'\rangle]\!]\,(m)(r)$. For any pair of micro-function representations $\langle I, U\rangle$, $\langle I', U'\rangle$,*

$$[\![\langle I, U\rangle \sqcap \langle I', U'\rangle]\!] = [\![\langle I, U\rangle]\!] \sqcap [\![\langle I', U'\rangle]\!]\,, \tag{6}$$

where the meet of two micro-function representations is defined as follows:

$$\langle I, U\rangle \sqcap \langle I', U'\rangle = \langle \lambda r\,.\,I(r) \cup I'(r),\ \lambda r\,.\,U(r) \cup U'(r)\rangle\,.$$

4.4 Theoretical Results

The following lemma shows that our analysis is sound, i.e. that the resulting IDE problem still considers all data-flow paths that are actually feasible.

Lemma 4 (Soundness). *Let P be an IFDS problem and $p = [\mathsf{start}_{\mathit{main}}, \ldots, n]$ a concrete execution path, and let $d \in D$. If $d \in M_F(p)(\varnothing)$, then*

$$d \in \mathcal{U}^{\Subset}\left(\mathcal{R}_{IDE}(\mathcal{T}_R^{\Subset}(P))\right)(n)\,.$$

We also show that the result of an IDE problem obtained through a correlated-calls transformation is a subset of the original IFDS result.

Lemma 5 (Precision). *For an IFDS problem P and all $n \in N^*$,*

$$\mathcal{U}^{\Subset}\left(\mathcal{R}_{IDE}(\mathcal{T}_R^{\Subset}(P))\right)(n) \subseteq \mathcal{R}_{IFDS}(P)(n)\,. \tag{7}$$

4.5 Correlated-Call Receivers

We will now show that in a correlated-calls transformation, it is enough to consider only some of the receivers of set R.

Definition 3. *If $r \in R$ is the receiver of at least two polymorphic call sites, then we call r a correlated-call receiver, and we define $R^{\Subset}$ as the set of all such receivers.*

We will show that it is sufficient for the correlated-calls micro-functions to be defined only on correlated-call receivers. Specifically, a "reduced" correlated-calls transformation that considers only correlated-call receivers in the micro-functions yields the same solution as the full correlated-calls transformation (i.e. no precision is lost).

Lemma 6. *Let P be an IFDS problem. Then*

$$\mathcal{U}^{\Subset}\left(\mathcal{R}_{IDE}\left(\mathcal{T}_{R^{\Subset}}^{\Subset}(P)\right)\right) = \mathcal{U}^{\Subset}(\mathcal{R}_{IDE}\left(\mathcal{T}_R^{\Subset}(P)\right))\,. \tag{8}$$

4.6 Efficiency

Both the IFDS and IDE algorithms have been proven to run in $O(ED^3)$ time [16, 18], where E is the number of edges in the (non-exploded) supergraph, and D is the size of the set of facts. The IDE algorithm may evaluate micro-functions up to $O(ED^3)$ times, so this running time must be multiplied by the cost of evaluating a micro-function. We show that the micro-functions in the correlated-calls IDE analysis can be evaluated in time $O(R^{\Subset}T)$, where $R^{\Subset}$ is the number of correlated-call receivers $R^{\Subset}$ and the T is the number of run-time types. Therefore, the overall worst-case cost of the correlated-calls IDE analysis is $O(ED^3R^{\Subset}T)$. In practice, $R^{\Subset}$ is much smaller than R, so Lemma 6 is significant for performance.

Specifically, the complexity proof for the IDE algorithm requires the implementation of the micro-functions to be *efficient* according to a list of specific criteria. Our micro-function implementation does satisfy the criteria:

Lemma 7. *The correlated-call representation of a micro function is efficient according to the IDE criteria [18] and the required operations on micro-functions can be computed in time $O(R^{\Subset}T)$.*

4.7 Implementation of the Correlated-Calls Analysis

We implemented the correlated-calls analysis in Scala [14]. Our implementation analyzes JVM bytecode compiled from input programs written in Java. We use WALA [5] to retrieve information about an input program, such as its control-flow supergraph and the set of receivers and their types. Since WALA does not contain an implementation of the IDE algorithm, we implemented it from scratch; we are working on contributing our infrastructure to WALA.

We tested our correlated-calls analysis using an IFDS taint-analysis as a client analysis. To this end, we converted the IFDS taint analysis into an IDE problem with an implementation of $\mathcal{T}^{\Subset}_{R^{\Subset}}$. We extensively tested the correlated-calls analysis to ensure that, in the absence of correlated calls, the analysis produces the same results as an IFDS-equivalent analysis, and that it produces more precise results in the presence of correlated calls as expected.

To evaluate the practicality of our approach, we applied two variants of the IFDS taint analysis to the SPEC JVM98 benchmarks: (i) an equivalent IDE taint analysis obtained using $\mathcal{T}^{\equiv}$, and (ii) an IDE taint analysis obtained using $\mathcal{T}^{\Subset}_{R^{\Subset}}$ that avoids imprecision due to correlated method calls.

The equivalence analysis is there for two reasons: (i) to explain how a correlated-calls-IDE problem can be derived from an IDE problem that has the same meaning as the original IFDS problem, and (ii) to provide a base line against which to compare the efficiency of the correlated-calls analysis. We compare the efficiency of the correlated-calls analysis against the equivalence-IDE analysis instead of the IFDS analysis because the time complexities of an IFDS and an equivalent IDE analysis are the same: an equivalent IDE analysis is just an IFDS analysis in which all edges are labeled with identity micro functions, and all operations on those functions are optimized to be constant-time.

The running times $t_{\Subset}$ of the correlated-calls and $t_{\equiv}$ equivalence analyses are shown in Table 1. In the table, N_r^* is the number of reachable nodes in the control-flow supergraph, and $N_r^{\#}$ the number of reachable nodes in the exploded supergraph.

Table 1: Running times of the analyses

Benchmark	$\boldsymbol{N_r^*}$	$\boldsymbol{N_r^{\#}}$	$\boldsymbol{t_{\equiv}}$	$\boldsymbol{t_{\Subset}}$
compress	2,155	24,730	0:00:02	0:00:04
db	2,285	22,938	0:00:06	0:00:12
jack	17,602	284,625	0:06:06	0:11:31
javac	40,430	510,810	0:46:06	1:45:57
jess	14,448	316,418	0:10:19	0:13:33
mpegaudio	11,959	224,886	0:01:57	0:00:54
mtrt	3,597	88,267	0:00:34	0:00:33
raytrace	3,597	88,267	0:00:38	0:00:37

The results suggest that the overhead of tracking correlated calls is acceptable. In particular, the correlated-calls analysis takes at most twice as long as the equivalence analysis. The absolute times range from a few seconds on the smaller SPEC programs to about two hours on `javac`.

Our implementation is a research prototype and many opportunities for optimization remain. For the specific combination of this IFDS client analysis and these benchmark programs, tracking correlated calls did not impact precision.

5 Related Work

The IFDS algorithm is an instance of the functional approach to data-flow analysis developed by Sharir and Pnueli [19]. IFDS has been used to encode a variety of data-flow problems such as typestate analysis [12, 23] and shape analysis [9]. IFDS has been used [2, 22] and extended [10] to solve taint-analysis problems.

Naeem and Lhoták [13] proposed several extensions of IFDS. In particular, they propose several techniques for improving the algorithm's efficiency, as well as a technique that improves expressiveness by extending applicability to a wider class of dataflow analysis problems. These extensions are orthogonal to, and could be combined with the approach presented in this paper. Our work differs from theirs by targeting analysis precision, not efficiency or expressiveness.

Bodden et al. [4] presents a framework for applying IFDS analyses to software product lines. Their approach enables the analysis of all possible products derived from a product line in a single analysis pass. Like our approach, their approach transforms IFDS problems to IDE problems. The micro-functions keep track of the possible program variations specified by the product line. Rodriguez and Lhoták evaluate a parallelized implementation of the IFDS algorithm using actors [17] that can take advantage of multiple processors.

The idea of using correlated calls to remove infeasible paths in data-flow analyses of object-oriented programs was introduced by Tip [21]. The possibility of using IDE to achieve this is mentioned, but not elaborated upon. Our work is the first to present and implement a concrete solution.

Recent work on correlation tracking for JavaScript [20] also eliminates infeasible paths. Instead of infeasible paths between dynamically dispatched method calls, their approach eliminates infeasible paths between reads and writes of different properties of an object. The approach differs from ours in that it targets points-to analysis rather than IFDS analyses, in that it targets infeasible paths due to different property names rather than different dynamically dispatched methods, and in that it employs context sensitivity to improve precision.

Our approach superficially resembles, but is orthogonal to, context sensitivity, including the CPA algorithm [1] and such variations as object sensitivity [11]. Context-sensitive points-to analysis is orthogonal to our work because it analyzes the flow of data (pointers), whereas we analyze control flow paths. Also, object-sensitive points-to analysis is flow-insensitive, while IFDS and IDE are flow-sensitive analyses. Note that our transformation only makes sense in a flow-sensitive setting since a flow-insensitive analysis already introduces many infeasible control flow paths.

It would be possible to simulate the effect of our correlated calls transformation in the following way inspired by context-sensitivity: we could re-analyze each method in a number of contexts. There would be a separate context for every possible assignment of concrete types to all of the pointers in the method that are used as receivers at a call site. The number of such contexts for each method would be $O(R^T)$, where R is the number of receiver pointers in the method and T is the number of possible concrete types that could be assigned to a receiver pointer. Our approach computes equally precise analysis results but avoids this exponential cost.

6 Conclusions

Previous algorithms for data-flow analysis are unable to avoid propagating data-flow facts along infeasible paths that arise in the presence of correlated polymorphic method calls. We present an approach for transforming an IFDS problem into an IDE problem in which path feasibility is encoded into functions associated with edges in an exploded control-flow supergraph. The solution to this IDE problem can be mapped back to the solution space of the original IFDS problem, and is more precise for some client programs because data flow along infeasible paths is prevented. We present a formalization of the transformation, prove its correctness, and briefly report on preliminary experiments with our prototype implementation. Full proof details are available in the Technical Report [15]. As future work, it is possible to adapt our approach to work on IDE problems. We would convert an initial IDE problem into a more complex IDE problem, such that the solution of the latter generates a more precise solution to former, by preventing data flow along infeasible paths.

Bibliography

[1] Agesen, O. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Stanford University, 1995.

[2] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y. L., Octeau, D., and McDaniel, P. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI'14*, page 29, 2014.

[3] Blackburn, S. M., Garner, R., Hoffmann, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A. L., Jump, M., Lee, H. B., Moss, J. E. B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*, pages 169–190, 2006.

[4] Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., and Mezini, M. SPLLIFT - statically analyzing software product lines in minutes instead of years. In *Software Engineering '14*, pages 81–82, 2014.

[5] Fink, S. and Dolby, J. WALA — the TJ Watson libraries for analysis. `http://wala.sourceforge.net`, 2012.

[6] Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., and Berg, R. Saving the world wide web from vulnerable JavaScript. In *ISSTA'11*, pages 177–187, 2011.

[7] Knoop, J. and Steffen, B. The interprocedural coincidence theorem. In *CC'92*, pages 125–140, 1992.

[8] Knoop, J., Steffen, B., and Vollmer, J. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst.*, (3):268–299, 1996.

[9] Kreiker, J., Reps, T. W., Rinetzky, N., Sagiv, M., Wilhelm, R., and Yahav, E. Interprocedural shape analysis for effectively cutpoint-free programs. In *Programming Logics, Springer LNCS Volume 7797*, pages 414–445, 2013.

[10] Lerch, J., Hermann, B., Bodden, E., and Mezini, M. FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases. In *FSE'14*, pages 98–108, 2014.

[11] Milanova, A., Rountev, A., and Ryder, B. G. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[12] Naeem, N. A. and Lhoták, O. Typestate-like analysis of multiple interacting objects. In *OOPSLA'08*, pages 347–366, 2008.

[13] Naeem, N. A., Lhoták, O., and Rodriguez, J. Practical extensions to the IFDS algorithm. In *CC'10*, pages 124–144, 2010.

[14] Odersky, M. Essentials of Scala. In *LMO'09*, page 2, 2009.

[15] Rapoport, M., Lhoták, O., and Tip, F. Precise data flow analysis in the presence of correlated method calls. Technical Report CS-2015-07, University of Waterloo, 2015.

[16] Reps, T. W., Horwitz, S., and Sagiv, S. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*, pages 49–61, 1995.

[17] Rodriguez, J. D. A concurrent IFDS dataflow analysis algorithm using actors. Master's thesis, University of Waterloo, 2010.

[18] Sagiv, S., Reps, T. W., and Horwitz, S. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT'95*, pages 651–665, 1995.

[19] Sharir, M. and Pnueli, A. Two approaches to interprocedural data flow analysis. *Program flow analysis: Theory and applications*, pages 189–234, 1981.

[20] Sridharan, M., Dolby, J., Chandra, S., Schäfer, M., and Tip, F. Correlation tracking for points-to analysis of JavaScript. In *ECOOP'12*, pages 435–458, 2012.

[21] Tip, F. Infeasible paths in object-oriented programs. *Sci. Comput. Program.*, 97:91–97, 2015.

[22] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. TAJ: effective taint analysis of web applications. In *PLDI'09*, pages 87–97, 2009.

[23] Zhang, X., Mangal, R., Grigore, R., Naik, M., and Yang, H. On abstraction refinement for program analyses in Datalog. In *PLDI'14*, page 27, 2014.