Overview of Comments

Data Flow Analysis in the Presence of Correlated 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

Frank wanted to discuss the title (M)

Abstract. We present a technique to improve the precision of data-flow analyses on object-oriented programs in the presence of *correlated calls*. Two method calls are correlated if they are polymorphic and are invoked on the same object. Correlated calls are problematic because they can make existing data-flow analyses consider certain infeasible data-flow paths as valid. This leads to loss in precision of the analysis solution.
We show how infeasible paths can be eliminated for *Inter-procedural Finite Distributive Subset* (IFDS) problems, a large class of data-flow analysis problems. We show how the precision of IFDS problems can be improved in the presence of correlated calls, by using the *Inter-procedural Distributive Environment* (IDE) algorithm to eliminate infeasible paths. Using IDE, we eliminate the infeasible paths and obtain a more precise result for the original IFDS problem.
Our analysis is implemented in Scala, using the WALA framework for static program analysis on Java bytecode.

1 Introduction

Data-flow analysis computes an approximation of how values may flow through a program, and has applications in compiler optimization, programming tools, and computer security, and many other areas. Data-flow analyses operate on *control-flow graphs* (CFGs) that model the order in which the instructions of a program are executed. This is typically done by associating *flow functions* that represent how data is propagated with the edges of the control-flow graph. A *confluence operator* specifies how the data facts that have been computed along different paths should be merged when the paths join.

If we introduce this abbreviation, should we replace the uses of "control-flow graph" with "CFG" in the paper? (M)

Since a control-flow graph 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 critically on its ability to detect and disregard such infeasible paths. The popular *Interprocedural Finite Distributive Subset* (IFDS) algorithm by Reps, Horwitz, and Sagiv [17] is a general data-flow analysis algorithm for computing solutions to standard finite distributive data-flow problems such as reaching definitions, available expressions, and taint analysis. A distinguishing characteristic of IFDS is that it avoids infeasible interprocedural paths in which calls and returns to/from functions are not properly matched. Sagiv, Reps, and Horwitz also presented the *Interprocedural Distributive Environment* (IDE) algorithm [19] that similarly only considers properly matched call/return edges, but that supports a broader range of dataflow problems by expanding the domain of flow functions to *environments* that go beyond the data-flow facts considered by IFDS.

We should define somewhere what taint analysis is, since it's currently commented out. **(M)**

This paper presents an approach to dataflow 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. In such cases, if the method calls are polymorphic (i.e., if they dispatch to different method definitions depending on the type of the receiver expression at run time), then their dispatch behaviors will be correlated. A recent paper [22] identified this problem but did not present a concrete solution or algorithm, and we are not aware of any existing dataflow analysis that is capable of avoiding infeasible paths that arise in the presence of correlated method calls.

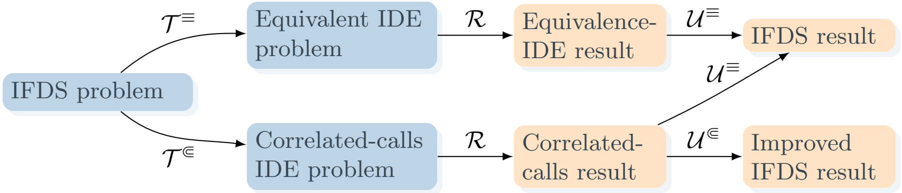


Fig. 1: Transformations between IFDS and IDE problems and their results

The approach taken in our work is to transform a standard 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 dataflow domain of the original IFDS problem.

We present a formalization of the transformation and prove its correctness as follows. First, we derive an "Equivalence IDE problem"from the original IFDS problem by associating the identity environment functions with all edges, and show that the solution to this Equivalence IDE problem can be mapped back to the solution of the original IFDS problem. Then, we derive a "Correlated Calls IDE Problem" from the original IFDS problem and show how a solution to this problem can be mapped to the solution to the original IFDS problem, but also how a more precise IFDS result can be derived from it. We also show that the correlated-calls analysis is sound, i.e., that it never considers concrete execution paths as infeasible. This is illustrated schematically in Figure 1[3].

The IDE paper already states that each IFDS problem can be solved with IDE. I wonder if this sounds like the equivalence transformation was our contribution. We just use the equivalence transformation to explain the CC transformation, and for the proofs. **(M)**

Do we need to mention the imprecise map from the CC-IDE result to the IFDS result? It doesn't really seem relevant, and could be confusing. **(O)**

would it make sense to add an edge in Figure 1 from the IFDS problem to the IFDS result? **(F)**

[3] The labels $\mathcal{R}$, $\mathcal{U}^{\equiv}$, $\mathcal{U}^{\Subset}$, $\mathcal{T}^{\equiv}$, and $\mathcal{T}^{\Subset}$ on edges in Figure 1 reflect a number of mappings and projections that will be defined in Section **▶ref◀** and that can be ignored here.

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]. We also report on preliminary experiments in which our correlated-calls transformation is applied to an IFDS formulation of a simple taint analysis. Our results show that solving the resulting IDE problem avoids infeasible paths due to correlated calls as expected.

In summary, the contributions of this paper are as follows:

- We present a general approach for transforming IFDS problems into corresponding IDE problems that avoid infeasible paths due to correlated method calls and prove its correctness.
- We implemented the approach in Scala, on top of the WALA program analysis framework and report on preliminary experiments.

The remainder of this paper is organized as follows. Section 2 presents a motivating example. Section 3 reviews the IFDS and IDE algorithms. Section **??** presents the correlated-calls transformation and a proof of its correctness. The implementation of our approach and preliminary experiments are discussed in Section 5. Related work is discussed in Section ►**ref**◄. Finally, conclusions and directions for future work are presented in Section ►**ref**◄.

2 Motivating Example

This section is current under revision. Please do not edit.. **(F)**

Consider a call site $r.m()$ in an object-oriented programming language, where the variable r is the *receiver* variable of the call site and m is the name of the invoked method[4]. In the rest of the paper, we use the general term *receiver* to mean a receiver variable. At runtime, the actual method that will be invoked by the call site depends on the runtime type of the object referenced by r. If the call site $r.m()$ can be associated with more than one method at compile time, we will say that the call site is *polymorphic*.

For example, in Listing 3, it is not possible to infer statically whether the runtime type of the variable `a` in the `main` method is `A` or `B`. The call `a.foo()` can be dispatched to either `A.foo` or `B.foo`, and `a.bar(v)` can be dispatched to either `A.bar` or `B.bar`. A concrete execution path for the main method might therefore go through `A.foo` and `A.bar`, or through `B.foo` and `B.bar`. However, there cannot be an execution path through `A.foo` and `B.bar` or through `B.foo` and `A.bar`.

We call the invocations to methods `foo` and `bar` *correlated*. More generally, correlated calls occur when more than one polymorphic call is invoked on the same receiver variable.

Suppose we wanted to perform a taint analysis on the program in Listing 3. Most dataflow-analysis algorithms, including IFDS, would conservatively assume

[4] We assume an internal representation of the program in which for each call site $e_r.m()$, the expression e_r has been evaluated to the variable r.

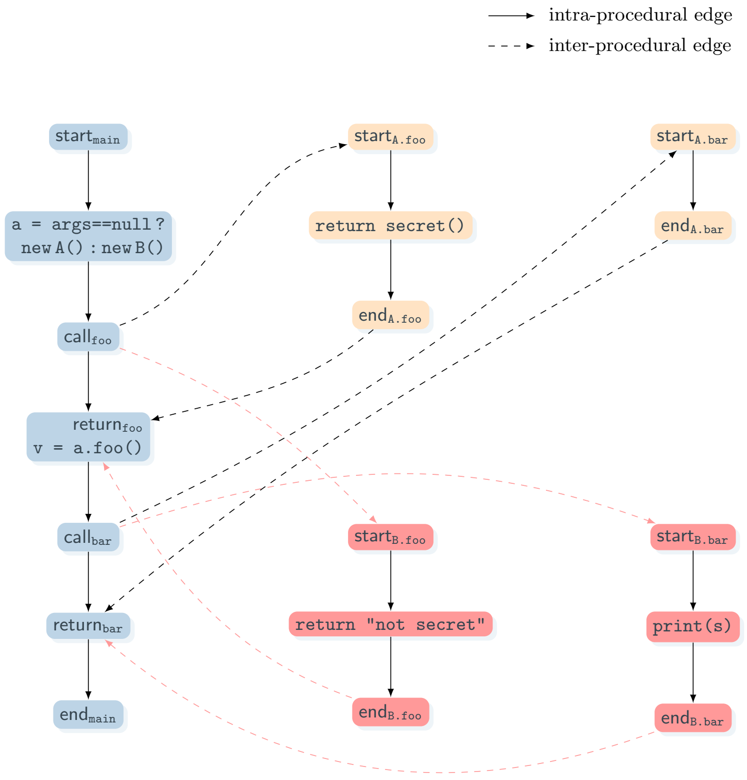

intra-procedural edge
inter-procedural edge
$\text{start}_{\texttt{main}}$
a = args==null ? new A() : new B()
$\text{call}_{\texttt{foo}}$
$\text{return}_{\texttt{foo}}$ v = a.foo()
$\text{call}_{\texttt{bar}}$
$\text{return}_{\texttt{bar}}$
$\text{end}_{\texttt{main}}$
$\text{start}_{\texttt{A.foo}}$
return secret()
$\text{end}_{\texttt{A.foo}}$
$\text{start}_{\texttt{A.bar}}$
$\text{end}_{\texttt{A.bar}}$
$\text{start}_{\texttt{B.foo}}$
return "not secret"
$\text{end}_{\texttt{B.foo}}$
$\text{start}_{\texttt{B.bar}}$
print(s)
$\text{end}_{\texttt{B.bar}}$

Fig. 2: An example supergraph for Listing **??**

```
class A {
  String foo {
    return secret();
  }

  void bar(String s) {}
}

class B extends A {
  String foo {
    return "not␣secret";
  }

  void bar(String s) {
    System.out.println(s);
  }
}

class Main {
  public static void main(String[] args) {
    A a = args == null ? new A() : new B();
    String v = a.foo();
    a.bar(v);
  }
}
```

Fig. 3: Example program containing correlated calls

that the call `a.bar` could be dispatched to both `A.bar` and `B.bar`, independently of what `a.foo` had been dispatched to in the previous line.

As a result, such an analysis would consider a path through `A.foo` and `B.bar` feasible. This means that the variable `v` would be considered secret. We would conclude that a secret value is passed to `B.bar` and printed to the user. In other words, we would consider the program to leak secret information, which it does not do in any concrete execution.

3 Background

This section defines basic 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, and which contains an edge from n_1 to n_2 whenever n_2 may execute immediately after n_1. A control-flow graph has a distinguished *start node* start_p and *end node* end_p. The *control-flow supergraph* of a program contains the control-flow graphs 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.

do we really need to extra "return nodes" that the IFDS/IDE papers add? **(O)**

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 kind of inverse τ 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: $\tau(s, f) = \{t \mid \mathsf{lookup}(s, t) = f\}$.

do we ever use src() and end()? **(O)**

We denote the source and end nodes of a graph edge e as $\mathsf{src}(e)$ and $\mathsf{end}(e)$.

A path through the control-flow supergraph is *valid* if every end-to-return edge on the path returns to the site of the most recent unmatched call. The set of all valid paths from the program entry point to a node n is denoted $\mathsf{VP}(n)$.

Should we give a formal definition? **(O)**

The definitions that Marianna had were for *complete* lattices/semilattices. All of the lattices in the context of IFDS/IDE are/must be complete. I'll just say that here. **(O)**

A *complete lattice* is a partially ordered set $(S, \sqsubseteq)$ in which every subset has a least upper bound and a greatest lower bound. A *complete meet semilattice* is a partially ordered set in which every subset need only have 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. Since all of the (semi)lattices discussed in this paper are required to be complete, we will henceforth leave out the *complete* qualifier.

In this paper, we denote a map m as a set of pairs of keys and values, in which each key appears at most once. For any map m, $m(k)$ is the value paired with the key k in m. 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 [17] is a precise and efficient algorithm for data-flow analysis that has been used to solve a variety of data-flow analysis problems [4,13,9,23]. The IFDS framework is an instance of the *functional approach* to data-flow analysis [20] because it constructs summaries of the effects of called procedures. The IFDS framework is applicable to *inter-procedural* data flow problems whose domain is *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 imprecise due to invalid paths, general but do not run in polynomial time [7,20], or handle a very specific set of problems [8].

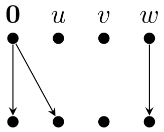
The input to the IFDS algorithm is specified as $(G^*, D, F, M_F, \sqcap)$, where $G^* = (N^*, E^*)$ is the supergraph of the input program with nodes N^* and edges E^*, D is a finite set of *data-flow facts*, F is a set of distributive dataflow functions of type $2^D \to 2^D$, $M_F : E^* \to F$ assigns a dataflow 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 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 represent both the input dataflow functions and the summary functions that it computes for called procedures. Specifically, the representation relation R_f of a function f is defined as:

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

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 can be visualized with the following graph:

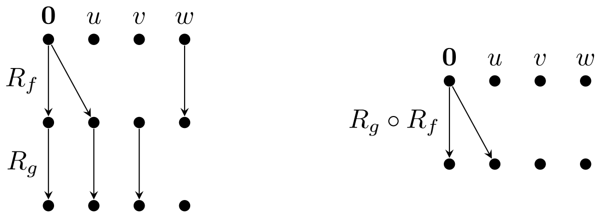


The representation relation decomposes a flow function into functions (edges) that operate on each fact individually. This is possible due to distributivity: applying the flow function to a set of facts is equivalent to applying it on each fact individually and taking the union of the results.

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.

The example used to say $R_f \circ R_g$, but I believe it showed $R_g \circ R_f$. I fixed it, but please check that this is correct. **(O)**

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 the following graph:



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 dataflow function for a control flow path corresponds to finding reachable nodes in a graph composed from the representation relations of the dataflow 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 $(n \to n') \in E^*$, the exploded supergraph contains a set of edges $(n, d_i) \to (n', d_j)$, which form the representation relation of the dataflow function $M_F(n \to n')$. 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 $(n_0, d_0) \to (n_1, d_1) \to \dots \to (n_m, d_m)$ and $n_0 \to n_1 \to \dots \to n_m$ is a valid path).

Tell the reader here to ignore the labels on the edges of the graph? **(M)**

Example 3. The exploded supergraph for Listing 3 is shown in Figure 4.We can see that there is a realizable path 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 at that node, `s` is considered secret.

A practical implementation of the IFDS algorithm generally takes as input a representation of the exploded supergraph edges $E^\sharp$ instead of explicit dataflow functions M_F. This is convenient because the two are equivalent in expressiveness, and because the IFDS algorithm works internally with the exploded supergraph. More specifically, the input generally provides a function $f : N^* \times D \times N^* \to 2^D$. Given a (non-exploded) supergraph edge $n \to n'$ and a dataflow fact d, $f(n, d, n')$ returns the set of all d' such that the exploded supergraph contains the edge $(n, d) \to (n', d')$. For convenience, the function f can be split up into separate functions that handle the cases when the $n \to n'$ is an intraprocedural edge, a call-to-start edge, or an end-to-return edge.

3.3 IDE

The IDE algorithm [19] extends IFDS to *inter-procedural distributive environment* problems. An *environment* problem is an analysis whose dataflow 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 dataflow 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 dataflow 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 $\mathsf{MVP}_{\mathsf{Env}}(n) = \bigsqcap_{q \in \mathsf{VP}(n)} M_{\mathsf{Env}}(q)(\top)$, where 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 functions 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. Let $\Omega = \lambda d.\top$ be the environment that maps every element of D to $\top$. Thanks to distributivity, every environment transformer $t : \mathsf{Env}(D, L) \to \mathsf{Env}(D, L)$ can be decomposed into its effect on Ω and on a set of environments $\Omega[d_i \to l]$ that map every element except one (d_i) to $\top$:

$$t(m)(d_j) = \lambda l.t(\Omega)(d_j) \sqcap \bigsqcap_{d_i \in D} \lambda l.t(\Omega[d_i \to l])(d_j).$$

The functions $\lambda l. \cdots$ in this decomposition are the micro-functions that appear on the edges of the pointwise representation edges from Λ to each d_j and from

[5] The IDE literature uses the symbol Λ for the node that is denoted **0** in the IFDS literature.

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 $\lambda l.\top$.

Discuss the micro-functions in the example exploded supergraph in Figure 3. **(O)**

The meet of two environment transformers t_1, t_2 can be computed by taking 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 way 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 (except that the edges are labelled with micro-functions). For each pair (n, d) of node and fact, the algorithm 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 dataflow functions for individual instructions, and that supports computing the meet and composition of micro-functions.

Similar to the IFDS algorithm, a practical implementation of the IDE algorithm requires the input dataflow functions to be provided in their pointwise representation as exploded supergraph edges labelled with micro-functions. Specifically, the input is generally provided as a function $f : N^* \times D \times N^* \rightarrow (D \rightarrow F)$, where F is the set of representations of micro-functions from L to L. Given a (non-exploded) supergraph edge $n \rightarrow n'$ and a dataflow fact d, $f(n, d, n')$ returns a map that gives for each fact d' the micro-function f that appears on the exploded supergraph edge $(n, d) \rightarrow (n', d')$. Like in the IFDS algorithm, the function f can be split up into separate functions that handle the cases when the $n \rightarrow n'$ is an intraprocedural edge, a call-to-start edge, or an end-to-return edge.

Should we cite the "Practical extensions" paper here that talks about the on-demand IFDS algorithm? **(M)**

In the next section, we talk about `EdgeFn`, which is inconsistent with how we define the functions to be implemented here. We need to make the two sections consistent, one way or the other. **(O)**

In Figure 4, should we say that ψ corresponds to the function's return value? **(M)**

we agreed to move this figure to the initial example section **(O)**

Move Figure 4 up to where it's mentioned first? If yes, the caption won't make sense. **(M)**

4 Correlated Calls Analysis

The correlated-calls analysis is defined as a transformation from an arbitrary IFDS problem to a corresponding IDE problem. The solution of the IDE problem is converted to a solution of the original IFDS problem. The converted IFDS result can be more precise than the original IFDS result because it avoids infeasible paths corresponding to correlated calls.

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

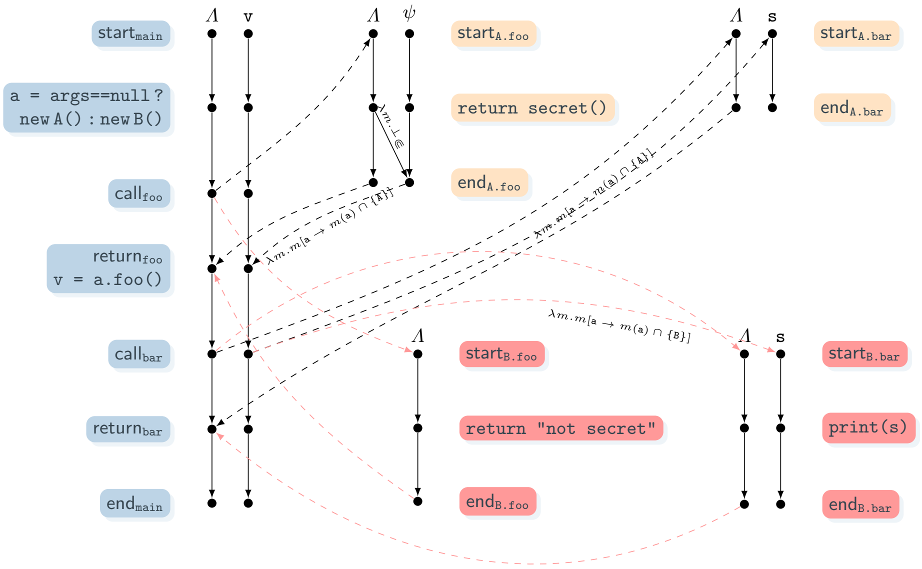


Fig. 4: An example program demonstrating correlated-call edge functions on the Λ-node path for Listing 3. All non-labeled edges are implicitly labeled with identity functions id. The variable corresponding to an initial secret value is denoted as ψ.

4.1 Transformations from IFDS to IDE

Let $G^{\#}$ be the exploded supergraph of an IFDS problem. A *transformation* $\mathcal{T} : (G^{\#}) \to (G^{\#}, L, \mathsf{EdgeFn})$ converts it into an equivalent IDE problem. We consider two transformations:

- an equivalence transformation $\mathcal{T}^{\equiv}$ (pronounced “t-equiv”) that generates IDE problems with the same precision as the original IFDS problem, and
- a correlated-call transformation $\mathcal{T}^{\Subset}$ (pronounced “t-c-c”) that generates IDE problems that exclude infeasible paths.

Both transformations keep the exploded supergraph $G^{\#}$ the same, and only generate different edge functions.

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

$$\mathsf{EdgeFn}^{\equiv} = \begin{cases} \lambda e \,.\, \lambda m \,.\, \bot & \text{if } d_1(e) = \Lambda \text{ and } d_2(e) \neq \Lambda; \\ \lambda e \,.\, \mathsf{id} & \text{otherwise,} \end{cases} \tag{1}$$

where $d_1(e)$ is the source fact of an edge e and $d_2(e)$ is its target fact. At a “diagonal” edge from a Λ-fact to a non-Λ-fact d, the micro function returns $\bot$ to make the fact d reachable. All other micro-functions are the identity. The equivalence transformation is thus defined as: $\mathcal{T}^{\equiv}((G^{\#})) = (G^{\#}, L^{\equiv}, \mathsf{EdgeFn}^{\equiv})$.

“the identity function” or is it okay ot just say “the identity”? **(M)**

Correlated-Calls Transformation In the correlated-calls transformation, 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 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 any feasible path if its corresponding lattice element maps any receiver to the empty set of types.

What if r points to null? **(O)**

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:

define $n_1(e)$ globally somewhere early on, and consider using a more decriptive name like src instead of n_1 **(O)**

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:*

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

$$\begin{cases} m[r \to m(r) \cap \tau(s_{\mathcal{F}},\, f)], & \text{if } e \text{ is a call-start edge, } r.c() \text{ is the call site at } \mathsf{src}(e),\ f \text{ is the called procedure with signature } s_{\mathcal{F}}, \text{ and } r \in S; \\ m[r \to m(r) \cap \tau(s_{\mathcal{F}},\, f)] \\ [v_1 \to \bot_T] \ldots [v_k \to \bot_T], & \text{if } e \text{ is an end-return edge, } r.c() \text{ is the call corresponding to the return node at } \mathsf{end}(e),\ f \text{ is the called method with signature } s_{\mathcal{F}},\ v_1, \ldots, v_k \in S \text{ are the local variables in } f, \text{ and } r \in S; \\ m\,[r \to \bot_T]\,, & \text{if } \mathsf{src}(e) \text{ contains an assignment for } r \in S; \\ m & \text{otherwise.} \end{cases}$$

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 Λ-edge, the micro function is the identity. All other functions can be described with ε_S. On a “diagonal” edge from Λ to a non-Λ fact that corresponds to some data flow fact becoming reachable, $\varepsilon_S(e)$ is applied to the initial map $\bot_{\Subset}$ that conservatively allows every receiver to point to an object of any 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:*

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

Example 4. Consider the program from Listing 3, whose exploded supergraph appeared in Figure 4.

Returning a secret value in method `A.foo` creates a “diagonal” edge from the Λ-fact to the secret fact ψ. The diagonal edge is labeled with the micro function $\lambda m\,.\,\bot_{\Subset}$. Thus, at the end node of the method, 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 for the receiver `a` is restricted by the micro function $\lambda m\,.\,m[\mathtt{a} \to m(\mathtt{a}) \cap \{\mathtt{A}\}]$ on the edge corresponding to the assignment of the return value ψ to `v`.

Similarly, on the call-start edge from `main` to `B.bar`, the possible types of the receiver `a` are further restricted by the micro-function $\lambda m\,.\,m[\mathtt{a} \to m(\mathtt{a}) \cap \{\mathtt{B}\}]$. 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 the receiver `a`, indicating that this data flow path that would result in an information leak is actually infeasible.

Finally, the correlated-calls transformation is defined as $\mathcal{T}^{\Subset}_S((G^{\#})) = \left(G^{\#},\, L^{\Subset}_S,\, \mathsf{EdgeFn}^{\Subset}_S\right)$.

4.2 Converting IDE Results to IFDS Results

For each program point n, the result of an IFDS analysis gives a set of facts d that may be reached at n. The result of an IDE analysis pairs each such fact d with a lattice element ℓ. Formally, for an IFDS problem P, the result $\mathcal{R}_{\text{IFDS}}$ has type $N \to D$. Similarly, for an IFDS problem Q, the result $\mathcal{R}_{\text{IDE}}$ has type $N \to D \times L$.

Recall that in the equivalence transformation lattice $L^{\equiv}$, $\bot$ means reachable and $\top$ means unreachable. Therefore, a result ρ of the equivalence IDE analysis is converted to an IFDS result as follows: $\mathcal{U}^{\equiv}(\rho) = \lambda n.\{d \mid \rho(n) = (d, \top)\}$.

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, a result ρ of the correlated-calls IDE analysis is converted to an IFDS result as follows: $\mathcal{U}^{\Subset}(\rho) = \lambda n.\{d \mid \rho(n) = (d, \ell), \forall r \,.\, \ell(r) \neq \top\}$.

4.3 Implementation of Correlated Calls Micro-Functions

move material from following section here **(O)**

Conceptually, micro-functions are functions from L to L, where L is the IDE lattice, either $L^{\equiv}$ or $L^{\Subset}$ in our context. However, the IDE algorithm requires an efficient representation of micro-functions. The chosen representation needs to support the basic micro-functions that we presented in Section 4.1. The representation must also 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. We define the meaning $[\![\langle I, U\rangle]\!]$ of a representation $\langle I, U\rangle$ as follows: $[\![\langle I, U\rangle]\!] = \lambda m \,.\, \lambda r \,.\, (m(r) \cap I(r)) \cup U(r)$. In words, the micro-function represented by $\langle I, U\rangle$ takes an existing map m from receivers to sets of types, and returns a map that maps each receiver r to the set $m(r)$ given by the original map m intersected with $I(r)$ and unioned with $U(r)$.

All of the basic micro-functions defined in Definition 1 can be expressed in this representation.

The implementation of function application follows directly from the definition of the representation: $\langle I, U\rangle\,(m) = \lambda r \,.\, (m(r) \cap I(r)) \cup U(r)$.

To compare two micro-functions for equality, it suffices to compare the corresponding sets $I(r)$ and $U(r)$ for all receivers r. The following lemma shows that this implementation of comparison corresponds to equality of the represented micro-functions:

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

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

Prove this lemma in the appendix. The parts of the proof can be found in section 5. **(O)**

The normalization function $\mathcal{N}$ was only needed in composition. I therefore "inlined" it into composition to save space. This makes composition a bit harder to intuitively verify, but saves complexity here. The verification can be in the proof in the supplementary appendix. **(O)**

The composition of two micro-function representations is defined as follows: $\langle I, U\rangle \circ \langle I', U'\rangle = \langle \lambda r\,.\,(I(r) \cap I'(r)) \cup U(r),\ \lambda r\,.\,(I(r) \cap U'(r)) \cup U(r)\rangle$. The following lemma shows that this implementation corresponds to the composition of the denoted functions:

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

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

Prove this lemma in the appendix. The parts of the proof can be found in section 5. **(O)**

The meet of two micro-function representations is defined as follows: $\langle I, U\rangle \sqcap \langle I', U'\rangle = \langle \lambda r\,.\,I(r) \cup I'(r),\ \lambda r\,.\,U(r) \cup U'(r)\rangle$. The following lemma shows that this implementation corresponds to the meet of the denoted functions:

Lemma 3. *For any pair of micro-function representations $\langle I, U\rangle$, $\langle I', U'\rangle$,*

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

Prove this lemma in the appendix. The parts of the proof can be found in section 5. **(O)**

4.4 Theoretical Results

In the following lemma we show that the result of an IDE problem obtained through a correlated-calls transformation is a subset of the original IFDS result.

Lemma 4 (Precision). *For an IFDS problem P and all $n \in N^*$,*

$$\mathcal{U}^{\Subset}\left(\mathcal{R}(\mathcal{T}_R^{\Subset}(P))\right)(n) \subseteq \mathcal{R}_{IFDS}(P)(n)\,. \tag{4}$$

Sagiv et al. define a notion of an "efficient" implementation of micro-functions. If the micro-functions are "efficient", then the asymptotic complexity bounds proven by Sagiv hold. Our implementation is "efficient" if we assume that the number of correlated receivers is a constant independent of the size of the program. That seems a stretch. If it's linear in the size of the program, then we would need to add a factor to their complexity bounds. Their bound is $O(ED^3)$, while ours would be $O(ED^3R)$. But then we can't directly use their complexity proof; we would have to adapt it. I'm leaning towards leaving out any mention of "efficient" implementation. **(O)**

We will next show that our analysis is sound, i.e. that the result of an IDE problem obtained through a correlated-calls transformation removes only facts that occur on infeasible paths.

Lemma 5 (Soundness). *Let $p = [\mathit{start}_{main}, \ldots, n]$ be a concrete execution path, and let $d \in D$. If $d \in M_F(p)(\varnothing)$, then*

$$d \in \mathcal{U}^{\Subset}\left(\mathcal{R}(\mathcal{T}_R^{\Subset}(P))\right)(n)\,. \tag{5}$$

Should we say $\mathcal{R}_{\mathrm{IDE}}$ instead of $\mathcal{R}$ everywhere, to be consistent with $\mathcal{R}_{\mathrm{IFDS}}$? **(O)**

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.

should this be a section or subsection? should it be moved somewhere else (or maybe not included at all)? **(O)**

Definition 3. *Let c_1 and c_2 be two call sites on a receiver $r \in R$. If both call sites are polymorphic, then we say that r is a correlated-call receiver.*

In other words, a correlated-call receiver is a receiver that has at least two polymorphic call invocations. We will denote the set of correlated-call receivers as $R^{\Subset}$.

We will describe a "reduced" correlated-calls transformation in which we only consider receivers from $R^{\Subset}$ and ignore other receivers of R. We will show that IDE problems obtained through ordinary and reduced correlated-calls transformations yield the same results. In other words, we show that if a correlated calls analysis considers only correlated-call receivers, no precision is lost.

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

$$\mathcal{U}^{\Subset}\left(\mathcal{R}\left(\mathcal{T}^{\Subset}_{R^{\Subset}}(P)\right)\right) = \mathcal{U}^{\Subset}(\mathcal{R}\left(\mathcal{T}^{\Subset}_{R}(P)\right)). \tag{6}$$

To summarize, Lemma 5 shows that the result $\mathcal{R}_{\Subset}$ of a correlated-calls analysis is sound since it overapproximates the data flow of all possible concrete execution paths. We have also shown in Lemma 4 that the correlated-calls analysis improves the precision of the original IFDS result $\mathcal{R}_{\text{IFDS}}$, because the correlated-calls result $\mathcal{R}_{\Subset}$ underapproximates an equivalence-IDE result $\mathcal{R}_{\equiv} = \mathcal{R}_{\text{IFDS}}$. Finally, we showed in Lemma 6 that a correlated-call transformation to IDE that considers only correlated-call receivers $R^{\Subset}$ achieves the same result $\mathcal{R}_{\Subset}$ that is obtained when considering all receivers R.

5 Evaluation

This section discusses implementation aspects of the correlated-calls analysis and presents experimental results.

5.1 Implementation of the Analysis

The correlated-calls analysis was implemented in the Scala programming language [16]. We chose Java as the target language for client programs of the analysis. To retrieve information about an input program, such as its control-flow supergraph or the set of receivers and their types, we used the WALA framework for static analysis on Java bytecode [5].

Since WALA currently only contains an implementation of IFDS, we implemented IDE from scratch. Instead of using WALA's IFDS implementation, to run an IFDS problem, we converted it to an IDE problem and used our own IDE solver.

Taint Analysis Using this representation of an IFDS problem, we implemented an IFDS problem instance for taint analysis. We used it as a sample IFDS problem on which to evaluate the correlated-calls-IDE construction.

Let N^* be the control-flow supergraph of a program and D the set of the program variables. Let $\mathsf{encl}(n)$ be a function that returns the enclosing method of a node $n \in N^*$. Finally, let the function $r_m : D \to 2^D$ be defined as follows:

$$r_m(d) = \begin{cases} \varnothing & \text{if } d \text{ is a local variable in method } m, \\ \{d\} & \text{otherwise.} \end{cases} \tag{7}$$

When defining the flow functions for a taint analysis, we will use r_m to avoid the propagation of local variables, as shown below.

For a fact $d_1 \in D \cup \{\mathbf{0}\}$ and two nodes $n_1, n_2 \in N^*$, the simplified[7] version of flow functions for a taint-analysis looks as follows.

If n_1 is a call node that calls method m, and n_2 is m's start node,

$$\mathsf{call\text{-}start}(n_1, d_1, n_2) = \begin{cases} r_{\mathsf{encl}(n_1)}(d_1) \cup \{v\} & \text{if } a \text{ is the } i\text{th argument of the call,} \\ & d_1 = a, \text{ and } v \text{ is the } i\text{th parameter} \\ & \text{of } m; \\ r_{\mathsf{encl}(n_1)}(d_1) & \text{otherwise.} \end{cases}$$

If n_1 is a call node with corresponding return node n_2,

$$\mathsf{call\text{-}return}(n_1, d_1, n_2) = \begin{cases} \{d_1\} & \text{if } d_1 \text{ is a local variable in } \mathsf{encl}(n_1), \\ \varnothing & \text{otherwise.} \end{cases}$$

If c is a call node calling method m, n_1 is m's end node, and n_2 is c's return node,

$$\mathsf{end\text{-}return}(n_1, d_1, n_2) = \begin{cases} r_{\mathsf{encl}(n_1)}(d_1) \cup \{x\} & \text{if } n_1 \text{ is a return statement} \\ & \text{returning } v,\ n_2 \text{ is an assignment} \\ & \text{with left-hand side } x, \text{ and } d_1 = v; \\ r_{\mathsf{encl}(n_1)}(d_1) & \text{otherwise.} \end{cases}$$

Otherwise,

$$\mathsf{default}(n_1, d_1, n_2) = \{d_1\}.$$

Example 5. Consider the supergraph in Figure **??**. The call-to-start flow function from method `main` to `f` looks as follows:

$$\begin{aligned} \mathsf{call\text{-}start}(\,\mathsf{call}_{\mathtt{A.f}}\,, \mathsf{a}, \,\mathsf{start}_{\mathtt{f}}\,) &= r_{\mathtt{main}}(\mathsf{a}) \cup \{\mathsf{s}\} \\ &= \{\mathsf{s}\}. \end{aligned}$$

We can see that correspondingly, the exploded supergraph contains an edge from $(\,\mathsf{call}_{\mathtt{A.f}}\,, \mathsf{a})$ to $(\,\mathsf{start}_{\mathtt{f}}\,, \mathsf{s})$.

[7] For simplicity, the shown flow functions do not account for different Java-specific features such as arrays, fields, operations on strings, etc.

IDE The correlated-calls analysis was implemented as an IDE problem instance.

We defined an IDE problem in the same way as an IFDS problem, except that the IDE flow functions are of type

$$(N \times D \times N) \to 2^{D \times (L \to L)}.$$

With the new flow functions, we can implement a labeled exploded supergraph, since the new flow functions return a set of facts that are paired with micro functions.

For example, if Q is an IDE problem, then the call-to-start flow function for Q is defined as follows:

$$\begin{aligned}&\textsf{call-start}^{Q}(n_1,\, d_1,\, n_2)\\ =&\Big\{(d_2,\, f) \,|\, d_2 \in D,\, f \in L^{Q} \to L^{Q} :\, \textsf{EdgeFn}^{Q}((n_1,\, d_1),\, (n_2,\, d_2)) = f\Big\}\,.\end{aligned}$$

The other flow functions are defined analogously.

5.2 Testing

In this section we assess the correctness and effectiveness of the correlated-calls analysis.

Conversion from IFDS to IDE We implemented the equivalence transformation $\mathcal{T}^{\equiv}$ and the correlated-calls transformation $\mathcal{T}^{\Subset}_{R^{\Subset}}$ from IFDS to IDE described in Section 4.1. To run an IFDS problem, we converted it to an IDE problem using $\mathcal{T}^{\equiv}$ and $\mathcal{T}^{\Subset}_{R^{\Subset}}$ and used our IDE analysis algorithm to run the latter.

Given an IFDS problem described with IFDS flow functions, an equivalence transformation creates an IDE problem described with the following IDE flow functions:

$$\begin{aligned}\textsf{call-start}^{\equiv}(n_1,\, d_1,\, n_2) =&\{(d_2,\, \epsilon(d_1,\, d_2)) \,|\, d_2 \in \textsf{call-start}(n_1,\, d_1,\, n_2)\}\\ \textsf{call-return}^{\equiv}(n_1,\, d_1,\, n_2) =&\{(d_2,\, \epsilon(d_1,\, d_2)) \,|\, d_2 \in \textsf{call-return}(n_1,\, d_1,\, n_2)\}\\ \textsf{end-return}^{\equiv}(n_1,\, d_1,\, n_2) =&\{(d_2,\, \epsilon(d_1,\, d_2)) \,|\, d_2 \in \textsf{end-return}(n_1,\, d_1,\, n_2)\}\\ \textsf{default}^{\equiv}(n_1,\, d_1,\, n_2) =&\{(d_2,\, \epsilon(d_1,\, d_2)) \,|\, d_2 \in \textsf{default}(n_1,\, d_1,\, n_2)\}\,,\end{aligned}$$

where ϵ is the bottom function on an edge from a $\mathit{\Lambda}$-fact to a non-$\mathit{\Lambda}$-fact, and the identity function otherwise:

$$\epsilon(d_1,\, d_2) = \begin{cases}\lambda l \,.\, \bot & \text{if } d_1 = \mathit{\Lambda} \text{ and } d_2 \neq \mathit{\Lambda};\\ \textsf{id} & \text{otherwise.}\end{cases}$$

We also implemented a correlated-call transformation from IFDS into IDE problems that consider correlated calls. This transformation is described in Section 4.1.

Regression Testing We used regression tests to assess the correctness of the implemented analyses. Each test involves running a certain analysis on one input Java program.

IDE-Implementation Correctness To test the correctness of the IDE algorithm implementation, we implemented a copy-constant-propagation IDE problem [19]. In a copy-constant propagation analysis, a variable is considered constant if it is assigned a constant literal or another variable that is also a constant. For example, in a program

```
int a = 1;
int b = a;
int c = a + b;
int d = a + 2;
```

a and b are considered constant, but c and d are not (although d would be considered constant in linear-constant propagation).

We tested the propagation of constants on different intra- and inter-procedural data-flow paths, in parameter passing, and in conditional branches. Each regression test contained assertions of the form “at the end of method m, variable with name x should be (not) constant”.

We also tested the implementation of the IDE algorithm on an IDE problem generated by conversion from an IFDS problem.

To do that, we implemented an IFDS instance for taint analysis.

Recall from Section 3.2 that taint analysis aims to discover variables that are secret at a given program point called a sink.

We used assertions of the form “at program statement n, variable x should be (not) secret” by defining the sink of a secret value through special `isSecret` and `notSecret` methods. Those methods asserted that the parameter passed to them is secret and not secret, respectively. To define a source secret value we created a static `secret()` method that returned a string.

Example 6. Listing 5 illustrates the use of the `isSecret` and `notSecret` assertions.

We tested data flow through

- method calls and returns;
- conditional branches and loops, including nested constructions, the ternary operator, and `switch` statements;
- arrays and fields[8];
- static and instance class members;

[8] In Java, arrays are allocated on the heap, and array elements can be aliases of each other. Hence, if any array element gets assigned a secret value, we considered all elements of any `String` or `Object` array in the program secret. For the same reason, if a field `f` of an object of class `A` is assigned a secret value, then we considered the field `f` of any object of class `A` secret.

```
public static void main(String[] args) {
  String n = ”not␣secret”;
  notSecret(n);  // assert that n is not secret
  String s1 = f(n);
  isSecret(s1);
  String s2 = f(secret());
}

static String f(String str) {
  isSecret(str);
  return str;
}

public static String secret() {  // the secret source
  return ”secret”;
}
```

Fig. 5: Example usage of `isSecret` and `notSecret` assertions in regression tests

- classes and interfaces that involve inheritance, overriding, and overloading;
- recursion;
- library calls[9];
- string concatenation and usage of the `StringBuffer` and `StringBuilder` classes[10];
- generics, type conversions through castings, and exception handling.

Our taint analysis implementation becomes unsound in the presence of static initializers. If a static field is initialized to a secret value, our analysis will not detect it as such.

A static initializer is invoked only once, before the instance creation of a class or the access of a static member of that class. Static initializers are invoked lazily by the Java Virtual Machine [11]. This makes finding out at which program point a static initializer is invoked undecidable [6]. To account for static initializers in the analysis would require modifying WALA's control-flow supergraph (which does not have edges to static initializers) or using a data-flow analysis for static initialization. Since the primary purpose of the taint-analysis implementation was to test the correlated-call analysis, we did not include a static-initializer analysis in this work.

[9] We created a specification for library functions that allowed us to indicate under which conditions a library function returned a secret value. This let us avoid the expensive analysis of library functions.

[10] Using mutation, objects of these classes can be converted into wrappers around secret strings. This is why we added a special handling for `StringBuffer` and `StringBuilder` objects. For instance, if a field had the `StringBuilder` type, it was considered secret.

Correlated-Calls-Analysis Correctness We tested the implementation of the correlated-calls analysis by converting the taint analysis into an IDE problem with an implementation of $\mathcal{T}^{\Subset}_{R^{\Subset}}$.

Since none of the test cases in the previous section contained correlated calls, we used the same tests with the same assertions to ensure that the correlated-calls analysis produces the same results as an IFDS-equivalent analysis in the absence of correlated calls.

We then added test cases that contained correlated calls. We added a new assertion method, `notSecretCC`. For the IFDS-equivalent analysis, the method asserted that the argument passed to it was secret, and for the correlated-calls analysis, it asserted that the argument was not secret.

Separately, we used unit tests to check the implementation correctness of micro functions. We wrote assertions for the results of the equality, meet, and composition operations on all possible combinations of the identity, top, bottom, and constant functions.

Benchmark Testing To assess the benefit of the correlated-calls analysis, we counted the frequencies of correlated-call occurrences in the Dacapo benchmarks [2]. We then ran the normal- and correlated-call-taint analysis on the Dacapo benchmarks to see what improvement we would get from the correlated-calls analysis.

We need to decide what exactly we want to include in the evaluation. The numbers of correlated calls are interesting. But I think we want to leave out all mention of the taint analysis. Perhaps we should have no empirical results at all (and only the theory/proofs)? **(O)**

Occurrences of Correlated Calls Our goal was to obtain an upper bound on the number of redundant IFDS-result nodes that could be potentially removed by our analysis. We counted the number of correlated calls that occurred in programs of the Dacapo benchmarks, as shown in Table 1.

In the table, the number of all call sites in a program is denoted as C. Polymorphic call sites are denoted as C_P, and correlated call sites as $C^{\Subset}$. The first four columns indicate the overall number of various call sites and correlated-call receivers in a program. The last three columns indicate the ratio of polymorphic to all call sites, the ratio of correlated to polymorphic call sites, and the ratio of correlated call sites to correlated-call receivers.

We can see that on average, 3% of all call sites C are polymorphic call sites C_P. Out of those call sites, 38% are correlated call sites $C^{\Subset}$. We also see that for one correlated-call receiver, there are on average three correlated calls.

Experiments We ran the analysis on the Dacapo benchmarks to test if the taint analysis would benefit from the improved, correlated-calls based, analysis. We defined any user input string to be considered a secret source and compared the overall number of results in the original and correlated-call taint analyses. If the number of secret values in the original result were larger than in the correlated-call result, we would see a practical benefit from our analysis.

However, even when we considered each program point as a sink, the "improved" analysis revealed the same number of secret values as the original taint analysis.

Table 1: Frequencies of correlated-call occurrences in the Dacapo benchmarks

Benchmark	$\lvert C\rvert$	$\lvert C_P\rvert$	$\lvert C^{\Subset}\rvert$	$\lvert R^{\Subset}\rvert$	$\frac{\lvert C_P\rvert}{\lvert C\rvert}$	$\frac{\lvert C^{\Subset}\rvert}{\lvert C_P\rvert}$	$\frac{\lvert C^{\Subset}\rvert}{\lvert R^{\Subset}\rvert}$
antlr	7,610	428	299	70	**6%**	**70%**	4
bloat	18,157	933	429	119	**5%**	**46%**	4
chart	18,101	466	195	61	**3%**	**42%**	3
eclipse	3,222	100	35	10	**3%**	**35%**	4
fop	4,831	129	40	12	**3%**	**31%**	3
hsqldb	3,573	81	35	10	**2%**	**43%**	4
jython	12,149	487	129	54	**4%**	**26%**	2
luindex	7,190	188	79	29	**3%**	**42%**	3
lusearch	9,043	350	126	47	**4%**	**36%**	3
pmd	10,972	219	68	23	**2%**	**31%**	3
xalan	3,889	110	35	10	**3%**	**32%**	4
Geom. mean	**7,572**	**240**	**91**	**29**	**3%**	**38%**	**3**

A correlated call that could affect a taint-analysis result could most likely occur in the following scenario:

- there is a receiver with at least two polymorphic calls;
- at least one of the calls c_1 returns a string — this would mean that the method potentially returns a secret value;
- at least one of the calls c_2 takes a string parameter — this would mean that a secret value could potentially be propagated to the method as an argument.

Then, if the correlated call occurred on an invocation $c_2(c_1())$, there might be a possibility of benefiting from the correlated-calls analysis. Given the relatively rare occurrence of correlated calls, this situation is not likely to appear often. This is illustrated in Table 2 which shows how often correlated calls would invoke methods that either take a string as a parameter *or* return a string. The set of receivers on which there are invocations of such methods is denoted as $R^{\Subset}{}_{S}$. A situation where one correlated call returned a string, *and* another correlated call on the same receiver took a string parameter, appeared in only one case in the `antlr` benchmark. However, the strings invoked were not designated as secret.

This explains why, specifically for a taint analysis as the client analysis, and specifically for the Dacapo benchmarks, the correlated call analysis did not make a difference.

5.3 Future Work

In this section we point out the limitations of the correlated-calls analysis and suggest improvements to the analysis for future work.

We should convert this section into two or three sentences to be added to the conclusion. They should focus especially on the interprocedurally-correlated receivers. **(O)**

One limitation of the analysis is that it only works for IFDS problems like taint analysis, reachable definitions, or available expressions. The correlated-call

Table 2: Frequency of correlated-call receivers for which at least one of the correlated calls takes a string as a parameter or returns a string

Benchmark	$\lvert R^{\Subset}{}_S\rvert$	$\lvert R^{\Subset}\rvert$	$\frac{\lvert R^{\Subset}{}_S\rvert}{\lvert R^{\Subset}\rvert}$
antlr	43	70	62%
bloat	0	119	0%
chart	1	61	2%
eclipse	0	10	0%
fop	0	12	0%
hsqldb	0	10	0%
jython	6	54	23%
luindex	0	29	0%
lusearch	2	47	6%
pmd	1	23	3%
xalan	0	10	0%
Geom. mean	**3**	**29**	**9**

analysis is not applicable to IDE problems like copy- or linear-constant propagation. Therefore, a possible direction for future work is to create a correlated-calls analysis that transforms an original IDE problem into one that considers correlated calls (with a modified lattice and edge function definition), and then transforms the correlated-calls result into a more precise result of the original IDE problem.

Another constraint of the algorithm is that it only accounts for intra-procedurally-correlated receivers, or receivers on which correlated calls occur within one method. For example, in Listing 6, `a` is a correlated-call receiver, since there are two polymorphic method invocations on `a`. However, the first one, `a.setString()`, is inside method `main`, and the second one, `a.printString()`, is inside method `propagate`. Therefore, we do not treat `a` as a correlated-call receiver, and the analysis would not improve the original IFDS result for this program.

Finally, correlated calls can occur on multiple receivers and other scenarios discussed in [22] that are not handled in this work.

6 Related Work

IFDS is a version of the functional approach to data-flow analysis developed by M. Sharir and A. Pnueli [20]. IFDS has been used to encode a variety of data-flow problems, for example, typestate analysis [13,24] or shape analysis [9]. IFDS has been broadly used [1,23] and extended [10] to solve taint-analysis problems.

IFDS is implemented for two popular static-analysis frameworks for Java bytecode, the T.J. Watson Libraries for Analysis (WALA) [5] and Heros [3].

Work on improving the IFDS algorithm includes the Practical Extensions to the IFDS algorithm [14]. Two of the four extensions improve the efficiency

```
class A {

  String string;

  public static void main(String[] args) {
    A a = args == null ? new A() : new B();
    a.setString();
    propagate(a);
  }

  static void propagate(A a) {
    a.printString();
  }

  void setString() {
    string = secret();
  }

  void printString() {
    System.out.println(”not␣secret”);
  }
}

class B extends A {
  void setString() {
    string = ”not␣secret”;
  }

  void printString() {
    System.out.println(a);
  }
}
```

Fig. 6: Inter-procedurally-correlated calls

of the IFDS analysis for certain classes of IFDS problems. Another extension widens the class of problems applicable for the IFDS analysis. Our analysis, in contrast, does not improve the efficiency or generality of IFDS, but it allows us to solve IFDS problems more precisely. The fourth extension is targeted towards programs that are represented in SSA form. Executing the IFDS analysis on such programs results in loss of precision in the presence of control-flow constructs (e.g. conditionals and loops), compared to programs in non-SSA form. The extension makes the IFDS analysis on programs in SSA form as precise as on programs that are not represented in SSA form. In contrast, the correlated-calls analysis is applicable to programs in both SSA and non-SSA forms. Even if applied to a program in SSA form, our analysis and the extension improve the precision of IFDS in unrelated situations: the first analysis handles correlated calls, and the latter handles control-flow constructs. Thus, an IFDS analysis could benefit from both precision improvements independently.

Another work on improving the efficiency of the IFDS algorithm is E. Bodden et al.'s framework for the analysis of software products lines [4]. Their paper uses transformations from IFDS to IDE problems, a technique we also employ. Finally, J. Rodriguez and O. Lhoták implemented a concurrent version of the IFDS algorithm using actors [18]. However, neither of those works is concerned with improving the precision of IFDS results.

The idea of using correlated calls to remove infeasible paths in data-flow analyses of object-oriented programs was introduced by F. Tip [22]. The possibility of using IDE to achieve this is mentioned, but not elaborated upon. Our work presents a concrete solution to the problem and an implementation of that solution.

The idea of eliminating infeasible paths caused by correlated calls is similar to M. Sridharan et al.'s work on improving the precision of pointer analysis for JavaScript programs [21]. For each pointer, a pointer analysis determines the possible set of objects (the *points-to* set) that the pointer can reference at a given program point. In JavaScript, it is challenging to compute the points-to set of fields because in general, field names can be derived from arbitrary expressions and bound at runtime. As a result, an imprecise data-flow analysis will include infeasible paths between values of the form `o[p]` (access of a property `p` of object `o`), where at compile time, `p` can be bound to different values. The idea of the paper is to track all dynamic property accesses (reads and writes) on an object `o` with property name `p`. The code snippets containing the references `o[p]` are then extracted into a separate function f. The analysis is then run so that for each possible value of `p`, f is analyzed separately; therefore, for a given property name, all correlated objects with that name are analyzed together.

The differences between this method of tracking correlated calls and our analysis are the following.

- *Type of target data-flow analysis* whose precision is to be improved. Our analysis improves the precision of IFDS data-flow analyses, whereas the JavaScript analysis improves the precision of pointer analysis.

- *Target language.* Our analysis is for object-oriented languages where polymorphic methods, and not property names (which are known at compile time), cause infeasible paths.
- *Different handling of correlated calls.* Extracting code that contains correlated calls into separate methods would not prevent infeasible paths. Instead, our analysis uses IDE flow functions to detect and eliminate infeasible paths caused by correlated calls.

7 Conclusions

We presented a technique to improve the precision of solutions to IFDS problems in the presence of correlated calls. Correlated calls occur when there are multiple polymorphic method invocations on the same receiver. Such method calls cause a data-flow analysis to consider infeasible paths, which makes the data-flow analysis less precise.

Our method of eliminating infeasible paths caused by correlated calls works by transforming an existing IFDS problem into a specialized IDE problem. In this way, we are able to track the classes to which method invocations get dispatched. After solving the specialized IDE problem, we convert its result into an IFDS result that is potentially more precise than the solution to the original IFDS problem. The increase in precision can occur for programs that contain correlated calls. Specifically, if, on a certain data-flow path, there are two polymorphic method invocations on the same receiver that dispatch to incompatible classes, the IDE analysis will consider the path infeasible.

We proved that the correlated-calls analysis is sound and that it improves the precision of IFDS results.

Our Scala implementation of the correlated-calls analysis includes

- an implementation of the IDE analysis, which is based on the WALA static program analysis framework;
- a taint-analysis implementation as an IFDS problem instance;
- a transformer of IFDS problems to equivalent IDE problems, and a second transformer that accounts for correlated calls.

We tested the correlated-calls analysis on our taint analysis implementation by comparing the number of secret values that were leaked when using an IFDS taint analysis and a taint analysis that accounts for correlated calls. We used the Dacapo benchmarks as input programs. Although the benchmarks contained a number of correlated calls, we were not able to improve the precision of the taint analysis, because the correlated calls did not occur on paths of secret information leaks.

We are hopeful that other analyses can benefit from the extra information provided by the correlated-calls analysis, and plan to test this hypothesis in the future.

References

1. 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 29, 2014.
2. Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006*, pages 169–190, 2006.
3. 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 2012, June 14, 2012*, pages 3–8, 2012.
4. Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. SPLLIFT - statically analyzing software product lines in minutes instead of years. In *Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, 25. Februar - 28. Februar 2014*, pages 81–82, 2014.
5. Stephen Fink and Julian Dolby. WALA — the TJ Watson libraries for analysis. `http://wala.sourceforge.net`, 2012.
6. Laurent Hubert and David Pichardie. Soundly handling static fields: Issues, semantics and analysis. *Electr. Notes Theor. Comput. Sci.*, (5):15–30, 2009.
7. Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In *Compiler Construction, 4th International Conference on Compiler Construction, CC'92, October 5-7, 1992, Proceedings*, pages 125–140, 1992.
8. Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst.*, (3):268–299, 1996.
9. Jörg Kreiker, Thomas W. Reps, Noam Rinetzky, Mooly Sagiv, Reinhard Wilhelm, and Eran Yahav. Interprocedural shape analysis for effectively cutpoint-free programs. In *Programming Logics — Essays in Memory of Harald Ganzinger*, pages 414–445, 2013.
10. Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 98–108, 2014.
11. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* 1997.
12. Markus Müller-Olm and Oliver Rüthing. On the complexity of constant propagation. In *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, April 2-6, 2001, Proceedings*, pages 190–205, 2001.
13. Nomair A. Naeem and Ondřej Lhoták. Typestate-like analysis of multiple interacting objects. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on*

Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, pages 347–366, 2008.

14. Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the IFDS algorithm. In *Compiler Construction, 19th International Conference, CC 2010, March 20-28, 2010. Proceedings*, pages 124–144, 2010.
15. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis (2. corr. print)*. 2005.
16. Martin Odersky. Essentials of Scala. In *Langages et Modèles à Objets, LMO 2009, 25-27 mars 2009*, page 2, 2009.
17. Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 23-25, 1995*, pages 49–61, 1995.
18. Jonathan David Rodriguez. A concurrent IFDS dataflow analysis algorithm using actors. Master's thesis, 2010.
19. Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, May 22-26, 1995, Proceedings*, pages 651–665, 1995.
20. Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. *Program flow analysis: Theory and applications*, pages 189–234, 1981.
21. Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, June 11-16, 2012. Proceedings*, pages 435–458, 2012.
22. Frank Tip. Infeasible paths in object-oriented programs. *Sci. Comput. Program.*, 97:91–97, 2015.
23. Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, June 15-21, 2009*, pages 87–97, 2009.
24. Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in Datalog. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 27, 2014.

Appendix

In this appendix we present the proofs to the Lemmas introduced in Section **??**.

Soundness and Precision

We start by proving the Lemmas of Soundness and Precision of the correlated-calls analysis.

Proof of Lemma 4. The transformation $\mathcal{U}^{\Subset}$ is the same as $\mathcal{U}^{\equiv}$, except that it can remove data-flow facts from the result:

$$\begin{aligned}\mathcal{U}^{\Subset}\left(\mathcal{R}(\mathcal{T}_R^{\Subset}(P))\right)(n) &= \{(n',\, D_n'^{\Subset}(\mathcal{R}(\mathcal{T}_R^{\Subset}(P)))) \mid n \in N^*\}(n)\\ &= D_n^{\Subset}(\mathcal{R}(\mathcal{T}_R^{\Subset}(P)))\\ &\subseteq \mathsf{MVP}_F(n)\\ &= \mathcal{R}_{\text{IFDS}}(P)(n)\,. \qquad \square\end{aligned}$$

To prove the Soundness Lemma, we first introduce Lemmas 7 and 8.

We will denote the top element in the environment lattice as

$$\mathit{\Omega} = \lambda d\,.\,\top_{\Subset}\,. \tag{8}$$

For the purpose of the proofs, we will rewrite Equation (3) that defines an edge function as follows:

$$\mathsf{EdgeFn}_S^{\Subset} = \lambda e\,.\begin{cases}\mathsf{id} & \text{if } d_1 = d_2 = \varLambda,\\ \lambda m\,.\,\varepsilon(e)(\delta(m)) & \text{otherwise,}\end{cases} \tag{9}$$

where $S \subseteq R$, d_1 and d_2 are the source and target facts, and for a map $m \in L_U^{\Subset}$, $\delta(m)$ is either m or $\bot_{\Subset}$:

$$\delta(m) = \begin{cases}\bot_{\Subset} & \text{if } d_1 = \varLambda\\ m & \text{otherwise.}\end{cases} \tag{10}$$

Additionally, for a path $p = [\mathsf{start}_{\mathtt{main}}, \ldots]$ and a fact $d \in D$, we will denote the lattice element that is mapped to d according to the flow functions of path p as follows:

$$\xi(p,\, d) = M_{\mathsf{Env}}(p)(\mathit{\Omega})(d)\,. \tag{11}$$

The following Lemma shows that the lattice elements (receiver-to-types maps) of a correlated-calls IDE analysis correctly overapproximate the possible types of a receiver in a program execution.

Lemma 7. *Let $p = [\mathsf{start}_{main}, \ldots, n]$ be some concrete execution trace of the program, and let $r \in R$ be a receiver. If after the execution trace p, at node n, r points to an object of runtime type t, and $d \in D$ is a fact such that $d \in M_F(p)(\varnothing)$, then*

$$t \in \xi(p,\, d)(r)\,. \tag{12}$$

Proof. By induction on the length of the trace.

Basis: $p = [\mathsf{start}_{\mathtt{main}}]$. Then there is no instruction at which a receiver r could be instantiated, and the Lemma is trivially true.

Induction hypothesis: Let $p = [\mathsf{start}_{\mathtt{main}}, \ldots, n_{k-1}]$, and let τ be the set of types to which $\xi(p,\, d_{k-1})$ maps r:

$$\tau = \xi(p,\, d_{k-1})(r)\,. \tag{13}$$

Assume that for a concrete execution path $p = [\mathsf{start}_{\mathtt{main}}, \ldots, n_{k-1}]$, at node (n_{k-1}, d_{k-1}), the Lemma holds, i.e. $t \in \boldsymbol{\tau}$.

Induction step: Let $p' = [\mathsf{start}_{\mathtt{main}}, \ldots, n_{k-1}, n_k]$ and $t' \in T$ be the type to which r is mapped at n_k.

For each i, let e_i be the edge $((n_{i-1}, d_{i-1}), (n_i, d_i))$. Note that

$$e_1 = ((\mathsf{start}_{\mathtt{main}}, \Lambda), (n_1, d_1)) \,.$$

Observe that

$$\begin{aligned}\xi(p', d) &= M_{\mathsf{Env}}(p')(\Omega)(d)\\ &= (M_{\mathsf{Env}}(e_k) \circ M_{\mathsf{Env}}(e_{k-1}) \circ \ldots \circ M_{\mathsf{Env}}(e_1))\,(\Omega)(d)\\ &= M_{\mathsf{Env}}(e_k)\,(M_{\mathsf{Env}}(p)(\Omega))\,(d)\,.\end{aligned}$$

According to (**??**),

$$\begin{aligned}&M_{\mathsf{Env}}(e_k)\,(M_{\mathsf{Env}}(p)(\Omega))\,(d)(r)\\ =&\Bigg(\mathsf{EdgeFn}_R^{\Subset}((n_{k-1}, \Lambda), (n_k, d))(\top_{\Subset}) \sqcap\\ &\bigsqcap_{d' \in D} \mathsf{EdgeFn}_R^{\Subset}((n_{k-1}, d'), (n_k, d))(M_{\mathsf{Env}}(p)(\Omega)(d'))\Bigg)(r)\\ \supseteq&\bigsqcap_{d' \in D} \mathsf{EdgeFn}_R^{\Subset}((n_{k-1}, d'), (n_k, d))(M_{\mathsf{Env}}(p)(\Omega)(d'))(r)\\ \supseteq&\,\mathsf{EdgeFn}_R^{\Subset}((n_{k-1}, d_{k-1}), (n_k, d))(\xi(p, d_{k-1}))(r)\,.\end{aligned}$$

Therefore,

$$\mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p, d_{k-1}))(r) \subseteq \xi(p', d)(r)\,. \tag{14}$$

We will now show that

$$t' \in \mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p, d_{k-1}))(r)\,,$$

which, due to (14), means that the Lemma holds.

According to (9), there are two cases in which $\mathsf{EdgeFn}_R^{\Subset}(e_k)$ could fall.

If $d_{k-1} = d_k = \Lambda$, then $d_k \notin M_F(p)(\varnothing)$, since it does not belong to the set D, and the Lemma trivially holds.

Otherwise,

$$\mathsf{EdgeFn}_R^{\Subset}(e_k) = \lambda m\,.\varepsilon(e_k)(\delta(m))\,.$$

It follows that

$$\begin{aligned}\mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p, d_{k-1}))(r) &= (\lambda m\,.\varepsilon(e_k)(\delta(m)))(\xi(p, d_{k-1}))(r)\\ &= \varepsilon(e_k)(\delta(\xi(p, d_{k-1})))(r).\end{aligned} \tag{15}$$

Let us denote the lattice element $\delta(\xi(p, d_{k-1}))$ with Δ:

$$\Delta = \delta(\xi(p, d_{k-1}))\,.$$

Note that since Δ, according to (10), can be either $\bot_{\Subset}$ or $\xi(p,\, d_{k-1})$, it always maps r to a set containing t:

$$t \in \Delta(r)\,. \tag{16}$$

Note also that unless the instruction at n_{k-1} contains an assignment for r, r is mapped to the same object of type t as at node n_{k-1}, and $t = t'$. Therefore, for the non-assignment instructions, it is sufficient to prove that $t \in \Delta(r)$.

Depending on the instructions at the nodes n_{k-1} and n_k, there are four cases:

1. The instruction at n_{k-1} is an assignment for a receiver $r' \in R$. Since $\varepsilon_R(e_k) = \lambda m\,.\, m[r' \to \bot_T]$,

$$\begin{aligned}\mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r) &= (\lambda m\,.\, m[r' \to \bot_T])(\Delta)(r)\\ &= \Delta[r' \to \bot_T](r)\,.\end{aligned}$$

   In the resulting map, r' is mapped to $\bot_T$. Then

   (a) if $r = r'$, then $\mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r) = \bot_T$, which contains t'.

   (b) If $r \neq r'$, then r has not been reassigned a value, and still maps to the same object of type t. The receiver r is mapped to $\Delta(r)$, which, according to (16), contains t. Since $t = t'$, $\Delta(r)$ contains t'.

2. e_k is a call-start edge with signature $s_{\mathcal{F}}$, and $f \in \mathcal{F}$ is the called procedure. Then

$$\begin{aligned}\mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r) &= (\lambda m\,.\, m[r' \to m(r') \cap \tau(s_{\mathcal{F}},\, f)])(\Delta)(r)\\ &= \Delta[r' \to \Delta(r') \cap \tau(s_{\mathcal{F}},\, f)]\,,\end{aligned}$$

   where r' is the receiver of the call.

   - If $r' = r$, then $\Delta(r') = \Delta(r)$ which contains t. Since $t \in \tau(s_{\mathcal{F}},\, f)$, it follows that $t \in \Delta(r) \cap \tau(s_{\mathcal{F}},\, f)$, and $t \in \mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r)$.
   - If $r' \neq r$, see (1b).

3. e_k is an end-return edge, $r_1,\, \ldots,\, r_k \in R$ are the local variables in the callee method, r' is the receiver of the call site corresponding to the return node n_k, and $f \in \mathcal{F}$ is the called method with signature $s_{\mathcal{F}}$. Then

$$\varepsilon_R(e_k) = \lambda m\,.\, m[r' \to m(r') \cap \tau(s_{\mathcal{F}},\, f)][r_1 \to \bot_T]\ldots[r_k \to \bot_T].$$

   If $r \in \{r_1,\, \ldots,\, r_k\}$, see Case 1. Otherwise, the case is analogous to Case 2.

4. The node contains any other instruction. Then

$$\mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r) = \mathsf{id}(\Delta)(r) = \Delta(r),$$

   which contains t according to (16). □

We will now show that on a node of a concrete execution path, the correlated-calls analysis does not map receivers to $\top_T$. In other words, the analysis never considers nodes of a concrete execution path unreachable.

Lemma 8. *Let $p = [\mathsf{start}_{\mathit{main}},\, \dots,\, n]$ be a concrete execution path, $r \in R$ a receiver, and $d \in D$ a data-flow fact. Then if $d \in M_F(p)(\varnothing)$,*

$$\xi(p,\, d)(r) \neq \top_T\,. \tag{17}$$

Proof. By induction on the length of the execution trace.

Basis: Let $p = [\mathsf{start}_{\mathtt{main}}]$. Since the only realizable path corresponding to p is $[(\mathsf{start}_{\mathtt{main}},\, \Lambda)]$, there is no fact $d \in D$ such that $d \in M_F(p)(\varnothing)$, and the claim follows immediately.

Induction hypothesis: Let $p = [\mathsf{start}_{\mathtt{main}},\, \dots,\, n_{k-1}]$. Let τ be the set of types to which r is mapped by $\xi(p,\, d_{k-1})$:

$$\tau = \xi(p,\, d_{k-1})(r)\,. \tag{18}$$

Assume the Lemma holds for that for a concrete execution path

$$p = [\mathsf{start}_{\mathtt{main}},\, n_1,\, \dots,\, n_{k-1}]\,,$$

i.e. $\tau \neq \top_T$ for an arbitrary $r \in R$ and $d_{k-1} \in D$.

Induction step: Let $p' = [\mathsf{start}_{\mathtt{main}},\, n_1,\, \dots, n_{k-1},\, n_k]$ be a concrete execution path.

Let $e_k = ((n_{k-1},\, d_{k-1}),\, (n_k,\, d))$. As shown in (14),

$$\xi(p',\, d)(r) \supseteq \mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r)\,.$$

From Definition 2, we can see that unless e_k is a call-start edge or an end-return edge, the result follows from the induction hypothesis. More formally, if e_k is not a call-start or end-return edge, then for all $m \in L_R^{\Subset}$,

$$\mathsf{EdgeFn}_R^{\Subset}(e_k)(m) \sqsubseteq m\,.$$

The edge function corresponding to the call-start and end-return edges is the only place in which the set of types that a receiver maps to can be reduced.

Assume that e_k is a end-return edge with a call on the receiver $r' \in R$ with a signature $s_{\mathcal{F}}$ to a function $f \in \mathcal{F}$.

$$\begin{aligned}
&\mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r)\\
&\qquad = (\lambda m\,.\, m[r' \to m(r) \cap \tau(s_{\mathcal{F}},\, f)][r_1 \to \bot_T] \dots [r_l \to \bot_T])\, (\xi(p,\, d_{k-1}))(r)\\
&\qquad = (\xi(p,\, d_{k-1})[r' \to \tau \cap \tau(s_{\mathcal{F}},\, f)][r_1 \to \bot_T] \dots [r_l \to \bot_T])\, (r)\,,
\end{aligned}$$

where $r_1,\, \dots, r_l \in R$ are the local variables in the called method.

If $r \in \{r_1,\, \dots,\, r_l\}$, then $\mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r) = \bot_T \ni t$[11].

Otherwise, if $r = r'$, then $\mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r) = \tau \cap \tau(s_{\mathcal{F}},\, f)$.

According to Lemma 7 and by the induction hypothesis, the runtime type t of r must be contained in $\xi(p,\, d_{k-1})(r) = \tau$. At the same time, by definition, t is part of $\tau(s_{\mathcal{F}},\, f)$. Therefore, $t \in \tau \cap \tau(s_{\mathcal{F}},\, f) \subseteq \mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r)$, which means that $\mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r) \neq \top_T$.

The same reasoning applies to the case where e_k is a call-start edge. □

[11] In the case of a recursive call, it is possible that both $r \in \{r_1,\, \dots,\, r_l\}$ and $r = r'$. In that case, the set to which r will be mapped would be still "overwritten" by $\bot_T$.

Finally, we can prove the Soundness Lemma.

Proof of Lemma 5. Let $\rho = \mathcal{R}(\mathcal{T}_R^{\Subset}(P))$. Then

$$\begin{aligned}\mathcal{U}^{\Subset}(\rho)(n) &= D_n^{\Subset}(\rho)\\ &= \{d' \mid d' \in \mathsf{MVP}_F(n) \wedge \forall r \in R :\ \rho(n,\ d')(r) \neq \top_T\}\,.\end{aligned}$$

Since $\mathsf{MVP}_F(n) = \bigsqcap_{q \in \mathsf{VP}(n)} M_F(q)(\varnothing)$, and $p \in \mathsf{VP}(n)$, it follows that

$$\begin{aligned}d &\in M_F(p)(\varnothing)\\ &\subseteq \mathsf{MVP}_F(n)\,.\end{aligned}$$

At the same time, for all receivers $r \in R$,

$$\begin{aligned}\rho(n,\ d)(r) &= \left(\bigsqcap_{q \in \mathsf{VP}(n)} \xi(q,\ d)\right)(r)\\ &= \bigsqcap_{q \in \mathsf{VP}(n)} \xi(q,\ d)(r)\,.\end{aligned}$$

According to Lemma 8, $\xi(p,\ d)(r) \neq \top_T$. Since $p \in \mathsf{VP}(n)$,

$$\xi(p,\ d)(r) \subseteq \bigsqcap_{q \in \mathsf{VP}(n)} \xi(q,\ d)(r)\,.$$

From $\bigsqcap_{q \in \mathsf{VP}(n)} \xi(q,\ d)(r) = \rho(n,\ d)(r)$ it follows that $\xi(p,\ d)(r) \subseteq \rho(n,\ d)(r)$. Therefore, $\rho(n,\ d)(r) \neq \top_T$, and $d \in D_n^{\Subset}(\rho) = \mathcal{U}^{\Subset}(\rho)(n)$. □

Correlated Call Receivers

This appendix contains the proof for Lemma **??** which shows that in a correlated-calls analysis, it is enough to consider only correlated-call receivers $R^{\Subset}$.

The following Lemma shows that the types to which a given receiver is mapped in the result of the algorithm is not affected by other receivers and the types to which they are mapped.

Lemma 9. *Let P be an IFDS problem. Let N^* be the supergraph for P, D the set of data-flow facts, $n \in N^*$ a node, and $p = [\textit{start}_{\textit{main}},\ \ldots,\ n]$ a path in the supergraph. Let $d \in D \cup \{\Lambda\}$. Then for any realizable path $p' \in \textit{RP}(p,\ d)$, set $S \subseteq R$, and receiver $r \in S$,*

$$\textit{EdgeFn}_S^{\Subset}(p')(\top_{\Subset})(r) = \textit{EdgeFn}_{\{r\}}^{\Subset}(p')(\top_{\Subset})(r)\,. \tag{19}$$

Proof. By induction on the length of p.

Basis: $p' = [(\mathsf{start}_{\mathtt{main}},\ \Lambda)]$. Then $\mathsf{EdgeFn}_S^{\Subset}(p') = \mathsf{id} = \mathsf{EdgeFn}_{\{r\}}^{\Subset}(p')$, and the Lemma follows directly.

Induction hypothesis: Suppose that for a path $q = [(\mathsf{start}_{\mathtt{main}}, \Lambda), \ldots, (n_{k-1}, d_{k-1})]$, where $q \in \mathsf{RP}(n, d)$, the Lemma holds, i.e. both edge functions map r to the same set of types τ:

$$\begin{aligned}\tau &= \mathsf{EdgeFn}^{\Subset}_{S}(q)(\top_{\Subset})(r)\\ &= \mathsf{EdgeFn}^{\Subset}_{\{r\}}(q)(\top_{\Subset})(r)\,.\end{aligned}$$

Induction step: Let $q' = [(\mathsf{start}_{\mathtt{main}}, \Lambda), \ldots, (n_{k-1}, d_{k-1}), (n_k, d_k)]$ and the edge $e_k = ((n_{k-1}, d_{k-1}), (n_k, d_k))$.

Observe that for any set $U \subseteq R$ such that $r \in U$,

$$\mathsf{EdgeFn}^{\Subset}_{U}(q')(\top_{\Subset})(r) = \mathsf{EdgeFn}^{\Subset}_{U}(e_k)(\mathsf{EdgeFn}^{\Subset}_{U}(q)(\top_{\Subset}))(r)\,. \tag{20}$$

We can see from (9) that there are two cases.

If $d_{k-1} = d_k = \Lambda$, $\mathsf{EdgeFn}^{\Subset}_{S}(e_k) = \mathsf{id} = \mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_k)$, and, due to (20),

$$\begin{aligned}\mathsf{EdgeFn}^{\Subset}_{S}(q')(\top_{\Subset})(r) &= \tau\\ &= \mathsf{EdgeFn}^{\Subset}_{\{r\}}(q')(\top_{\Subset})(r)\,.\end{aligned}$$

Otherwise, there are four sub-cases.

1. e_k is a call-start edge, $r'.c()$ is the call site at n_{k-1} with signature $s_{\mathcal{F}}$, $f \in \mathcal{F}$ is the called procedure, and $r' \in U$. Then

$$\mathsf{EdgeFn}^{\Subset}_{U}(e_k) = \lambda m\,.\,\delta(m)[r' \to \delta(m)(r) \cap \tau(s_{\mathcal{F}}, f)]\,.$$

There are two sub-cases.

(a) If $r = r'$, then, according to (20), the resulting set of types

$$\mathsf{EdgeFn}^{\Subset}_{U}(q')(\top_{\Subset})(r) = \delta(\mathsf{EdgeFn}^{\Subset}_{U}(q)(\top_{\Subset}))(r) \cap \tau(s_{\mathcal{F}}, f).$$

If $d_{k-1} = \Lambda$, then $\delta(\mathsf{EdgeFn}^{\Subset}_{U}(q)(\top_{\Subset}))(r) = \bot_{\Subset}(r) = \bot_T$. If $d_{k-1} \neq \Lambda$, then $\delta(\mathsf{EdgeFn}^{\Subset}_{U}(q)(\top_{\Subset}))(r) = \mathsf{EdgeFn}^{\Subset}_{U}(q)(\top_{\Subset})(r) = \tau$. The set $\tau(s_{\mathcal{F}}, f)$ is the same for either case.

Therefore, the value of $\mathsf{EdgeFn}^{\Subset}_{U}(q')(\top_{\Subset})(r)$ has the same result regardless of U, which means that $\mathsf{EdgeFn}^{\Subset}_{S}(q')(\top_{\Subset})(r) = \mathsf{EdgeFn}^{\Subset}_{\{r\}}(q')(\top_{\Subset})(r)$, and the Lemma holds.

(b) If $r \neq r'$, then

$$\mathsf{EdgeFn}^{\Subset}_{U}(q')(\top_{\Subset})(r) = \delta(\mathsf{EdgeFn}^{\Subset}_{U}(q)(\top_{\Subset}))(r)\,, \tag{21}$$

which, as we have seen in Case (1a), does not depend on U, and the Lemma holds.

2. e_k is an end-return edge, $r_1, \ldots, r_l \in U$ are the local variables in the callee method, $r'.c()$ is the call corresponding to the return node at n_k, $f \in \mathcal{F}$ is the called method with signature $s_{\mathcal{F}}$, and $r' \in U$. Then

$$\mathsf{EdgeFn}^{\Subset}_{U}(e_k) = \lambda m\,.\,\delta(m)[r' \to \delta(m)(r) \cap \tau(s_{\mathcal{F}}, f)][r_1 \to \bot_T]\ldots[r_l \to \bot_T]\,.$$

There are three sub-cases.

(a) If $r \in \{r_1, \ldots, r_l\}$, then regardless of the value of U,

$$\mathsf{EdgeFn}^{\Subset}_{U}(q')(\top_{\Subset})(r) = \bot_T\,,$$

and the Lemma holds.

(b) Otherwise, if $r = r'$, the case is analogous to Case (1a).

(c) If $r \notin \{r', r_1, \ldots, r_l\}$, then see Case (1b).

3. n_{k-1} contains an assignment for $r' \in U$. Then

$$\mathsf{EdgeFn}^{\Subset}_{U}(e_k) = \lambda m\,.\,\delta(m)[r' \to \bot_T]\,.$$

If $r = r'$, see Case (2a). If $r \neq r'$, see Case (1b).

4. Otherwise,

$$\mathsf{EdgeFn}^{\Subset}_{U}(e_k) = \lambda m\,.\,\delta(m)\,,$$

and the case is analogous to Case (1b). □

The following Lemma shows that the correlated-calls analysis computes the results for each receiver independently, or separately. To compute the set of types to which a receiver r is mapped at each exploded-graph node, we can exclude all other receivers in the program from the analysis (recall from (3) that the set of receivers that are considered in the analysis is specified by the set S in a correlated-calls transformation $\mathcal{T}^{\Subset}_{S}$). Therefore, for a given receiver r, the results of a $\mathcal{T}^{\Subset}_{S}$- and a $\mathcal{T}^{\Subset}_{\{r\}}$-analysis are the same.

Lemma 10. *Let P be an IFDS problem. Let N^* be the supergraph for P, D the set of data-flow facts, and $S \subseteq R$ a set of receivers. Then for any $n \in N^*$, $d \in D$, and receiver $r \in S$,*

$$\mathcal{R}\left(\mathcal{T}^{\Subset}_{S}(P)\right)(n,\, d)(r) = \mathcal{R}(\mathcal{T}^{\Subset}_{\{r\}}(P))(n,\, d)(r)\,. \tag{22}$$

Proof. According to (**??**), (**??**), and (**??**),

$$\begin{aligned}
\mathcal{R}\left(\mathcal{T}^{\Subset}_{S}(P)\right)(n,\, d)(r) &= \mathsf{MVP}_{\mathsf{Env}}(n,\, d)(r) \\
&= \left(\bigsqcap_{q \in \mathsf{VP}(n)} M_{\mathsf{Env}}(q)(\varOmega)(d)\right)(r) \\
&= \left(\bigsqcap_{q \in \mathsf{VP}(n)}\ \bigsqcap_{q' \in \mathsf{RP}(q,\, d)} \mathsf{EdgeFn}^{\Subset}_{S}(q')(\top_{\Subset})\right)(r) \\
&= \bigcup_{q \in \mathsf{VP}(n)}\ \bigcup_{q' \in \mathsf{RP}(q,\, d)} \mathsf{EdgeFn}^{\Subset}_{S}(q')(\top_{\Subset})(r)\,.
\end{aligned} \tag{23}$$

Then from Lemma 9,

$$\begin{aligned}
\mathcal{R}\left(\mathcal{T}^{\Subset}_{S}(P)\right)(n,\, d)(r) &= \bigcup_{q \in \mathsf{VP}(n)}\ \bigcup_{q' \in \mathsf{RP}(q,\, d)} \mathsf{EdgeFn}^{\Subset}_{\{r\}}(q')(\top_{\Subset})(r) \\
&= \mathcal{R}\left(\mathcal{T}^{\Subset}_{\{r\}}(P)\right)(n,\, d)(r)\,.
\end{aligned}$$

□

The next lemma shows that the set of types to which a receiver is mapped in a correlated-calls lattice element can be represented as an intersection of static-type function applications $\tau(s_{\mathcal{F}_i},\, f_i)$.

Lemma 11. *For an IFDS problem P, a node $n \in N^*$, and fact $d \in D$, let $p \in \mathsf{RP}(n,\, d)$ be a realizable path and $r \in R$ a receiver. Then there exists a non-negative number γ of calls on the receiver r with signatures $s_{\mathcal{F}_\gamma}$ to the functions $f_\gamma \in \mathcal{F}_\gamma$, for which*

$$\mathit{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})(r) = \bigcap_{\gamma \geq 0} \tau(s_{\mathcal{F}_\gamma},\, f_\gamma)\,.$$

Proof. Let p have the following form[12]:

$$p = [(\mathsf{start}_{\mathtt{main}},\, \Lambda),\, (n_1,\, \Lambda),\, \ldots,\, (n_k,\, \Lambda), (n_{k+1},\, d_{k+1}),\, \ldots,\, (n_{k+l},\, d_{k+l})]\,,$$

where $l \geq 1$ and the facts for all nodes up to n_k are equal to Λ and $d_{k+i} \in D$ for $0 < i \leq l$.

As previously, for all i, we will denote the edge $(n_i,\, n_{i+1})$ by e_i.

From (3) we can infer that

$$\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p) = \mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l}) \circ \ldots \circ \mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+2}) \circ (\lambda m\,.\,\beta) \circ \mathsf{id} \circ \ldots \circ \mathsf{id}\,,$$

where

$$\beta = \begin{cases} \bot_{\Subset}[r \to \tau(s_{\mathcal{F}},\, f)] & \text{if } (n_k,\, n_{k+1}) \text{ is a call-start or end-return edge, and} \\ & \text{the call site } r.c() \text{ with signature } s_{\mathcal{F}} \text{ to the function} \\ & f \in \mathcal{F} \text{ corresponds to the call-start or end-return edge,} \\ \bot_{\Subset} & \text{otherwise}^{13}. \end{cases}$$

Therefore,

$$\begin{aligned} \mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset}) &= \Big(\mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l}) \circ \ldots \circ \mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+2})\Big)\,((\lambda m\,.\,\beta)(\top_{\Subset})) \\ &= \Big(\mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l}) \circ \ldots \circ \mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+2}) \circ \mathsf{id}\Big)\,(\beta)\,. \qquad (24) \end{aligned}$$

We can now prove the lemma by induction on l.

Basis: If $l = 1$, then $\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset}) = \mathsf{id}(\beta) = \beta$. There are two cases.

[12] It can be shown from the definition of a pointwise representation in Sagiv et al. [19] that in a realizable path, there is never an edge from a fact of the set D to a Λ fact. Therefore, we can represent p as a sequence of nodes that has a prefix of Λ-fact nodes, after which all nodes are non-Λ facts.

[13] Since $d_k = \Lambda$ and $d_{k+1} \neq \Lambda$, the micro function for the edge e_{k+1} is equal to $\lambda m\,.\,\varepsilon_{\{r\}}(e_{k+1})(\bot_{\Subset})$. From the definition of ε_S (2) we can see that the only case where $\varepsilon_{\{r\}}(e_{k+1})(m)$ would not be equal to $\bot_{\Subset}$ is when e_{k+1} is call-start or end-return edge.

If $\beta = \bot_{\Subset}$, then

$$\begin{aligned}\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})(r) &= \beta(r)\\ &= \bot_T\,,\end{aligned}$$

and $\gamma = 0$.

If $\beta = \bot_{\Subset}[r \to \tau(s_{\mathcal{F}},\, f)]$, then

$$\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})(r) = \tau(s_{\mathcal{F}},\, f)\,,$$

and $\gamma = 1$.

Induction hypothesis: Assume that for a path $p = [(\mathsf{start}_{\mathtt{main}},\, \Lambda),\, \ldots,\, (n_{k+l},\, d_{k+l})]$, the Lemma holds for $\gamma = N$, where $N \geq 0$.

Induction step: Let $p' = [(\mathsf{start}_{\mathtt{main}},\, \Lambda),\, \ldots,\, (n_{k+l},\, d_{k+l}),\, (n_{k+l+1},\, d_{k+l+1})]$.

Recall that

$$\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p')(\top_{\Subset})(r) = \mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l+1})\left(\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})\right)(r)\,.$$

From (2) we can see that unless e_{k+l+1} is a call-start or end-return edge corresponding to a call on the receiver r, then $\mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l+1})(r)$ must be equal to either $\bot_T$ or $m(r)$, where $m = \mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})$.

If $\mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l+1})(r) = \bot_T$, then the Lemma holds for $\gamma = 0$.

Otherwise,

$$\begin{aligned}\mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l+1})(\top_{\Subset})(r) &= \mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})(r)\\ &= \bigcap_N \tau(s_{\mathcal{F}_N},\, f_N)\,,\end{aligned}$$

and therefore $\gamma = N$.

Suppose that e_{k+l+1} is a call-start edge with a call on the receiver r with signature $s_{\mathcal{G}}$ to a function $g \in \mathcal{G}$. Then, according to (2),

$$\mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l+1}) = \lambda m\,.\, m[r \to m(r) \cap \tau(s_{\mathcal{G}},\, g)]\,.$$

Therefore,

$$\begin{aligned}&\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p')(\top_{\Subset})(r)\\ &\qquad= \lambda m\,.\, m[r \to m(r) \cap \tau(s_{\mathcal{G}},\, g)]\left(\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})\right)(r)\\ &\qquad= \mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})(r) \cap \tau(s_{\mathcal{G}},\, g)\\ &\qquad= \left(\bigcap_N \tau(s_{\mathcal{F}_N},\, f_N)\right) \cap \tau(s_{\mathcal{G}},\, g)\,,\end{aligned}$$

and the Lemma holds for $\gamma = N + 1$.

The case where e_{k+l+1} is an end-return edge is analogous to the previous case. □

We now show that a receiver will be only mapped to $\top_{\Subset}$ if it is the receiver of a correlated call.

Lemma 12. *For an IFDS problem P, let $n \in N^*$ be a node, and $d \in D$ a dataflow fact such that there exists a realizable path $p \in \mathit{RP}(n, d)$. Let T be the set of all types in the program. If there exists a receiver $r \in R$ such that*

$$\mathsf{\mathit{EdgeFn}}^{\Subset}_{\{r\}}(p)(\top_{\Subset})(r) = \top_T \,,$$

then $r \in R^{\Subset}$.

Proof. According to Lemma 11,

$$\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})(r) = \bigcap_{\gamma \geq 0} \tau(s_{\mathcal{F}_\gamma},\, f_\gamma).$$

Let $\tau_i = \tau(s_{\mathcal{F}_i},\, f_i)$. For a given k, let $r.m_k()$ be the call site corresponding to τ_k, and T' the set of types compatible with the static type of r. Recall from Section **??** that

- $\tau_k \neq \top_T$;
- if $\tau_k = T'$ then the corresponding call site is monomorphic;
- if $\tau_k \subset T'$ then the call site is polymorphic.

From the conditions of the Lemma,

$$\bigcap_{\gamma \geq 0} \tau_\gamma = \top_T \,. \tag{25}$$

If all $\tau_k = T'$, then $\bigcap_{\gamma \geq 0} \tau_\gamma$ is also equal to T'. Since $T' \neq \top_T$, this is a contradiction.

If exactly one $\tau_k \subset T'$ and the rest are equal to T', then $\bigcap_{\gamma \geq 0} \tau_\gamma$ is equal to τ_k, which cannot be $\top_T$ either.

Therefore, there are at least two sets, τ_i and τ_j, which are strict subsets of T'. Since both τ_i and τ_j are non-empty and their intersection equals $\top_T$, τ_i and τ_j must be disjoint. If τ_i and τ_j are disjoint, they must correspond to different call sites.

In other words, there are at least two calls on the same receiver for which the static-type function is a strict subset of the set of types compatible with a given receiver r. It follows that both calls have to be polymorphic. Therefore, $r \in R^{\Subset}$. □

We will now show that if a receiver ever gets mapped to top, then it is a correlated-calls receiver.

Lemma 13. *For an IFDS problem P, let $n \in N^*$ be a node, and $d \in D$ a dataflow fact such there exists a realizable path $p \in \mathit{RP}(n, d)$. Then, if there exists a receiver $r \in R$, such that*

$$\mathcal{R}\left(\mathcal{T}^{\Subset}_{\{r\}}(P)\right)(n,\, d)(r) = \top_T \,,$$

then $r \in R^{\Subset}$.

Proof. As shown in (23),

$$\mathcal{R}\left(\mathcal{T}^{\Subset}_{\{r\}}(P)\right)(n,\, d)(r) = \bigcup_{q \in \mathsf{VP}(n)} \bigcup_{q' \in \mathsf{RP}(q,\, d)} \mathsf{EdgeFn}^{\Subset}_{\{r\}}(q')(\top_{\Subset})(r)\,.$$

Since the latter is equal to $\top_T$, it follows that for each realizable path p' to node n, $\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p')(\top)(r) = \top_T$. According to Lemma 13, this is only possible if $r \in R^{\Subset}$. □

Finally, we present the proof for Lemma **??** which states that a correlated-calls analysis that considers all receivers computes the same result as an analysis that considers only correlated-call receivers.

Proof of Lemma 6. From (**??**) we know that

$$\mathcal{U}^{\Subset}(\mathcal{R}\left(\mathcal{T}^{\Subset}_{R}(P)\right)) = \left\{(n,\, D^{\Subset}_{n}(\mathcal{R}(\mathcal{T}^{\Subset}_{R}(P)))) \mid n \in N^*\right\}.$$

According to (**??**) and Lemma 10, for a given $n \in N^*$,

$$\begin{aligned}
&D^{\Subset}_{n}(\mathcal{R}(\mathcal{T}^{\Subset}_{R}(P))) \\
&\quad = \left\{d \mid d \in \mathsf{MVP}_F(n) \,\wedge\, \forall r \in R: \; \left\{(r,\, \mathcal{R}(\mathcal{T}^{\Subset}_{\{r\}}(P))(n,\, d)(r)) \mid r \in R\right\}(r) \neq \top_T\right\} \\
&\quad = \left\{d \mid d \in \mathsf{MVP}_F(n) \,\wedge\, \forall \boldsymbol{r} \in \boldsymbol{R}: \; \mathcal{R}(\mathcal{T}^{\Subset}_{\{r\}}(P))(n,\, d)(r) \neq \top_T\right\}.
\end{aligned}$$

Since, according to Lemma 13, $\mathcal{R}(\mathcal{T}^{\Subset}_{\{r\}}(P))(n,\, d)(r)$ can only be equal to $\top_T$ when $r \in R^{\Subset}$, we can conclude that

$$\begin{aligned}
&D^{\Subset}_{n}(\mathcal{R}(\mathcal{T}^{\Subset}_{R}(P))) \\
&\quad = \left\{d \mid d \in \mathsf{MVP}_F(n) \,\wedge\, \forall \boldsymbol{r} \in \boldsymbol{R}^{\Subset}: \; \mathcal{R}(\mathcal{T}^{\Subset}_{\{r\}}(P))(n,\, d)(r) \neq \top_T\right\} \\
&\quad = D^{\Subset}_{n}(\mathcal{R}(\mathcal{T}^{\Subset}_{R^{\Subset}}(P))).
\end{aligned}$$

Therefore,

$$\begin{aligned}
\mathcal{U}^{\Subset}(\mathcal{R}\left(\mathcal{T}^{\Subset}_{R}(P)\right)) &= \left\{(n,\, D^{\Subset}_{n}(\mathcal{R}(\mathcal{T}^{\Subset}_{R^{\Subset}}(P)))) \mid n \in N^*\right\} \\
&= \mathcal{U}^{\Subset}(\mathcal{R}\left(\mathcal{T}^{\Subset}_{R^{\Subset}}(P)\right)).
\end{aligned}$$

□

Representation of Micro Functions

In this part of the appendix we present the proofs to the Lemmas related to the representation of micro functions with update maps.

Proof of Lemma ??. Let $\mathsf{update}^*_{f,\,r} = \langle I,\, U\rangle$. For any $\tau \in T$,

$$\begin{aligned}
\left[\!\left[\mathcal{N}(\mathsf{update}^*_{f,\,r})\right]\!\right](\tau) &= [\![\mathcal{N}(\langle I,\ U\rangle)]\!](\tau)\\
&= [\![\langle I \cup U,\, U\rangle]\!](\tau)\\
&= \tau \cap (I \cup U) \cup U\\
&= (\tau \cap I) \cup (\tau \cap U) \cup U\\
&= \tau \cap I \cup U\\
&= [\![\langle I,\, U\rangle]\!](\tau)\\
&= \left[\!\left[\mathsf{update}^*_{f,\,r}\right]\!\right](\tau)\,.
\end{aligned}$$

Thus, $\left[\!\left[\mathcal{N}(\mathsf{update}^*_{f,\,r})\right]\!\right] = \left[\!\left[\mathsf{update}^*_{f,\,r}\right]\!\right]$. □

Proof of Lemma ??. Let us show that there always exists a set $\tau \subseteq T$ such that $[\![\langle I,\, U\rangle]\!](\tau) \neq [\![\langle I',\, U'\rangle]\!](\tau)$. There are two cases:

1. $U \neq U'$. Then for the empty set $\tau = \varnothing$,

$$[\![\langle I,\, U\rangle]\!](\tau) = [\![\langle I,\, U\rangle]\!](\varnothing) = (\varnothing \cap I) \cup U = U\,,$$

whereas

$$[\![\langle I',\, U'\rangle]\!](\tau) = [\![\langle I',\, U'\rangle]\!](\varnothing) = (\varnothing \cap I') \cup U' = U'.$$

Hence, $[\![\langle I,\, U\rangle]\!] \neq [\![\langle I',\, U'\rangle]\!]$.

2. $I \neq I'$. Then for the set of all types $\tau = T$,

$$[\![\langle I,\, U\rangle]\!](\tau) = [\![\langle I,\, U\rangle]\!](T) = (T \cap I) \cup U = I \cup U\,.$$

Since $\langle I,\, U\rangle$ is normalized, $U \subseteq I$, and

$$I \cup U = I\,.$$

At the same time,

$$[\![\langle I',\, U'\rangle]\!](\tau) = [\![\langle I',\, U'\rangle]\!](T) = (T \cap I') \cup U' = I' \cup U' = I'\,.$$

Since $I \neq I'$, it follows that $[\![\langle I,\, U\rangle]\!] \neq [\![\langle I',\, U'\rangle]\!]$. □

Proof of Lemma ??. 1. The identity function is represented as

$$[\![\mathsf{id}]\!] = \{(r,\, \langle \bot_T,\, \top_T\rangle) \mid r \in R^{\Subset}\}\,;$$

the top function is represented as

$$[\![\lambda m\,.\,\top_{\Subset}]\!] = \{(r,\, \langle \top_T,\, \top_T\rangle) \mid r \in R^{\Subset}\}\,.$$

2. Equations (**??**) and (**??**) show that the representation of micro functions is closed under composition and meet.

3. To show that our representation for micro functions forms a lattice with finite height, let us first show that $L^{\Subset}_{R^{\Subset}} : R^{\Subset} \rightarrow 2^T$ forms a lattice. Since T is a finite set, $(2^T, \subseteq)$ is a finite-height lattice. $R^{\Subset}$ is a finite set. Hence, the mapping
$$R^{\Subset} \mapsto 2^T = \{(r, t) \,|\, r \in R^{\Subset}, t \in 2^T\} = L^{\Subset}_{R^{\Subset}}$$
also forms a finite-height lattice [15].
Furthermore, $L^{\Subset}_{R^{\Subset}}$ is a finite set. Every element of $L^{\Subset}_{R^{\Subset}}$ can be applied to $|R^{\Subset}|$ receivers, where each receiver is mapped to a set of types. There are $|R^{\Subset}| \cdot 2^{|T|}$ different possibilities to form those mappings, so
$$|L^{\Subset}_{R^{\Subset}}| = |R^{\Subset}| \cdot 2^{|T|}.$$
Therefore, $L^{\Subset}_{R^{\Subset}} \mapsto L^{\Subset}_{R^{\Subset}}$ also forms a finite-height lattice.
4. All operations can be computed in $O(R^{\Subset} \times T)$ time. Note that the $R^{\Subset}$ and T sets are an input to the correlated-calls analysis, and the time it takes to compute the meet or composition of micro functions is independent of the representation of the specific operand micro functions.
5. The space bound is $O(R^{\Subset} \times T)$.

□