Technical Report

Nr. TUD-CS-2013-0253
September 16th, 2013

Efficiently updating IDE-based data-flow analyses in response to incremental program changes

TECHNISCHE UNIVERSITÄT DARMSTADT

EC SPRIDE
EUROPEAN CENTER FOR SECURITY AND PRIVACY BY DESIGN

Authors
Steven Arzt (EC SPRIDE / Technische Universität Darmstadt)
Eric Bodden (EC SPRIDE / Fraunhofer SIT, Technische Universität Darmstadt)

Efficiently updating IDE-based data-flow analyses in response to incremental program changes

Steven Arzt[1] and Eric Bodden[1,2]
Secure Software Engineering Group
European Center for Security and Privacy by Design (EC SPRIDE)
[1]Technische Universität Darmstadt, [2]Fraunhofer SIT
Darmstadt, Germany
{firstname.lastname}@ec-spride.de

ABSTRACT

Most application code evolves incrementally, and especially so when being maintained after the applications have been deployed. Yet, most data-flow analyses do not take advantage of this fact. Instead they require clients to recompute the entire analysis even if little code has changed—a time consuming undertaking, especially with large libraries or when running static analyses often, e.g., on a continuous-integration server.

In this work, we present Reviser, a novel approach for automatically and efficiently updating inter-procedural data-flow analysis results in response to incremental program changes. Reviser follows a clear-and-propagate philosophy, aiming at clearing and recomputing analysis information only where required, thereby greatly reducing the required computational effort. The Reviser algorithm is formulated as an extension to the IDE framework for Inter-procedural Finite Distributed Environment problems and automatically updates arbitrary IDE-based analyses.

We have implemented Reviser as an open-source extension to the Heros IFDS/IDE solver and the Soot program-analysis framework. An evaluation of Reviser on various client analyses and target programs shows performance gains of up to 80% in comparison to a full recomputation. The experiments also show Reviser to compute the same results as a full recomputation on all instances tested.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Design, Languages, Performance

Keywords

Inter-procedural static analysis, flow-sensitive analysis, IFDS, IDE, Incremental analysis

1. INTRODUCTION

Within minutes to hours, static data-flow analysis can often produce results that would take weeks or even months to derive through manual code inspection. Code analyses are used for a variety of tasks including program understanding, compiler optimization, and security analysis. On the other hand, though, highly precise static analyses on large code bases are often still quite expensive to compute.

As others have argued before [5, 18, 19], a large margin of this inefficiency is due to the fact that current data-flow analyses lack a way to respond to incremental program changes. Once an expensive analysis run of some program p has completed, whenever p changes the analysis will typically need to be re-computed entirely once again—a costly undertaking. This is especially problematic with large libraries that change rarely but contribute much to the overall size of the program.

In this work we present Reviser, an approach for automatically and efficiently updating static-analysis results for a broad class of inter-procedural, flow-sensitive, context-sensitive data-flow analyses. Reviser assumes that analyses are implemented in Sagiv, Reps and Horwitz's framework for Inter-procedural Distributive Environment Transformers (IDE) [20] or within the IFDS framework for Interprocedural Finite Distributive Subset problems [16], an often-used specialization of IDE. Both frameworks require that flow functions are distributive over the merge operator. Although this is a limitation in some cases, IFDS and IDE have been used to express a large variety of practical analysis problems such as secure information flow [7, 9, 14], typestate [6, 12], alias sets [13], specification inference [22], and shape analysis [17, 26].

The IFDS/IDE frameworks allow data-flow analyses to be expressed in a template-driven style. This means that users simply define a set of flow functions. A generic IDE solver then uses those flow functions to compute analysis results for the entire program. Reviser replaces the standard IDE solver with a solver that automatically copes with incremental program changes, at the same time allowing users to reuse the original flow functions with only minor modifications and without restricting the expressiveness of the IDE framework. Reviser works by first applying the usual IDE algorithm once to the application's entire code base. Then, on receiving a modified version of the code, it determines the changes in terms of the application's control flow graph. Afterwards, Reviser invokes a specialized "revision algorithm" that updates analysis results only where necessary. This algorithm is the core contribution of this paper.

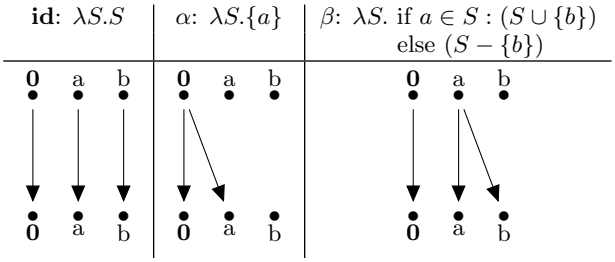


Figure 1: Function representation in IFDS, reproduced from [16]

A tool such as Reviser is useful if it produces correct and precise results, saves analysis time, and does not impose too large a burden on the static-analysis programmer. We have thus evaluated Reviser on a number of client analyses and target programs including revision histories of JUnit [8] as well as the abc [1] and Soot [25] frameworks, showing that Reviser produces the same results as a full recomputation, can save up to 80% of the time required for a full recomputation. At the same time, client analyses only require simple changes to be made compatible with Reviser.

To summarize, this paper presents the following original contributions:

- an algorithm for computing over-approximate structural diffs for code changes over a control-flow graph,
- an algorithm for efficiently updating IDE results based on a structural comparison of two program versions,
- an open-source implementation of the approach, and
- a set of experiments showing that the implementation computes correct (and precise) results, and saves, on average, about 80% of analysis time in comparison to a full recomputation.

The implementation of Reviser is available as an open-source extension to the language-independent Heros [2] IDE solver and the Soot program analysis framework [11, 25] for Java, along with all benchmarks, documentation and scripts necessary to reproduce our experimental results: `https://github.com/StevenArzt/reviser`

The remainder of this paper is structured as follows. Section 2 gives background information about the IFDS and IDE frameworks. In Section 3, we present the core of this paper, Reviser's algorithm for incremental recomputation. Section 4 presents important details about the implementation, and particularly discusses design decisions and how client analyses need to be adapted for use with Reviser. In Section 5, we discuss the algorithm for structurally comparing two versions of a program. Section 6 presents our experimental results. We discuss related work in Section 7 and conclude in Section 8.

2. BACKGROUND ON IFDS/IDE

Reviser applies to analyses formulated as Inter-procedural Distributive Environment Transformers. For such analyses, the IDE framework [20] defines a very efficient, tabulation-based solution strategy based on computing summary functions for each method. To achieve efficiency, these summaries are only computed once, to achieve context-sensitivity, though, they are re-applied anew at every calling context. While many ideas of Reviser may carry over to updating inter-procedural analysis results in general, our current formulation of Reviser exploits certain properties of the IDE framework that may not easily carry over to a more general setting.

IDE is a generalization of IFDS, a solution framework for Inter-procedural Finite Distributive Subset Problems. Since IFDS is easier to understand than IDE, we will present IFDS first and also focus the remaining presentation in this paper on IFDS, explaining elements important to treating full IDE only where necessary.

2.1 Overview of the IFDS Framework

The major idea of the IFDS framework is to reduce any program-analysis problem formulated in this framework to a pure graph-reachability problem. Based on the program's inter-procedural control-flow graph, the IFDS algorithm builds a so-called *exploded super graph*. The exploded graph contains a node for every combination of a statement (a node in the original control flow graph) and a statically decidable *fact* about the program. In an information-flow analysis, for instance, the node $(s, \mathbf{x})$ could denote that variable `x` holds confidential data at statement s. Edges between nodes in the exploded supergraph model data flow functions. Figure 1 gives three different examples on how simple flow functions can be represented as graph segments in IFDS. The nodes at the top represent facts before the given statement s, the nodes at the bottom represent facts after s. At the very left, the identity function **id** maps each data-flow fact before a statement onto itself. The special fact **0** is associated with every node in the exploded supergraph and denotes a tautology, a fact that always holds. As can be seen in the middle flow function α, this fact can be used to unconditionally generate the data-flow fact **a**: by connecting it to **0**, the fact becomes unconditionally reachable in the graph. At the same time, α kills the data-flow fact **b** by making it no longer reachable.

Function β at the right side is a typical function modeling data-flow through an assignment statement `b=a`. Here, `a` has the same value as before the assignment, modeled by the arrow from **a** to **a**, and `b` obtains `a`'s value, modeled by the arrow from **a** to **b**. The previous value associated with `b` is killed in the process: there is no edge from **b** to **b**.

It is important to note that data-flow facts are by no means limited to simple values such as the local variables in our example. Much more sophisticated abstractions exist, in which facts can, for instance, model aliasing through sets of access paths [24] or even the abstract typestate of combinations of multiple objects [12]. The IFDS framework itself, however, is oblivious to the concrete abstraction being used; the abstraction is a free parameter to the framework.

IFDS is efficient because its graph-based data-flow functions can easily be composed to so-called *path edges*. A path edge always starts at a method's header and ends at some statement within the same method. The edge summarizes the data flows along all paths from the method's entry to this statement. Path edges that end at one of the method's exit points are called summary edges. (See Figure 3 on page 6 for an example.) Crucially, the IFDS solution algorithm computes such summaries only once for each method. Reviser must take care to update these summaries where required.

2.2 The IDE Framework

As in IFDS, the IDE framework [20] models data flow through edges in an exploded super graph. In addition to

IFDS, however, IDE allows for the computation of distributive functions along those edges. A fact is no longer a simple value, but an *environment*, mapping facts $d \in D$ to values v from a separate value domain V. The flow functions thus transform environments $\{d \mapsto v\}$ to other environments $\{d' \mapsto v'\}$. In comparison to IFDS, this widens the class of problems that can be expressed in the framework.

In general, IDE computes and maintains the following types of functions:

Flow functions are used to compute environment transformers between nodes (s, d), and (s', d'). Flow functions themselves are never stored, they are just used to compute longer and longer jump functions.

Jump functions correspond to path edges in IFDS. They encode summaries of flow functions, i.e., compositions of environment transformers. They effectively store the composition of flow functions computed from the method's start point to the current statement. After the entire method was processed, jump functions are turned into summary functions.

Summary functions encode the summarized effects of a method. They are jump functions to the exit statements (return/throw) of the respective method. Summaries save the solver from having to recompute a callee's information anew for every call site.

If a statement is changed, its flow function may also change. As a consequence, the jump functions of all transitive successors may change as well, since they encode information computed involving the changed statement's flow function. The same holds for the summary functions of the method containing the changed statement. Section 3 describes in detail the revisions that Reviser needs to perform to bring jump and summary functions back to a correct state after an incremental program update.

One important difference between IFDS and IDE is that the IDE algorithm does not just compute graph reachability. Instead it comprises a second phase in which the computed environment transformers are actually applied along all edges in the graph to yield individual values for every program statement. Reviser incrementally updates both phases of the IDE algorithm.

3. INCREMENTAL UPDATES

Reviser assumes a scenario in which the user first triggers a complete analysis computation within the IFDS/IDE framework and then submits an incremental program update, for instance by committing a new version of the code to a version-control system or by triggering an incremental build in the user's integrated development environment. Reviser first compares both code versions, generating a structural diff. This diff gives information about which nodes and edges in the program's inter-procedural control-flow graph (ICFG) were added or removed.

In the following, we will call all nodes that were added or removed *changed* nodes. One of the guiding principles of Reviser is to infer from all changed nodes the set of so-called *affected* nodes, i.e, nodes whose information was outdated by the change and needs to be updated. Reviser treats as affected all nodes reachable from changed nodes when following edges in the novel, updated ICFG. This is an over-approximation. Reviser then follows a *clear-and-propagate* strategy: for each affected node it first clears the analysis information computed and then re-propagates the information from all the node's predecessors.

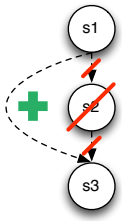
Note that by this design it is always safe to over-approximate the set of affected nodes. Assuming some un-affected nodes as affected will lead to superfluous re-computation but it can never lead to incorrect analysis results. Determining the set of affected nodes precisely can sometimes be more time consuming than re-computing additional analysis information. For this reason, Reviser does not necessarily seek to determine a maximally precise set of affected nodes but rather an efficiently computable approximation that is good enough in practice. As our experiments show, the set of affected nodes is usually just a small fraction of all ICFG nodes yielding a high potential for approaches such as Reviser.

The remainder of this section explains Reviser's *clear-and-propagate* algorithm in detail. Section 3.1 focuses on how to identify affected nodes. Section 3.2 then introduces the notion of *safe* and *start* nodes, and explains how re-propagations are triggered. Section 3.4 discusses how Reviser handles method calls. Section 3.5 explains how to incrementally update the actual result values that IDE (opposed to IFDS) computes for each statement, while Section 3.6 presents some important optimizations.

3.1 Computing Changesets

We assume a structural differencing algorithm which models all code alterations through added and deleted statements, represented by added and deleted nodes and edges in the inter-procedural control-flow graph. In particular, modified statements are represented through combinations of added and deleted statements.

Given the structural diff, one can express the differences between two program versions using two sets E^+ and E^-, containing all edges added, respectively removed, from the original ICFG. In the example shown to the right the original edges $s1 \to s2$ and $s2 \to s3$ are removed and a new edge $s1 \to s3$ is added. For simplicity, we also define the sets N^+ for the added nodes ($\emptyset$ in the example) and N^- for the removed nodes ($\{s2\}$ in the example).



Reviser uses this program-change information to find affected nodes. In general, Reviser considers as affected all nodes that are transitively reachable from changed nodes via the new ICFG's successor relation. Reviser does implement some optimizations, however, to further restrict this set. We will discuss those optimizations in Section 3.6.

Note that while Reviser itself is language independent, the ICFG is always language specific. For our instantiation for Java, Reviser adds all nodes of added methods and removes all nodes of removed methods, along with the respective call edges. For modified methods, Reviser uses a lightweight graph-differencing algorithm which computes a superset of the added and deleted control-flow graph edges. For performance reasons, Reviser approximates the computed superset, and is thus not always maximally precise. As our experiments show, however, the computed sets are typically small enough to yield a short re-analysis time. Section 5 presents Reviser's current differencing algorithm in detail.

```
1 void test() {
2     a = 42;
3     while (a > 40) {
4       a--;
5       a = 42;
6     }
7     print(a);
8   }
```

Listing 1: Safe and Non-Safe Nodes

Algorithm 1 presents Reviser's update algorithm, as an initialization phase, followed by two outer loops (Phases A and B) and a value-computation phase that is required for IDE problems only (not for IFDS). Phases Phases A and B call the function *ForwardTabulateSLRPs*, which is described in Algorithm 2. But let us consider the initialization phase first. The computation of change sets is shown in line 3. Here the algorithm populates the four sets E^+, E^-, N^+ and N^- using the differencing algorithm. It then purges E^+ and E^-, removing edges starting at changed nodes. Since the remaining algorithm will consider such edges automatically, this avoids superfluous re-computations.

Line 10 clears the outdated analysis information, i.e., path edges, at deleted nodes. For efficiency, Reviser clears the outdated analysis information for all other affected nodes on the fly. Section 3.3 will give further details. Next, we discuss how Reviser initializes its repropagation in Phase A.

3.2 Safe Repropagations

After deleting outdated analysis information at a changed node c, Reviser must compute the updated analysis information. Computing this updated information is easy assuming that c's predecessor nodes are unaffected, i.e., their IDE results are unchanged. We call such predecessors *safe*.

It is important to note that not all statements unchanged by the incremental code update are automatically safe. Listing 1 gives an example. Assume that the incremental update removes line 5. Further assume that the analysis is computing constant propagation. Both lines 2 and 4 are unchanged but, while line 2 is safe, line 4 is not. The reason is that it is reachable from line 5, and its abstract value even depends on the one computed at line 5. Generally problematic are thus recursive data-flow dependencies introduced by loops.

To find nodes guaranteed to be safe, Reviser thus walks up the control flow graph. For every changed node, it follows the sequence of predecessors until it finds a node which is not itself (transitively) preceded by any changed nodes. This node is then necessarily safe. If the changed node is not part of a loop, the candidate for the safe node is always the direct predecessor of the first changed node in the statement sequence. If the changed node is part of a loop, Reviser selects the predecessor of the head of the outermost loop. Then, if still at an affected node, Reviser walks up the statement sequence until it reaches the predecessor of the first affected node and regards that node as a safe node. This is sound because nodes that are neither changed nor have any affected predecessors are safe. Note that the converse does not hold in the general case. In Algorithm 1, this is done in lines 21 to 25 for all changed nodes.

Note that predecessors of safe nodes are always safe as well if they are not changed nodes on their own. This directly follows from the definition of a safe node. Therefore, one simple selection of safe nodes would be all of the program's

Algorithm 1 Incremental IDE - Initialization & outer loop

```
1: procedure IncrementalAnalysis(c^old, c^new)
2:   // Get the edges we must update
3:   ⟨E+, E−, N+, N−⟩ := ComputeCFGChanges(c^old, c^new)
4:   // Purge the edges
5:   E+ := E+ \ {(n, m) : n ∈ N+}
6:   E− := E− \ {(n, m) : n ∈ N−}
7:   E# := ∅ // callers of modified methods, see Sec. 3.6
8:   // Delete all facts for all deleted statements
9:   for all n ∈ N−; d1, d2 ∈ D; v ∈ V do
10:    PathEdge := PathEdge \ ⟨d1, n, d2⟩
11:    val := val \ ⟨n, d1, v⟩
12:    EndSum[⟨n, d1⟩] = ∅
13:  if E+ ∪ E− = ∅ then return
14:  changedNodes := ∅
15:  chgEndSums := ∅
16:  WorkList := ∅
17:  allChangedNs := ∅
18:  oldES := EndSummaries
19:  // Phase A: Update jump functions
20:  for all ⟨n1, n2⟩ ∈ (E+ ∪ E− ∪ E#) do
21:    if isPartOfLoop(n1) then
22:      ls := getLoopStart(n1)
23:      N := getPredsOf(ls)
24:    else
25:      N := {n1}
26:    for all n ∈ N, d1, d2 : (d1, n, d2) ∈ PathEdge do
27:      WorkList := WorkList ∪ {⟨d1, n, d2⟩}
28:    ForwardTabulateSLRPs(Update, c^new)
29:    // Return node or end summary changed?
30:    if ∃e_p ∈ e_proc(d2) : e_p ∈ (N+ ∪ N−)∨
31:          ∃d1 ∈ D, s_p ∈ s_proc(d2) :
32:            oldES[⟨s_p, d1⟩] ≠ EndSummaries[⟨s_p, d1⟩] then
33:      for all c ∈ CallSite(proc(d2)), d ∈ succs(c) do
34:        E# = E# ∪ ⟨c, d⟩
35:    allChangedNs = allChangedNs ∪ changedNodes
36:    changedNodes = ∅
37:  // Phase B: Recompute information at merge points
38:  for all n ∈ allChangedNs do
39:    // Check for merge point, both straight-line and
40:    // interprocedural
41:    preds := {m : m → n ∈ c^new}
42:    if |preds| ≥ 2 then
43:      for all m ∈ preds do
44:        for all d1, d2 : (d1, m, d2) ∈ PathEdge do
45:          WorkList := WorkList ∪ {⟨d1, m, d2⟩}
46:  ForwardTabulateSLRPs(Compute, cfg)
47:  // Incremental Phase II from [20]
48:  for all n1 ∈ allChangedNs do
49:    for all n_i : ∃n1, ..., n_i ∈ N; ∀i : n_i → n_{i+1} ∈ cfg do
50:      for all d ∈ D; val(n, d) ≠ ∅ do
51:        val(n, d) = ∅
52:  for all n1 ∈ allChangedNs do
53:    // Run original Phase II for (n,d), see [20, page 149]
```

entry points. This choice will often be sub-optimal, however, as it could cause the unnecessary re-computation also at safe nodes. Reviser therefore determines start nodes as follows.

For a given affected node a and a path p to a, a safe node s on this path is considered a *start node* if there is no other safe node closer to s on p. One observation following from this definition is that if an affected node has multiple (transitive) predecessors, the start node need not be unique.

3.3 Iterative Propagation

Once start nodes are known, a trivial approach would be to just pass over the changed program twice: One would

Algorithm 2 Incremental IDE - Iterations

```
54: procedure ForwardTabulateSLRPs(mode, cfg)
55:   while WorkList ≠ ∅ do
56:     Pop an edge ⟨s_p, d_1⟩ → ⟨n, d_2⟩ from WorkList
57:     switch n do
58:       case n ∈ Call_p
59:         if d_2 = ϵ then
60:           MaybeClearAndPropagate(⟨d_1, retSite(n), ϵ⟩)
61:           continue
62:         for all d_3 ∈ passArgs(⟨n, d_2⟩) do
63:           Propagate(⟨d_3, s_calledProc(n), d_3⟩)
64:           Incoming[⟨s_calledProc(n), d_3⟩]∪ := ⟨n, d_2⟩
65:           for all ⟨e_p, d_4⟩ ∈ EndSum[⟨s_callee(n), d_3⟩] do
66:             for all d_5 ∈ retVal(⟨e_p, d_4⟩, ⟨n, d_2⟩) do
67:               MaybeClearAndPropagate(⟨d_1, retSite(n), d_5⟩)
68:           if retVal(⟨e_p, d_4⟩, ⟨n, d_2⟩) = ∅ ∧
69:            mode = Update then
70:             MaybeClearAndPropagate(⟨d_1, retSite(n), ϵ⟩)
71:       case n ∈ e_p
72:         // Clear all potentially outdated end summaries
73:         if mode = Update then
74:           if ⟨s_p, d_1⟩ ∉ chgEndSums then
75:             chgEndSums = chgEndSums ∪ ⟨s_p, d_1⟩
76:             EndSum[⟨s_p, d_1⟩] := ∅
77:         // Add new end summary
78:         if d_2 ≠ ϵ then
79:           EndSum[⟨s_p, d_1⟩] := EndSum[⟨s_p, d_1⟩] ∪ ⟨e_p, d_2⟩
80:         // Optimization: No automatic caller update
81:         if mode=Compute then
82:           for all ⟨c, d_4⟩ ∈ Incoming[⟨s_p, d_1⟩] do
83:             if d_2 = ϵ then
84:               Propagate(⟨d_1, retSite(c), ϵ⟩)
85:               continue
86:             returnVals = returnVal(⟨e_p, d_2⟩, ⟨c, d_4⟩)
87:             for all d_5 ∈ returnVals, d_3 :
88:              ⟨s_procOf(c), d_3⟩ → ⟨c, d_4⟩ ∈ PathEdge do
89:               Propagate(⟨d_3, retSite(n), d_5⟩)
90:       case n ∈ (N_p \ Call_p \ {e_p})
91:         if d_2 = ϵ then
92:           for all m : n → m ∈ c^new do
93:             MaybeClearAndPropagate(⟨d_1, m, ϵ⟩)
94:           continue
95:         succs = {⟨m, d_3⟩ : n → m ∈ cfg ∧ d_3 ∈ flow(⟨n, d_2⟩, π)}
96:         for all ⟨m, d_3⟩ ∈ succs do
97:           MaybeClearAndPropagate(⟨d_1, m, d_3⟩)
98:         if |succs| = 0 ∧ mode = Update then
99:           MaybeClearAndPropagate(⟨d_1, m, ϵ⟩)
```

Algorithm 3 Incremental IDE - Clear and Propagate

```
100: procedure MaybeClearAndPropagate(e := ⟨d_1, n, d_2⟩)
101:   if mode=Update then
102:     // Only clear if we haven't changed this node yet
103:     if n ∉ changedNodes then
104:       changedNodes := changedNodes ∪ {n}
105:       for all d_3 ∈ D : (d_1, n, d_3) ∈ PathEdge do
106:         PathEdge := PathEdge \ (d_1, n, d_3)
107:   if d_2 ≠ ϵ then
108:     Propagate(e)
```

first clear all IDE results at all affected nodes (the transitive successors of the start nodes as described in Section 3.1) and then start a new propagation from the start nodes as described in Section 3.2. For efficiency, however, Reviser follows a more advanced approach by combining both passes into a single *clear-and-propagate* step which we show in Algorithm 3. When Reviser processes a node, it first clears the IDE results associated with this node to then compute the new analysis information through a propagation from the node's already re-computed predecessors.

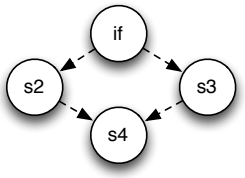


Figure 2: Example with control-flow branch

Reviser commences at the start nodes, clearing and re-propagating the analysis information for each node reachable from there. After this step, all affected nodes are associated with the correct analysis information for the new version of the target application. For our example from Listing 1, line 2 is the only safe node and thus also the only start node. By keeping a record of nodes already visited and never clearing a node twice, this procedure is guaranteed to reach a fixed point in all cases in which the forward propagation as defined in the original IDE algorithm reaches such a fixed point.

While efficient, this *clear-and-propagate* approach requires some precaution: if a node can be reached on more than one path, Reviser must make sure to clear it only once. The example in Figure 2 contains a simple conditional. Assume the *if*-statement to be a start node. In this case, statement $s4$ will be reached along two different paths. Without loss of generality, assume the propagation via the left branch to occur first. Reviser would first clear the information at $s4$ to then add the information computed along this branch. Afterwards, when processing the right branch, Reviser would clear the information at $s4$ once again, thus resulting in final analysis information for $s4$ which only contains results from the right branch. The jump functions previously computed along the left branch would be lost. To circumvent this problem, Reviser remembers all nodes for which the analysis information has been cleared already, preventing them from being cleared a second time, see line 103.

Additionally, Reviser must make sure to actually start a repropagation *on all paths* reaching an affected node n, even for paths to n which themselves contain no change. In our example from Figure 2, assume a statement in the right branch to be changed and the start node to also reside within the right branch as well. In this case, $s4$ would be cleared and the analysis information computed along the right branch would be added, but the one from the left branch would be lost since no propagation is ever performed along this path. The algorithm therefore performs in a second phase (Phase B), after all *clear-and-propagate* cycles have completed, a single forward-only propagation step for all control-flow merge points whose analysis information was updated (see line 37). In other words, Reviser starts an artificial propagation from all non-unique predecessors of nodes that have been cleared at some point during Phase A. This is correct due to the following observation: after the clear-and-propagate phase has completed, no outdated analysis information is left in the graph. Thus, a forward propagation cannot possibly re-introduce spurious analysis information.

The iterative steps of the incremental update (see Algorithm 2) are very similar to those of the original IDE algorithm. Generally, Reviser needs to perform a *clear-and-*

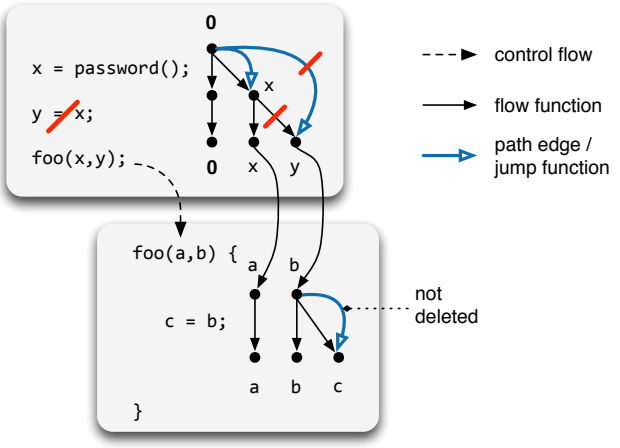


Figure 3: Update to method-call arguments

propagate step whenever the original algorithm performs a *propagate* step (see, for instance, line 97). For performance reasons, however, Reviser deviates from this rule in the case of method calls (see Section 3.4).

Finally, Reviser must make sure to clear the analysis information at a successor node even if no new analysis information is to be propagated because otherwise outdated analysis information would be retained. In such cases, Reviser needs to *clear* without having anything to *propagate*. Reviser thus uses the pseudo-value ϵ as a marker to denote the obligation to clear (lines 70 and 99). ϵ values are transparently repropagated to transitive successors just like new jump functions would be, so that all affected nodes are actually assured to be cleared (lines 60, 79, and 93).

3.4 Method Calls

IDE is a framework for solving *inter*-procedural analysis problems. When the IDE algorithm processes a call site, it uses the so-called *call flow function* to propagate the current analysis information (i.e., jump functions) to the start nodes of all possible callees. It would be sound for Reviser to replace this *propagate* step with a *clear-and-propagate* step (see line 63). This, however, would in some cases require the analysis to recompute the information for all methods in the analyzed program (for instance if the program's main method has changed), even if neither those methods' code nor their call parameters have changed. This would severely impair performance. As described in Section 3.3, Reviser thus treats call edges separately and for those edges only performs a regular forward propagation without clearing potentially outdated analysis information (see line 63). In result, a method is only entered if it is reached by new analysis information via the call edge.

In case an update to a caller deletes analysis information, this approach might lead to situations where jump functions associated with the callee become unreachable. Figure 3 shows such a situation. Assume a taint analysis tracking the return value of `password()`. After the code update removing the second assignment, Reviser deletes the caller-side jump function as indicated, but it retains the callee-side jump function from `b` to `c`, leaving this jump function unreachable in the super graph. This has the additional advantage of having the jump function still available, should it become reachable again later, but at the same time the disadvantage of wasting memory in the meantime. While our Reviser prototype does not clean up this memory, an implementation in an industrial setting would probably want to implement some idle-time garbage collection to collect unreachable functions at selected points in time.

3.5 Value Computation

In the above, we have explained how Reviser updates "analysis information". In IDE, this information actually itself resembles environment transformers, i.e., functions, which in a second step are applied to some initial values taken from a pre-defined value domain. Values are computed intra-procedurally, using jump functions. A value can thus only become outdated through a change to a jump function or through a change to values at the jump function's start node. The latter, however, can only change through a change to a jump function in the caller. Thus, the transitive closure of all nodes with changed jump functions safely overapproximates the set of expired values. Reviser deletes all values at these nodes (see lines 48 to 51 in Algorithm 1). Afterwards, to obtain all updated values, Reviser starts a regular value-propagation task from the start nodes of all changed methods (line 52). Since the new jump functions are complete and correct, and since the IDE value-propagation algorithm (which we use without changes) is sound, the incremental computation is sound as well. Note that there is no risk of repropagating old values since all values not guaranteed to still be valid have been removed.

3.6 Optimizations

Reviser applies a couple of sound optimizations to the *clear-and-propagate* principle. Firstly, when the iteration reaches a node that itself serves as a start node for a clear-and-propagate cycle, it can safely abort the propagation. (This can happen when the incremental update comprises changes to multiple parts of the code.) Secondly, if there are two changes in the same method, only the first one needs to be propagated if the clear-and-propagate cycles will reach the start node of the second change anyway. Note that the second optimization does not include the first one in the case of interprocedural loops. Finally, when processing return edges, Reviser does not need to clear-and-propagate back into the caller if the return site is a start node, since in this case this node will trigger a recomputation on its own. To avoid cluttering the presentation, these optimizations are not shown in Algorithm 1.

Reviser also implements another less obvious optimization: Instead of performing *clear-and-propagate* steps over return edges, Reviser stops at the end of each method (see line 81). After all updates in the current method have completed, Reviser checks whether either one of the return statements is in the set of changed nodes or whether an end summary entry has been changed (see line 32). Only if this is the case, Reviser adds the caller to the set of changed nodes (artificial changed edge set $E^{\#}$, see line 34) and schedules it to be updated. This is sound since the return edge is computed locally on the return statement and the jump functions reaching it in the callee. The latter, however, are equal to the end summaries for that function. (When we say that Reviser is *sound* then we mean that it yields the same analysis information as a full re-computation.)

In essence, the return statement serves as a checkpoint during which Reviser checks whether additional propagations are required. Such checkpoint could also be defined

at other statements. Initially we even experimented with an implementation that would first compute *all* analysis information for an entire statement to then discontinue the propagation if the analysis information obtained that way was found to be equal to the original information at this statement. We found, however, that this design was causing drastic slowdowns in our implementation. This is because our IDE solver Heros gains much efficiency by computing individual data flows concurrently. An eager equality check at each statement requires synchronizing the computations for each statement, however, which we found to cause too much thread contention to pay off in practice.

4. IMPLEMENTATION

We have implemented REVISER on top of the open source solver Heros [2]. Heros provides an open interface for ICFGs, which allows it to be used in combination with program-analysis frameworks for various target languages. We have extended this interface for changeset computation and provide a reference implementation which integrates with Soot [25] for analyzing Java programs. This raised some technical challenges as Soot normally does not support parts of the AST and intermediate program representation to be exchanged while Soot is running. We thus decided to decorate all relevant Soot objects with wrappers that support dynamic replacement. REVISER's modified IDE solver then accesses these wrappers only. Naturally, the indirection introduced through the wrappers incurs some overhead. As our evaluation shows, though, in practice the overhead is quite acceptable in comparison to the savings achieved through incremental evaluation.

Additionally, the client analysis developer is required to adapt his code using a simple template. Instead of directly accessing AST objects, he must request the corresponding wrapper from REVISER. Listing 2 shows how an uninitialized-variables analysis has to be adapted for the use with REVISER. The underlined changes show the calls to the wrapper construction and deconstruction methods as well as the special wrapper class *UpdatableWrapper* which takes the place of the original data types. Additionally, a special updatable interprocedural control flow graph must be used instead of the default one. For Soot and it's Jimple intermediate representation, we provide such an updatable CFG as a drop-in replacement.

5. GRAPH DIFFERENCING ALGORITHM

In this section, we describe an efficient, yet precise enough algorithm for computing the differences of two program versions. As explained in Section refSec:Incremental:Changesets, REVISER is compatible with all differencing algorithms capable of producing the four sets N^+, N^-, E^+ and E^-. Recall that REVISER only models added and removed nodes and egdes in the control flow graph, whereas changed edges are treated as the old edge being removed and the new one being added.

Our algorithm for computing code changes does not necessarily aim at computing a change set that is minimal but instead tries to compute as quickly as possible a change set that is sound, i.e., includes at least all edges that have actually been changed, and is also reasonable precise. If the change set is not minimal, this does not jeopardize soundness, although it may, in theory, impact performance. As our experimental results presented in Section 6 show, our change-set computation gives sufficiently precise results in practice.

```
1  id = readID();
2  manager = getUserManager();
3  cust = manager.EMTPY_EMPLOYEE;
4  if (id > 0)
5    cust = getCustomerByID(id);
6  print(cust);
```

Listing 3: Diff example: Customer management

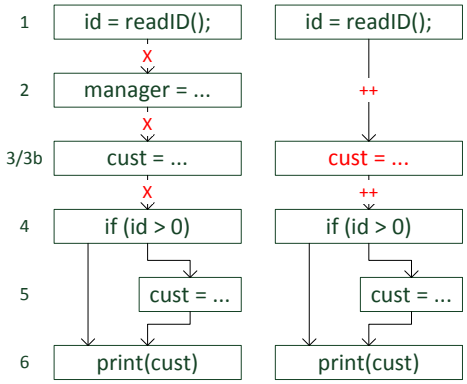


Figure 4: CFG changes in running example; crossed-out edges were removed, edges with ++ were added

As an example, take the Java code in Listing 3 from which we remove line 2 and change line 3 to no longer use the *EMPTY_EMPLOYEE* field, but instead create a new customer object. We would then expect the change set depicted in Figure 4. We label the new version of line 3 with line 3b. To compute this graph-based diff, we apply the greedy algorithm shown in Algorithm 4 to each method body in the target code. The algorithm first iterates over each statement in the new method to check whether it was already present in the old version (line 9). During this process, one must make sure not to consider statements that occur out of order: If statement s_1 is followed by statement s_2 in the new method, we only assume s_2 to be retained from the old version only if its old counterpart also follows a counterpart of s_1. In other words, the order of retained statements must be preserved.

To this end, the algorithm uses pointers to a current node for both the old and the new version of the code. The algorithm works by comparing the statements these pointers refer to as shown in Figure 5. One side (the left one in the figure) is called the "reference side": For every statement on the reference side , we try to find a corresponding one on the other side, called the "lookup side", by moving the pointer on the lookup side forward (line 27). If no corresponding statement was found (i.e. *findStatement* returns ϵ), we know that there must be a new edge in the reference side that does not exist on the lookup side, and record this edge as added (first part of line 30). We then restore the lookup side pointer to its original position as we have not found any better correspondance. On the other hand, if we have found a match, this gives us a new lookup pointer position for the subsequent searches (see line 35). Concerning edges, there are two possibilities: If we have found a direct successor, no

```
1  void analyzeUninitializedVariables() {
2    UpdatableInterproceduralCFG<Unit, SootMethod> icfg =
3      new UpdatableJimpleBasedInterproceduralCFG();
4
5    IFDSTabulationProblem<UpdatableWrapper<Unit>, UpdatableWrapper<Local>,UpdatableWrapper<SootMethod>
6      UpdatableInterproceduralCFG<Unit, SootMethod>> problem = new IFDSUninitializedVariables(icfg);
7    IFDSSolver<UpdatableWrapper<Unit>,UpdatableWrapper<Local>,UpdatableWrapper<SootMethod>,
8        UpdatableInterproceduralCFG<Unit,SootMethod>> solver =
9      new IFDSSolver<UpdatableWrapper<Unit>,UpdatableWrapper<Local>,UpdatableWrapper<SootMethod>,
10       UpdatableInterproceduralCFG<Unit,SootMethod>>(problem);
11
12   solver.solve();
13 }
```

Listing 2: Changes to Client Analysis

changes have happened between both versions of the code. The statements on both sides have the same direct successors. If we have however skipped statements and found a match further down, i.e. we have moved the pointer on the lookup side by more than one entry, this however gives us a new edge: The statement now has direct a successor that has not been there before (last part of line 30).

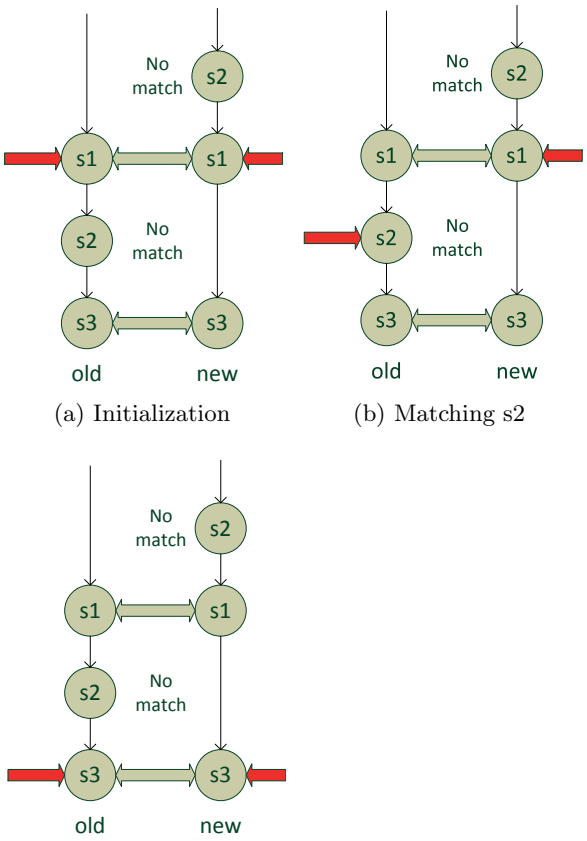
The pointer on the reference side is always moved down by one statement as we try to match every statement one at a time, regardless of whether or how far the pointer on the lookup side is moved.

The pointer on the lookup side is initialized with ϵ, a special value for indicating that there is no current node. If the current pointer is ϵ, the search starts above the first node. Note that if the first statement in the method has been changed so that we don't find a correspondance on the lookup side, the ϵ value is kept after initialization until the first reference-side node has been matched.

See Figure 5 for a graphical example of the algorithm. In this case, we intend to find edges that have been removed. We start at statement s_1, the first statement in the old version of the code. In the new code, we find s_1 at line 2 and move the pointers accordingly (see red arrows in Figure 5a). As of yet, no change is recorded. Afterwards, we find the successor statement s_2 of s_1. While looking forward from our current pointer position in "new", we find that there is no match for s_2, and mark s_2 as "removed". We thus reset the right pointer to s_1 (see Figure 5b). Next, we move the left pointer down by one to look for the next statement which is s_3. Statement s_3 can be matched on the right, and we set the right pointer to s_3 accordingly as shown in Figure 5c. We have then reached the end of the left method and are done. The algorithm computed that s_2 was removed, and that therefore $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_3$ must be removed from the graph. In the end, both pointers are at s_3.

To each pair of old and new control-flow graphs, the algorithm is essentially applied twice, once for finding new edges and once for finding removed ones. Mainly, this means that the lookup side is changed, that the other program graph is used for computing successor relations and that the interpretation of not finding a correspondence is swapped between "add edge" and "removed edge" (compare lines 9 and 10 in the algorithm). Since we regard changed statements as having been deleted and re-inserted, this method gives us all changed edges induced by the code change.

In practice, Reviser works on Java classes and not on single methods. This requires Reviser to first perform a structural analysis to detect new methods (all statements



(a) Initialization (b) Matching s2

(c) Matching s3

Figure 5: CFG changeset computation

induce added edges), removed methods (all statements induce removed edges) and changed methods that are analyzed as described. If a class is new, all its methods are regarded as new; if a class got removed we mark all methods as removed. In addition, modified code may lead to modified method-call edges. For the moment, Reviser re-resolves method-call edges by re-running a complete call-graph analysis on the new program version. Industrial-strenght implementations would probably combine our approach with incremental approaches to call-graph computations such as the one by Souter and Pollock [23].

6. EXPERIMENTS

Algorithm 4 Procedure diff algorithm

```
1: function ComputeCFGChanges(oldcfg, newcfg)
2:   // Get the old and new start points of all changed
3:   // methods
4:   for all ⟨s_p^old, s_p^new⟩ ∈ chgMeths(oldcfg, newcfg) do
5:     newN := ∅ // new nodes
6:     rmvN := ∅ // removed nodes
7:     newE := ∅ // new edges
8:     rmvE := ∅ // removed edges
9:     ⟨newN, newE⟩ := scan(newcfg, oldcfg);
10:    ⟨rmvN, rmvE⟩ := scan(oldcfg, newcfg);
11:    return ⟨newE, rmvE, newN, rmvN⟩
12:
13: function scan(c1, c2)
14:   queue := {⟨s_p^new, s_p^old⟩}
15:   doneList := ∅
16:   N := ∅
17:   E := ∅
18:   while queue ≠ ∅ do
19:     ⟨s1, s2⟩ := queue.pop()
20:     doneList := doneList ∪ {s2}
21:     for all s1' : s1 → s1' ∈ c1 do
22:       // Find the current left-side successor in the
23:       // successors of the right-side statement
24:       if s2 = ϵ then
25:         s2' := ϵ
26:       else
27:         s2' := findStatement(s2, s1', c2)
28:       // Changed edge on no match, changed node
29:       // or new/removed direct successor
30:       if s2' = ϵ ∨ s1 ∈ N ∨ s2 → s2' ∉ c2 then
31:         E := E ∪ {⟨notE(s2, s1), notE(s2', s1')⟩}
32:       if s2' = ϵ then
33:         N := N ∪ {s1'}
34:       if s1' ∉ doneList then
35:         queue := queue ∪ {⟨s1', notE(s2', s2)⟩}
36:   return ⟨N, E⟩
37:
38: function notE(s, t)
39:   if s = ϵ then return t
40:   else return s;
41:
42: function findStatement(s1, s2, c)
43:   doneList := ∅
44:   queue := {s1' : s1 → s1' ∈ c}
45:   while queue ≠ ∅ do
46:     s := queue.pop()
47:     doneList := doneList ∪ {s}
48:     // Check for statement equivalence. At the moment,
49:     // we use string equality.
50:     if s ≈ s2 then return s
51:     for all {s' : s → s' ∈ c} do
52:       if s' ∉ doneList then
53:         queue := queue ∪ s'
    return ϵ
```

Our evaluation addresses both the correctness and performance of Reviser's implementation. To assess correctness, we compare the results computed by a full recomputation on the target code with the results of an incremental update. To evaluate performance, we measure the time required to initially run Reviser on the original code, as well as the time required for the incremental update, and compare it to the time a full recomputation of the changed code would have taken in the original, unchanged Heros solver.

For all our tests, we use the following target applications and test cases:

JUnit Special Tests These test cases were hand written by ourselves; they comprise important corner cases such as changes in methods close to the entry point, loops in callees, etc. The tests all consist of code changes to JUnit 4.10. They are especially useful to assess the correctness of Reviser.

JUnit Update JUnit is upgraded from version 4.10 to version 4.11, changing a large amount of code. This test case evaluates a typical worst-case scenario to see whether the algorithm degenerates for large changes.

JUnit, Soot, and abc Revisions Performs incremental updates for 25 revisions from the JUnit and Soot git repositories as well as the abc SVN repository to evaluate Reviser on changes submitted to real-world version-control systems. We envision this to be the main scenario for Reviser: Frequent updates of smaller scale on a continuous integration server or even inside a development environment to ensure software quality.

All benchmarks are available on our project website.

Firstly, we ran every target application with an interprocedural reaching-definitions analysis. The results of this IFDS client analysis change at a rather large scale (every modified definition statement changes at least one result) and is thus a good candidate for evaluating the lower performance bounds of an incremental algorithm. Secondly, we ran an uninitialized-variable analysis on all target programs to evaluate Reviser with a more lightweight client analysis. For maximum analysis precision, the full JDK was included with the target program for all JUnit test cases. For the soot and abc test cases, memory constraints required us to exclude the JDK.

6.1 Implementation Correctness

To validate the correctness of our implementation, we applied Reviser to all test cases mentioned above, i.e., both the set of artificial test cases on JUnit as well as all real-world change sets. For two subsequent versions $v1$ and $v2$ of any target program, we first ran all analyses using the unmodified Heros solver on version $v2$ and recorded the results. Afterwards, we ran Reviser on $v1$, incrementally updated the results to version $v2$, and compared the results with the ones from the unchanged Heros solver. Our experiments confirm that Reviser computes the same results as a full recomputation in all cases.

6.2 Performance

All results reported in this section are averaged over 10 runs. Every run was allotted a maximum heap size of 35 GB on a computation server with 12 AMD Opteron 8356 cores running Debian Linux 2.6 with Oracle's Java HotSpot 64-Bit Server VM version 1.6.0. We allotted a large amount of memory since some client analyses require it. This is not a requirement introduced by Reviser. All times are measured for the client analysis only excluding aspects like loading the target program into memory or constructing a callgraph.

6.2.1 Reaching-Definitions Analysis

We first show how Reviser performs with an interprocedural reaching definitions analysis. The results of the JUnit

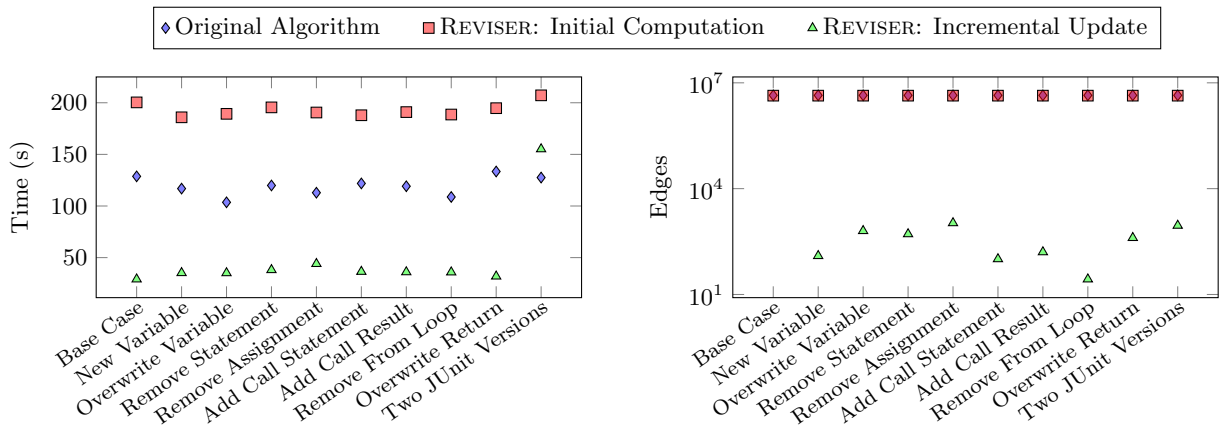


Figure 6: JUnit Special Tests - Reaching Definitions

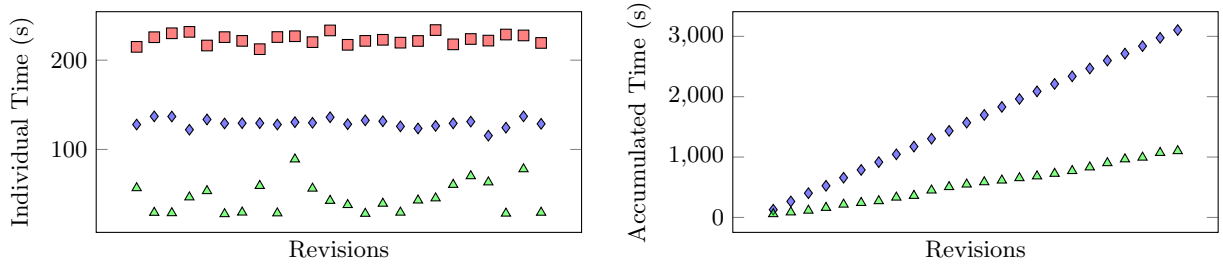


Figure 7: JUnit GIT Commits - Reaching Definitions

special tests cases and the JUnit version upgrade are shown in Figure 6. The left side shows the timings, the right side the number of processed edges. Note that the edge count is shown on a logarithmic scale. Though Reviser introduces some overhead on the initial computation, it requires only few edges to be recomputed, making incremental updates fast. Therefore, the initial overhead (which only occurs once) quickly pays off when performing multiple incremental updates. With the JUnit special tests, the first update already makes up for initial overhead, with Reviser saving about 70% of the time required for a full recomputation using the original, unmodified HEROS solver.

In the worst-case scenario of large changes to the code base as in the JUnit update case, shown as the rightmost data point in Figure 6, the time required for an incremental update (205 seconds) is approximately equal to the time of a complete recomputation in Reviser (215 seconds). In this scenario, 9 classes and 40 methods in existing classes were added, 28 methods were removed, and 87 methods were changed. The changed methods included *hashCode()* and *equals()* methods, requiring the changed return definitions to be propagated through all callers including those in the JDK. In result, in this case one really cannot expect much time to be saved by an incremental update.

On the 25 most recent check-ins to the JUnit GIT repository, Reviser greatly outperforms a recomputation. The results are shown in Figure 7 and are very similar to our hand-crafted test cases. Reviser saves about 65% of the full recomputation time with the original Heros solver and about 80% of its own initial computation time. In concrete numbers, incremental updates on the JUnit GIT commit set take 46 seconds on average, about 30 seconds of which are due to changeset computation. The right side of Figure 7 shows how the savings are accumulated over time to more than 3,000 seconds.

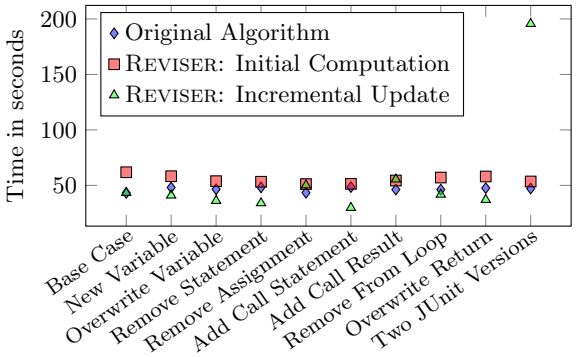


Figure 13: JUnit Special Tests - Uninitialized Variables

For the updates to the Soot repository, about 75% of both the initial computation time and the full recomputation time with an unchanged Heros solver are saved for the reaching definitions analysis, see Figure 8. An update takes 12 seconds on average; 10 seconds are were spent on changeset computation. The overhead of the initial computation is about 8 seconds or 18% on average.

On the 25 most recent commits to the SVN repository of the AspectBench Compiler abc [1], Reviser saves about 64% of the original computation time as shown in Figure 9. The initial overhead was below the standard deviation of the original computation time and thus negligible. Even in the worst case scenario in which a commit modularized and restructured the project, Reviser performed an update in about 95% of the original computation time.

6.2.2 *Uninitialized-Variable Analysis*

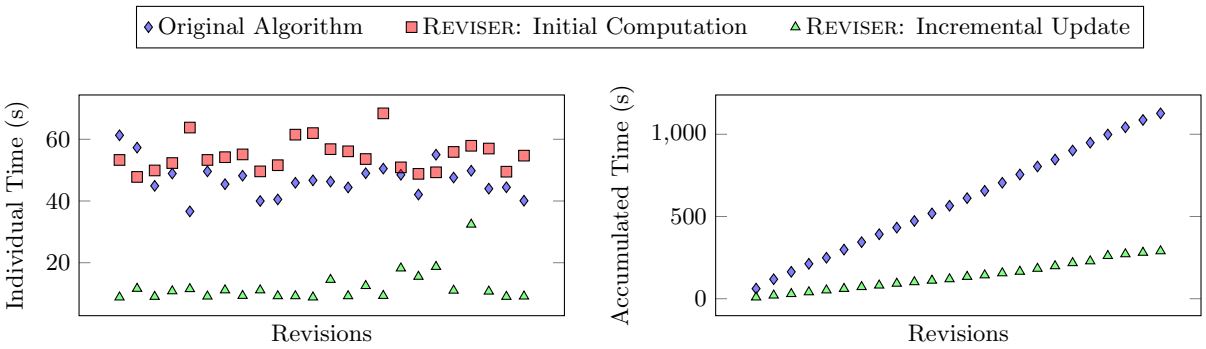


Figure 8: Soot GIT Commits - Reaching Definitions

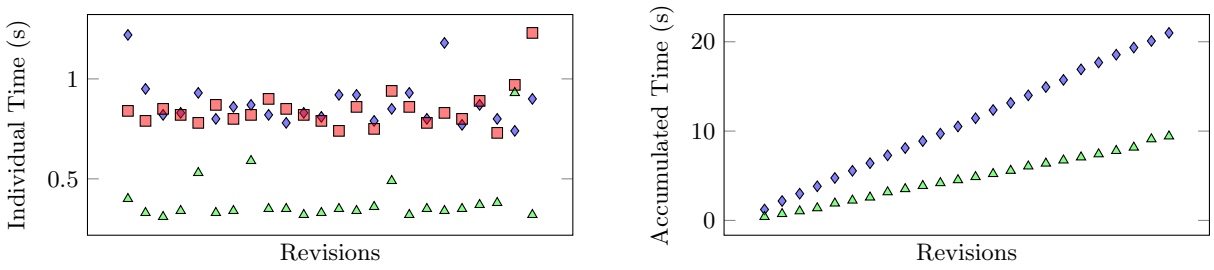


Figure 9: abc GIT Commits - Reaching Definitions

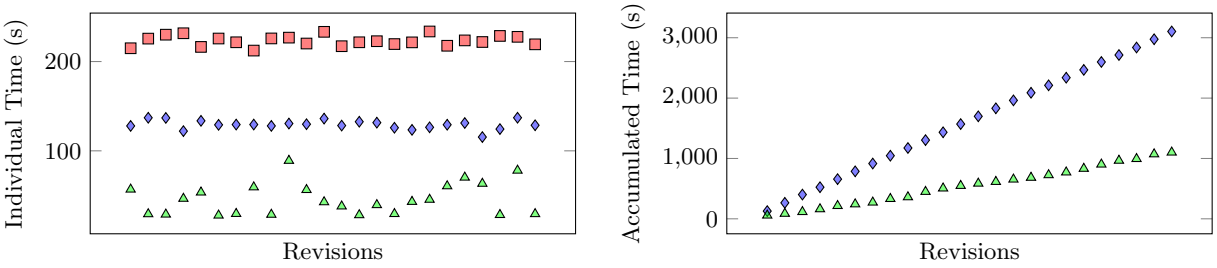


Figure 10: JUnit GIT Commits - Uninitialized Variables

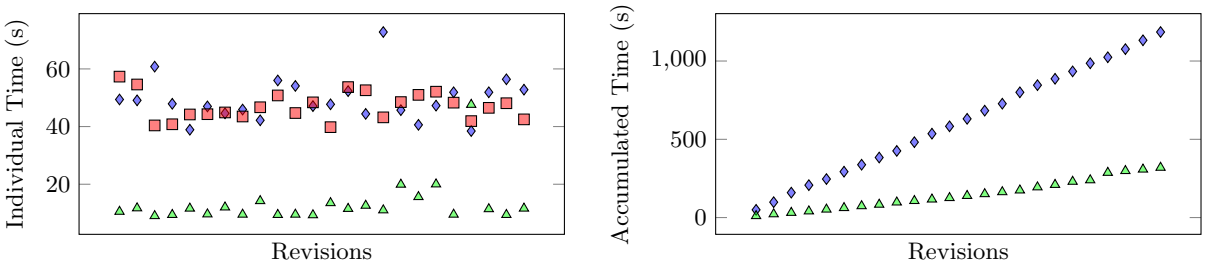


Figure 11: Soot GIT Commits - Uninitialized Variables

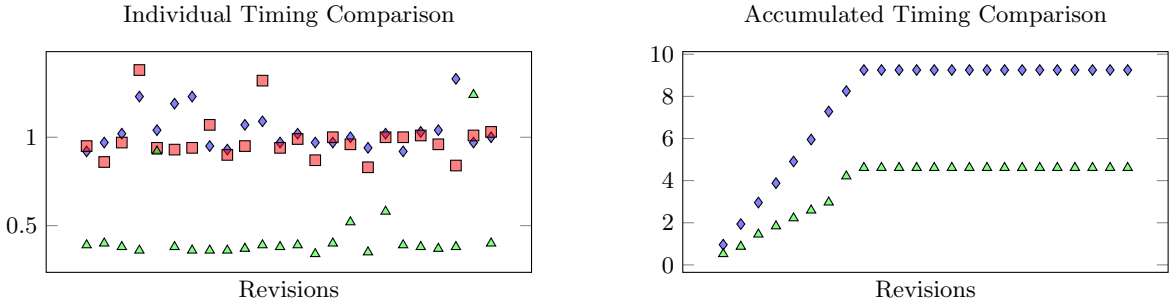


Figure 12: abc GIT Commits - Uninitialized Variables

On the specialized test cases, Reviser saves about 13% of the original time and about 26% of its own initial computation time when performing incremental updates. The overhead of the initial computation is about 20%. As shown in Figure 13, the savings are smaller than for the reaching definitions analysis because the analysis as such is much faster. Still, Reviser saves time in all cases except the version upgrade. Here, the huge number of changed edges severely affects performance.

On the 25 most recent commits to the JUnit repository (Figure 10), Reviser saves about 16% of is initial computation time (56 seconds on average) and about 21% of the original solver's time (59 seconds on average). The initial overhead is negligible since it was well inside the standard deviation of the original computation time (about 4 seconds).

For an uninitialized-variables analysis on the 25 most recent Soot commits, Reviser saves about 70% of both the initial and the original computation time, see Figure 11. An update takes about 13 seconds on average (19 seconds if only counting updates that actually changed code); 10 seconds thereof are spent on the changeset computation. The initial overhead is negligible for this experiment - the unchanged solver took 48 seconds on average with a standard deviation of 7.4 seconds, while Reviser took 47 seconds on average.

On the most recent commits to the abc repository, Reviser saves about 64% of the original computation time on average as shown in Figure 12 with an initial overhead of less than 9%. There is a single case in which the incremental update with Reviser takes 20% longer than the initial computation. This commit modularized abc, i.e., contained major code restructuring, which is rare.

6.3 Discussion

The performance advantage of Reviser in comparison to full recomputations is visible best when applied repeatedly. When we assume that the static analysis is performed on a continuous-integration server for every check-in into a version control system, savings are accumulated over time as depicted on the right sides of Figures 7–12. For instance, over the 25 most recent Soot revisions, the 75% savings introduced by Reviser accumulate to over 11,000 seconds (3 hours).

In general, the performance of Reviser depends on the number of jump functions that need to be recomputed. The larger the impact of the code change is, the more edges are affected. Clearing and repropagating over an edge takes generally longer than the initial computation for that edge. Performance gains are thus achieved by recomputing only (a sufficiently precise overapproximation of) the affected nodes which is usually a small subset of all nodes in the program graph. In the worst case, all edges must be recomputed and the algorithm degenerates to a recomputation with some overhead, caused by changeset computation and wrapper lookups as described in Section 4. This, for instance, happens for the JUnit update from version 4.10 to version 4.11.

Furthermore, Reviser is not applicable in cases in which the time required for changeset computation already exceeds the time a full recomputation would take, or, in other words, a recomputation is extremely fast which erases the need for incrementalization anyway. If such a situation is detected during the first incremental update, further revisions should rather be recomputed than incrementally updated.

In the situations we consider primary use cases for Reviser, changes to the source code are frequent, but reasonably small, e.g., a single bug fix or new feature checked into a version control system. In these cases, Reviser can save up to 80% of the computation time across different client programs and analyses as we have shown in our evaluation.

We have also experimented with a less precise, but faster changeset-computation algorithm that performs a string comparison on method bodies in an attempt to reduce this overhead. In the case of inequality, all statements in the old version are marked as removed and all statements in the new method are considered as added. Note that this quick differencing algorithm is equal to the full differencing algorithm if the target application has no code changes. For the example of JUnit, the roughly 11 seconds for the structural diff and the about 14 seconds required to remap equal statements to their new objects after reloading the program graph must always be invested which restricts the savings possible by using a more coarse-grained diff algorithm. On the other hand, the number of reprocessed edges was increased from about 34,000 to approximately 42,000. In total, we thus obtained timing results almost identical to those of the full diff algorithm, showing that Reviser's total computation time is rather stable even with different diff algorithms. Note that Reviser is completely oblivious to the concrete diff algorithm being used. Inside a development environment like Eclipse, which provides diff information to its build steps anyway, the boundary of when incremental updates become feasible and when it is better to recompute may shift.

7. RELATED WORK

A method for constructing incremental versions of kill-gen problems has been proposed by Pollock and Soffa [15]. Ryder [19] has shown an incremental solver based on linear equations for partitionable problems. These approaches however do not easily translate to arbitrary IDE problems. Carroll and Ryder [4] incrementalize arbitrary elimination-based data flow algorithms by reducing them to the domintator-tree problem and applying a variant of Reps' optimal attribute parse-tree update algorithm. In general, Ryder and Burke [3] have surveyed many older approached which are limited to lowering lattice values in IDE problems leading to results that differ from a complete recomputation. Reviser always produces the same results as a full recomputation.

Ismail [10] and Souter et al. [23] have presented methods for incrementally updating call graphs. This work is orthogonal to Reviser which focuses on the IDE results alone. At the moment, Reviser re-computes the complete call graph.

Sharp [21] has presented an approach for pre-computing summaries for type and dependence analysis in library functions which change rarely. Our approach is orthogonal as it focuses on identifying changes and propagating their effects across the code under analysis. Rountev et al. [18] discuss how to compute more concise summaries of libraries for generic IDE problems. Integrating these results into our incremental solver is subject to future work. We envision using summaries when calls into libraries are changed, but not the libraries themselves, so as not to repropagate the changes through the complete library.

Tripp et al. [24] efficiently update data-flow results by replacing the whole-program control-flow graph and pointer analysis with local information computed on demand during the taint propagation. This removes the need of recomputing these data structures when the target changes. For the data-flow facts as such, their tool Andromeda computes the

transitive closure of all facts influenced by the code change, removes them, and then starts a recomputation. While the latter is quite close to Reviser's clear-and-propagate approach, Reviser only deletes jump functions on demand and thus can abort earlier if it detects that a jump function would be recreated in exactly the same way, i.e., it does not delete functions that might not change at all just because they are in the change's transitive dependence set.

Eichberg et al. [5] use incremental tabled evaluation to update analysis results deduced in a logic language. This approach comes with the cost of having to translate the program graph to Prolog facts and depends on the performance of the underlying general-purpose reasoning engine.

8. CONCLUSION

We have presented Reviser, a novel approach for efficiently and incrementally updating analysis results to reflect changes to the analyzed code. Reviser's update algorithm is based on the clear-and-propagate principle. It detects the origin of a change, then clears the information of its successors and propagates the new results from there on. We have implemented Reviser on top of Heros and Soot. Using this implementation, we have shown that on the initial computation Reviser only introduces a small overhead (under 20% in most cases, sometimes even within the standard deviation), but greatly outperforms recomputation (saving up to 80% of the time) when performing incremental updates. Therefore, the initial overhead quickly pays off when updating IDE results on a regular basis, e.g., at every check-in on a continuous-integration server. Over the 25 most recent commits to the Soot GIT repository, the savings achieved with Reviser accumulated to over 11,000 seconds (3 hours). For future work, we envision integrating dynamic approaches for updating the points-to analysis results and the callgraph. Furthermore, we envision integrating precomputed summaries for rarely changed libraries.

Acknowledgements. We would like to thank Cheng Zhang, Marc-André Laverdière-Papineau and the anonymous reviewers of OOPSLA'13 for their very helpful comments on a first versions of this paper. This work was supported by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and by the DFG within the project RUNSECURE.

9. REFERENCES

[1] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *AOSD*, pages 87–98, 2005.

[2] Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *SOAP 2012*, pages 3–8, July 2012.

[3] M.G. Burke and B.G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transactions on Software Engineering (TSE)*, 16(7):723–728, 1990.

[4] M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute update. In *POPL '88*, pages 274–284, New York, NY, USA, 1988. ACM.

[5] Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and Klaus Ostermann. Automatic incrementalization of prolog based static analyses. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, volume 4354 of *LNCS*, pages 109–123. Springer Berlin Heidelberg, 2007.

[6] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *ISSTA '06*, pages 133–144, New York, NY, USA, 2006. ACM.

[7] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. Highly precise taint analysis for android applications. Technical Report TUD-CS-2013-0113, EC SPRIDE, May 2013. `http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf`.

[8] Erich Gamma and Kent Beck. JUnit: A Regression Testing Framework. http://www.junit.org, 2000.

[9] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable JavaScript. In *ISSTA '11*, pages 177–187, 2011.

[10] Usman Ismail. Incremental call graph construction for the Eclipse IDE. Technical Report CS-2009-07, University of Waterloo, 2009.

[11] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, October 2011.

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

[13] Nomair A. Naeem and Ondřej Lhoták. Efficient alias set analysis using SSA form. In *ISMM '09*, pages 79–88, 2009.

[14] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *USENIX Security Symposium 2013*, August 2013.

[15] Lori L. Pollock and Mary Lou Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering (TSE)*, 15(12):1537–1549, December 1989.

[16] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61, 1995.

[17] Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural shape analysis for cutpoint-free programs. In Chris Hankin and Igor Siveroni, editors, *Static Analysis*, volume 3672 of *LNCS*, pages 284–302. Springer Berlin / Heidelberg, 2005.

[18] Atanas Rountev, Mariana Sharp, and Guoqing Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In *Compiler Construction*, volume 4959 of *LNCS*, pages 53–68, 2008.

[19] Barbara G. Ryder. Incremental data flow analysis. In *POPL '83*, pages 167–176, New York, NY, USA, 1983. ACM.

[20] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1):131–170, 1996.

[21] M.L. Sharp. *Static analyses for Java in the presence of distributed components and large libraries*. PhD thesis, The Ohio State University, 2007.

[22] S. Shoham, E. Yahav, S.J. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering (TSE)*, 34(5):651–666, 2008.

[23] A.L. Souter and L.L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *ICSM '01*, pages 682 –691, 2001.

[24] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In Vittorio Cortellessa and Daniel Varro, editors, *Fundamental Approaches to Software Engineering*, volume 7793 of *LNCS*, pages 210–225. Springer Berlin Heidelberg, 2013.

[25] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *CASCON '99*, pages 125–135, 1999.

[26] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.