Refining Interprocedural Change-Impact Analysis using Equivalence Relations

Alex Gyori
University of Illinois, USA
gyori@illinois.edu

Shuvendu K. Lahiri
Microsoft Research, USA
shuvendu@microsoft.com

Nimrod Partush
Technion, Israel
nimi@cs.technion.ac.il

ABSTRACT

Change-impact analysis (CIA) is the task of determining the set of program elements impacted by a program change. Precise CIA has great potential to avoid expensive testing and code reviews for (parts of) changes that are refactorings (semantics-preserving). However most statement-level CIA techniques suffer from imprecision as they do not incorporate the semantics of the change.

We formalize change impact in terms of the trace semantics of two program versions. We show how to leverage equivalence relations to make dataflow-based CIA aware of the change semantics, thereby improving precision in the presence of semantics-preserving changes. We propose an *anytime* algorithm that applies costly equivalence-relation inference incrementally to refine the set of impacted statements. We implemented a prototype and evaluated it on 322 real-world changes from open-source projects and benchmark programs used by prior research. The evaluation results show an average 35% improvement in the number of impacted statements compared to prior dataflow-based techniques.

CCS CONCEPTS

•**Software and its engineering** →**Automated static analysis;** *Software verification;*

KEYWORDS

Impact Analysis, Software Maintenance, Equivalence

ACM Reference format:
Alex Gyori, Shuvendu K. Lahiri, and Nimrod Partush. 2017. Refining Interprocedural Change-Impact Analysis using Equivalence Relations. In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 11 pages.
DOI: 10.1145/3092703.3092719

1 INTRODUCTION

Software constantly evolves to add and improve features, eliminate bugs, improve design, etc. As software evolves faster than ever, it requires rigorous techniques to ensure that changes do not modify existing behavior in unintended ways. Some of the emerging approaches to ensure the quality of a change are code reviews [35], regression testing [19, 44], test-suite augmentation [34, 39, 40], code contracts [5, 25], regression verification [20, 38] and verification modulo versions [33]; all benefit from change-impact analysis (CIA).

Change-Impact Analysis determines the set of program elements that may be impacted by a syntactic change. Traditional approaches are coarse-grained and operate at the level of types and classes [1, 2], or files [19] to retain soundness. Fine-grained techniques that aim to work at the level of statements are typically based on performing dataflow analysis [43] on one program to propagate the change along data and control flow edges [3, 10, 32]. Such techniques fail to take the *semantics of the change* into account; therefore, they cannot distinguish between changes that a user expects to have only local impact on existing code (e.g., a code refactoring) from ones that have substantial impact on existing code (e.g., changing the functionality or fixing a bug). The ability to distinguish changes whose impact is local (limited to the changed procedure or a few callers or callees within one or two levels) can help with code review and regression-testing efforts. Changes with substantial impact can be prioritized for more rigorous code reviews and more testing.

In this paper, we aim to improve the precision of CIA by leveraging *equivalence relations* between the variables of two programs across a change. At a high level, these equivalences help prune the flow of a change along the data or control flow edges of the changed program. To integrate such equivalences, we first formalize the notion of change impact precisely in terms of the trace semantics of two programs. Next, we show how to make CIA change-semantics aware by incorporating various equivalence relations into an interprocedural dataflow analysis. Since computing equivalence relations is expensive, we propose an *anytime algorithm* [46, 48] to incrementally compute equivalence relations.

1.1 Overview

Figure 1 shows a running example in C. The example is inspired by real commits to `Coreutils`, in files `paste.c` [13] and `sort.c` [14]. The program has three changes. Two are semantics-preserving: (i) extracting the literal `'\n'` into the variable `line_delim` in the procedure `print_product_info` (lines 1, 4, 5) and (ii) replacing the conditional operator with a double negation in `locale_ok` (lines 22, 23)[1]. The third change is not semantics-preserving: it sets the `line_delim` variable to `'\0'` (a different value than in the old version) in the procedure `print_product_info` (lines 12, 13, 14), which impacts statements in `print_minor_vers`. We claim the only (syntactically unchanged) line that is *impacted by the changes* is the highlighted line 41 (assume for this example that all executions start from the procedure `print_product_info`); a statement is impacted, intuitively, if the *sequence of values* it reads can differ when executing the two versions of the program in the same environment. For brevity, we omit the definitions of the

[1]Negation in C coerces the values to 0 or 1.

DOI: 10.1145/3092703.3092719

```
1  +static unsigned char line_delim='\n';
2   int print_product_info(int name, int version) {
3     int locale, prin=0;
4  -  print_header('\n');
5  +  print_header(line_delim); // spurious impact
6     locale=locale_ok(); // spurious impact
7     if (name) {
8       prin=print_name(locale);
9     }
10    if (version && prin) {
11      prin=print_major_vers(locale));
12 -    prin=print_minor_vers(locale,'\n');
13 +    line_delim='\0';
14 +    prin=print_minor_vers(locale,line_delim));
15    }
16    return prin;
17  }
18  void print_header(char delim) {
19    printf("%s%c",HEADER,delim);
20  }
21  int locale_ok() {
22 -  return setlocale (LC_ALL,"") ? 1 : 0;
23 +  return !!setlocale (LC_ALL,"");
24  }
25  int print_name(int locale) {
26    if (locale) {
27      printf("%s",locale_format("Coreutils"));
28      return 1;
29    }
30  return 0;
31 }
32  int print_major_vers(int locale) {
33    if (locale) {
34      printf("%s",locale_format("8"));
35      return 1;
36    }
37    return 0;
38  }
39  int print_minor_vers(int locale, char delim) {
40    if (locale) {
41     printf("%s%c",locale_format(".12"),delim);
42      return 1;
43    }
44    return 0;
45  }
```

Figure 1: Example program change. The lines with − and + represent deleted and added lines, respectively.

`setlocale` and `locale_format` procedures and the `LC_ALL` and `HEADER` constants as they are not relevant. We will now analyze the change through the lens of a standard dataflow analysis [43] and traditional equivalence checking [20, 28] and sketch our technique.

Dataflow: A dataflow analysis technique starts at the sources of change and propagates them through data and control edges (typically in the changed program). Dataflow techniques are not aware of the change semantics, and thus cannot exploit semantics-preserving changes. Initially, the call to `print_header` on line 5 has a change to its argument that marks all the statements in the procedure as impacted because they all depend on the changed argument. Next, the call to `locale_ok` on line 6 impacts the `locale` variable because of the change to the body of `locale_ok` and the data dependency of the return value on the change. This in turn will mark the input of `print_name` as impacted at line 8, which in turn flows to its output because the return value is control dependent on the input variable marked as impacted (a context-insensitive analysis will impact the return at *all* call sites to `print_name`). This impact through the return value will propagate to the call to `print_major_vers` and `print_minor_vers` because of the control dependency on `prin` and will impact all the statements in these procedures as well as all the returns at both call sites. Finally, the call to `print_minor_vers` will impact all of the callee statements. A context-sensitive analysis does not help either because the body of `locale_ok` changes, which implies that the return value may change across the two versions. This is sound but *imprecise* since the analysis is unable to determine that the statements in `print_name` and `print_major_vers` are not impacted.

Equivalence: A traditional interprocedural equivalence checking [20, 28] (checking if two procedures have identical input-output behavior) will find that `locale_ok`, `print_name`, `print_header`, `print_major_vers`, and `print_minor_vers` have identical summaries. This is *unsound* for the question of impact analysis, as the statement of `print_minor_vers` is impacted due to the change of print delimiter. This illustrates the difference between CIA and (traditional) equivalence checking: two procedures can be equivalent, but still impacted, because they may get called under different contexts and exhibit different behaviors.

Our approach: Our *change-semantics aware* CIA works as follows: it infers equivalence relations over variables and determines that the arguments at all call sites to `print_name` and `print_major_vers` are equal in both versions and stops propagating impacts through their arguments. Further, `locale_ok` has an equivalent summary in the two versions (by using equivalence checking)—this ensures that two call sites with equal arguments return equal results. From these two facts, the technique infers (by simple dataflow analysis) that arguments to `print_name` and `print_major_vers` are not impacted and therefore the statements in both `print_name` and `print_major_vers` are not impacted. Thus, our approach precisely identifies the only unchanged impacted line as line 41.

1.2 Contributions

In this work, we make the following contributions:

(1) We precisely formalize the set of statements *impacted* by a change, in terms of the trace semantics of two versions of a program (§ 3.1).
(2) We make a dataflow-based CIA change-semantics aware by incorporating various equivalence relations (§ 4).
(3) We describe an anytime algorithm that allows incrementally computing more equivalences to refine the analysis at the expense of time (§ 4.1).
(4) We have implemented a prototype using SymDiff [28, 29], and evaluated our technique on 322 real-world changes collected from GitHub open-source projects and several standard benchmark programs used in prior research [24].

2 BACKGROUND

For the ease of presentation, we will formalize the problem and our technique over a simple language. We can compile most features of general-purpose imperative programming languages to our simple language [4, 12, 18, 41]; we discuss this in § 2.2.

2.1 A Simple Language

A program consists of procedures represented as control-flow graphs, statements, and expressions.

Expressions: $e \in Exprs$ in the language are built up from constants, variables and operator applications:

$$e \in Exprs \quad ::= \quad c \mid x \mid y \mid \ldots \mid \mathbf{op}(e_1, \ldots, e_k)$$

Here c represents *constant* values of different types such as $\{\mathbf{true}, \mathbf{false}\}$ for Booleans, $\{\ldots, -1, 0, 1, \ldots\}$ for integers, and x denotes variables in scope. An operator **op** is a function or predicate symbol that can be uninterpreted or interpreted by some theories (e.g., $\{+, -, *, \leq, \geq, \ldots\}$ by the theory of arithmetic). We represent a vector of variables and expressions using $\overline{x}$ and $\overline{e}$, respectively.

Statements: $st \in Stmts$ are comprised of *assign*, *assume*, *skip* and procedure *call* statements.

$$\begin{aligned} st \in Stmts \quad ::= \quad & x := e \mid \mathbf{assume}\ e \mid \mathbf{skip} \mid \\ & \mathbf{call}\ x_1, x_2, \ldots, x_k := f(e_1, e_2, \ldots, e_m) \end{aligned}$$

The argument to **assume** is a Boolean-valued expression, and a **skip** is a no-op. A call statement can have multiple return values and they are assigned to variables x_i at the call site.

Procedures: A procedure $f \in Procs$ is represented as a control-flow graph consisting of $(N_f, E_f, In_f, Out_f, Vars_f, n_f^e, n_f^x)$, where:

- N_f is a set of control-flow locations in f,
- $E_f \subseteq N_f \times N_f$ is a set of edges over N_f denoting control-flow,
- In_f (respectively, Out_f) is the vector of *input* (respectively, *output*) formals of f. The output formals model return values and out parameters.
- $Vars_f$ is the set of variables in the scope of f and includes In_f, Out_f, and local variables of f,
- $n_f^e \in N_f$ (respectively, $n_f^x \in N_f$) is the unique *entry* (respectively, *exit*) node of f.

Let $N = \bigcup_{f \in Procs} N_f$ and $Vars = \bigcup_{f \in Procs} Vars_f$. Nodes and variables in a procedure f are often denoted by n_f and x_f respectively. For any node $n_f \in N_f$, we define the *readset* $RVars(n_f)$ and *write-set* $WVars(n_f)$ as the set of variables that are read and written to respectively in the statement at n_f.

A *program* $Prog \in Programs$ is a tuple $(Procs, main, StmtAt)$ where (i) $Procs$ is a set of procedures in the program, (ii) $main \in Procs$ is the entry procedure from which the program execution starts, and (iii) $StmtAt : N \to Stmts$ maps a node $n \in N$ in a procedure f to a *statement*. For any f, we assume that both $StmtAt(n_f^e) = \mathbf{skip}$ and $StmtAt(n_f^x) = \mathbf{skip}$.

2.2 Expressiveness

We can compile most constructs in general-purpose imperative programming languages to our simple language. This follows the same principle as translators from languages such as C and Java to the Boogie language [4, 12, 18, 41].

Control flow: Loops can be automatically transformed into tail-recursive procedures [20, 28, 29]. We use $n_1 : st; \mathbf{goto}\ n_2, n_3;$ to express that $StmtAt(n_1) = st$ and $\{(n_1, n_2)(n_1, n_3)\} \subseteq E_f$. A conditional statement **if** (e) st_1 **else** st_2 is modeled as:

$$\begin{gathered} n_1 : x := e; \mathbf{goto}\ n_2, n_3; \\ n_2 : \mathbf{assume}\ x; st_1; \mathbf{goto}\ n_4; n_3 : \mathbf{assume}\ \neg x; st_2; \mathbf{goto}\ n_4; \end{gathered}$$

where a fresh Boolean variable x captures the value of the condition e[2]. We assume that each node $n \in N_f$ has at most two successor nodes in E_f corresponding to conditional statements branches. The only use of an **assume** statement is to model a conditional statement. We refer to n_1 as a *branching* node with two successors in E with complementary expressions in **assume** statements.

Globals and heap: Richer data types such as arrays and maps can be modeled, e.g., array read $x[e]$ is modeled using $sel(x, e)$ and a write $x[e_1] := e_2$ is modeled using $x := update(x, e_1, e_2)$ [6]. Arrays are in turn used to model the heap in imperative programs and are standard in most software verification tools [12, 18, 41]. Additional internal non-determinism (e.g. read from file, network) is lifted as reads from immutable input arrays of *main*, making programs deterministic in our language [28]. We desugar the program's global variables (including the heap) as additional input and output formal arguments to a procedure. We transform each procedure into its *Static Single Assignment* (SSA) form [17], where a variable is assigned at exactly one program node.

2.3 Semantics

Let $\mathcal{V}$ denote the set of values that variables and expressions can evaluate to. Let $\theta \in \Theta$ be a *store* mapping variables to values in $\mathcal{V}$. For $x \in Vars$, we define $x \in \theta$ if x is a variable in the domain of θ. For $x \in \theta$, $\theta(x)$ denotes the value of variable x. The store $[x \to v]$ represents a singleton store that maps x to v. The store $\theta|_{Vars_1}$ restricts the domain of the store to variables in $Vars_1$. For stores θ_1 and θ_2, the store $\theta_3 \doteq \theta_1 \oplus \theta_2$ is defined as follows for any variable $x \in \theta_1$ or $x \in \theta_2$:

$$\theta_3(x) = \begin{cases} \theta_2(x), & \text{if } x \in \theta_2 \\ \theta_1(x), & \text{otherwise} \end{cases}$$

The value of an expression $e \in Exprs$ $(\theta(e))$ is defined inductively on the structure of e (we omit it for brevity as it is fairly standard).

Calls: Let $cs \in (N \times Vars^* \times \Theta)^*$ be a *call stack* that is a sequence of tuples $\langle (n_0, \overline{r}_0, \theta_0), (n_1, \overline{r}_1, \theta_1), \ldots \rangle$, where n_i is the i-th call site on the call stack (n_0 is the most recent), $\overline{r}_i$ and θ_i, respectively, are the vector of return actuals and the valuation of the local variables of the caller, at the corresponding call site. Let CS denote the set of all such call stacks, ϵ denotes an empty stack, and $(n, \overline{r}, \theta) :: cs$ denotes the *concatenation* operator.

Transition Relation: A *state* $\sigma \in \Sigma$ is a tuple $(n, \theta, cs) \in N \times \Theta \times CS$ that denotes a point in program execution where n is the current node being executed in a procedure f, θ is the valuation of variables in $Vars_f$ and cs is the current call stack.

A *state transition* denoted as $(n_f, \theta_1, cs_1) \rightsquigarrow (n_2, \theta_2, cs_2)$ is a relation over $\Sigma \times \Sigma$ that holds only if:

[2]The introduction of x simplifies determining if control flow is impacted by only inspecting the conditional node

(1) $StmtAt(n_f) \doteq x := e$, $n_2 \in N_f$, $\theta_2 = \theta_1 \oplus [x \to \theta_1(e)]$, $(n_f, n_2) \in E_f$, and $cs_1 = cs_2$, or
(2) $StmtAt(n_f) \doteq$ **assume** e, $n_2 \in N_f$, $\theta_1(e) =$ **true**, $(n_f, n_2) \in E_f$, $\theta_1 = \theta_2$ and $cs_1 = cs_2$, or
(3) $StmtAt(n_f) \doteq$ **skip**, $n_f \neq n_f^x$, $n_2 \in N_f$, $(n_f, n_2) \in E_f$, $\theta_1 = \theta_2$ and $cs_1 = cs_2$, or
(4) $StmtAt(n_f) \doteq$ **call** $\overline{r} := g(\overline{e})$. Let n be the unique successor of n_f in f, and $\overline{x}$ be the vector of input formals for g in $n_2 = n_g^e$, $cs_2 = (n, \overline{r}, \theta_1) :: cs_1$ and $\theta_2 = [\overline{x} \to \theta_1(\overline{e})]$, or
(5) $StmtAt(n_f) \doteq$ **skip**, $n_f = n_f^x$, $cs_1 \doteq (n_g, \overline{r}, \theta_3) :: cs_3$. Let $\overline{y}$ be the vector of output formals for f in $n_2 = n_g$, $\theta_2 = (\theta_3 \oplus [\overline{r} \to \theta_1(\overline{y})])|_{Vars_g}$, $cs_2 = cs_3$.

A transitive edge $\sigma_0 \rightsquigarrow^* \sigma_n$ exists if $\sigma_n \equiv \sigma_0$ or there exists a sequence of transitions $\sigma_0 \rightsquigarrow \sigma_1, \ldots \sigma_{n-1} \rightsquigarrow \sigma_n$, where $\sigma_i \rightsquigarrow \sigma_{i+1}$, for all $i \in [0, \ldots, n)$. For a procedure f, we denote the input-output *transition relation* $\Omega_f \doteq \{(\theta_1, \theta_2) \mid (n_f^e, \theta_1, \epsilon) \rightsquigarrow^* (n_f^x, \theta_2, \epsilon)\}$.

Execution Traces: An *execution trace* τ is a (possibly infinite) sequence of states $\langle \sigma_0, \sigma_1, \ldots \rangle$, where $\sigma_i \rightsquigarrow \sigma_{i+1}$, for any adjacent pair of states in the sequence. For a trace τ and a node $n \in N$, $\tau|_n$ denotes the (maximal) subsequence of τ containing states of the form $(n, \_, \_)$. For such a trace τ of length at least $i + 1$, $\tau[i]$ denotes the state at position i (namely σ_i). For any procedure f, let Γ_f be the set of *maximal* traces of f. That is, Γ_f is the set of all traces τ such that (i) $\tau[0] \doteq (n_f^e, \_, \epsilon)$, and (ii) either (a) the final state σ_n has no successors, or (b) the trace is non-terminating. Traces with no successors can either terminate *normally* in a state $(n_f^x, \_, \epsilon)$, or could be *blocked* due to no successors in E or due to an unsatisfied **assume** statement. For a store $\theta \in \Theta$, we denote $\tau_f(\theta)$ as the maximal trace (due to determinism) of f that starts in a store θ.

3 PROBLEM STATEMENT

In this section, we formalize the problem of *semantic change-impact analysis* and provide a simple solution based on dataflow-based static analysis.

3.1 Representing Changes

We denote $Prog^1, Prog^2 \in Programs$ as two versions of a program. Similarly $\sigma^i, \theta^i, \tau^i, Procs^i, main^i, N^i, StmtAt^i$ denote entities for $Prog^i$, without making $Prog^i$ explicit.

To ease presentation we assume the two programs in a *normalized* form, where (i) each procedure in $Procs^1$ has a corresponding procedure in $Procs^2$ and vice versa, and (ii) for each $f \in Procs^i$, the vector of variables in $Vars_f$, and the set of nodes N_f (but not necessarily E_f) are identical with the ones in the corresponding procedure. We preprocess the programs to obtain their normalized form, by introducing additional procedures, variables (uninitialized) and nodes (for any missing node n, we add an unreachable node in N_f with a **skip** statement and empty successor list).

Differencing: Given the two versions, a differencing algorithm produces a mapping between nodes in the two programs. We assume we are given a sound diff algorithm to label the sources of change. A diff algorithm is sound if it produces a partial function $\pi : N^1 \nrightarrow N^2$ such that:

(1) π is a partial bijection[3] and $StmtAt(n_f) = StmtAt(\pi(n_f))$.
(2) π will map entry nodes n_f^e (and exit nodes n_f^x) in one procedure to entry nodes in the corresponding procedure (and exit nodes respectively).
(3) For any two traces $\tau^1 \doteq \tau^1_{main}(\theta)$ in $Prog^1$ and $\tau^2 \doteq \tau^2_{main}(\theta)$ in $Prog^2$, τ^1 only executes statements in $Dom(\pi)$ iff τ^2 only executes statements in $Im(\pi)$
(4) For any two traces $\tau^1 \doteq \tau^1_{main}(\theta)$ in $Prog^1$ and $\tau^2 \doteq \tau^2_{main}(\theta)$ in $Prog^2$, where τ^1 only executes statements in $Dom(\pi)$ or τ^2 only executes statements in $Im(\pi)$, then $\tau^1 = \tau^2$.

The mapped nodes MAPPED $\doteq Dom(\pi) \cup Im(\pi)$ underapproximate the set of nodes that are syntactically unchanged. Intuitively, if a program executes only statements in MAPPED then the program behaves the same in both versions; statements that are not in MAPPED are the sources of change.

We describe for illustrative purposes a simple differencing algorithm which is sound. The algorithm proceeds to produce a mapping π as follows: Let $Procs^{\Delta} \subseteq Procs$ be the set of procedures that have some syntactic change. Any node not in $f \in Procs^{\Delta}$ is trivially mapped as the control-flow graphs are identical in the two versions. Any node in $f \in Procs^{\Delta}$ is conservatively treated as not mapped. Our formulation is parameterized by a diff algorithm which can either be based on text [47] or more sophisticated notions such as abstract syntax trees [16] or program-dependency-graphs [27] as long as they satisfy the soundness criteria.

3.2 Semantic Change Impact

We can now state the meaning of a node being impacted by a program change, in terms of the trace semantics of the two programs and the set MAPPED.

For a sequence of states $\overline{\sigma}$ and a variable $x \in Vars$, $\overline{\sigma}\rfloor_x \in (\mathcal{V} \cup \{\bot\})^*$ denotes the sequence of values $\overline{v}$ with same length as $\overline{\sigma}$, and

$$v_i = \begin{cases} \theta(x), & \sigma_i \doteq (\_, \theta, \_) \text{ and } x \in \theta \\ \bot & \text{otherwise} \end{cases}$$

Definition 3.1 (Impacted nodes). Given $Prog^1, Prog^2$ and MAPPED, a node $n \in N^1 \cup N^2$ is *impacted* if either $Impacted(n, Prog^1, Prog^2, \pi)$ or $Impacted(\pi(n), Prog^2, Prog^1, \pi^{-1})$ holds, where π^{-1} is the inverse. N^i is the corresponding N for $Prog^i$.

We define $Impacted(k, Prog^a, Prog^b, \varpi)$:

(1) $k \notin Dom(\varpi)$, or
(2) there exists a store θ, pair of traces $\tau^a \doteq \tau^a_{main}(\theta)$ for $Prog^a$ and $\tau^b \doteq \tau^b_{main}(\theta)$ for $Prog^b$, and a variable $x \in RVars(n)$ such that $(\tau^a|_k)\rfloor_x \neq (\tau^b|_{\varpi(k)})\rfloor_x$.

We conservatively treat any unmapped node as impacted. A mapped node n is not impacted if the sequence of values of variables in $RVars(n)$ is identical for any two execution traces τ^a (in $Prog^a$) and τ^b (in $Prog^b$) starting from a common input store θ to *main*. For our low-level language, the $RVars(n)$ of a statement includes the state of the heap and address being written to. For example, the C# statement *x.length* = y is translated to $n : Length := update(Length, x, y)$, ($Length$ is an array representing the state of *length* field/attribute in all objects) with $RVars(n) = \{Length, x, y\}$.

[3] A partial bijection is a partial function that is injective when defined and (trivially) surjective when restricted to its image [21].

Table 1: Predicates used for dataflow analysis.

Predicate name	Definition
$\textsc{branchingNode}(n)$	if n is a branching node
$\textsc{ControlDependent}(n_2, n_1)$	if n_2 is *control-dependent* on n_1 [17]
$\textsc{Callsite}(n, f, g)$	if $StmtAt(n)$ is a call to f within a caller g.
$\textsc{InFormal}(x, i, f)$	if x is the i-th input formal of f
$\textsc{OutFormal}(x, i, f)$	if x is the i-th output formal of f
$\textsc{InActual}(e, i, f, n)$	if the expression e is the i-th actual argument to a call to f at a callsite n
$\textsc{OutActual}(r, i, f, n)$	if the variable r receives the i-th output formal to a call to f at a callsite n

3.3 Dataflow-Based Change-Impact Analysis

In this section, we describe *Dataflow-based Change-Impact Analysis* (DCIA), a *change semantics unaware* static analysis that provides a conservative estimate of the set of impacted nodes. The static analysis is an interprocedural dataflow analysis [43] that starts with a program $Prog^i$ ($i \in 1, 2$) and a conservative estimate of the syntactically-changed nodes, nodes not in $\textsc{mapped}$, and returns an upper bound on the set of (a) impacted nodes, (b) impacted variables, and (c) output variables whose summary may have changed.

Predicates: Table 1 defines some straightforward predicates used in the inference rules. The $\textsc{OutActual}(r, i, f, n)$ predicate holds when the i^{th} return value is assigned to variable r, at the call to f from the node n (note that we allow multiple return values); we call r the output actual to differentiate it from the i^{th} output formal inside the callee. For $\textsc{ControlDependent}(n_2, n_1)$, a node n_2 is control-dependent on node n_1 iff (i) there exists a path from n_1 to n_2 s.t. every node in the path other than n_1 and n_2 is *post-dominated* by n_2, and (ii) n_1 is not post-dominated by n_2 [17].

Dependency: Figure 2 describes a set of inference rules to compute two relations $\textsc{DependsOnVar}$ and $\textsc{DependsOnNode}$. For a pair of variables $x, y \in Vars_f$ such that y is either data- or control-dependent on x, then $\textsc{DependsOnVar}(y, x, f)$ holds. Similarly, a node $n \in N_f$ and a variable x that is updated at n, $\textsc{DependsOnNode}(x, n, f)$ holds. Subsequently, any variable y such that y is data or control dependent on such a variable x, then $\textsc{DependsOnNode}(y, n, f)$ holds. An inference rule (e.g. DEPENDS-NODE) lists a set of antecedents (above the line) and the consequent (below the line). Applying an inference rule results in adding a tuple to the relation in the consequent (e.g. $\textsc{DependsOnNode}$). The inference rules are applied repeatedly until a fix-point is reached.

Most of the inference rules are straightforward encoding of program data- and control flow. The rule CONTROL-DEPENDS expresses that if n_1 is a branching node, whose condition depends on x and y is written in a control-dependent node n_2, then y depends on x. The rule SUMMARY-DEPENDS captures the dependency of an actual return r on a variable w passed as an argument to f in a caller g, where w indirectly flows to r through a procedure call to f. For this callsite, the i-th output formal y (which is assigned to the output actual r) is dependent on the j-th input formal x, which in turn is assigned the actual e at the callsite.

DEPENDS-ENTRY
$$\frac{x \in In_f}{\textsc{DependsOnVar}(x, x, f)}$$

DEPENDS-WRITE
$$\frac{x \in RVars(n) \qquad y \in WVars(n) \qquad n \in N_f}{\textsc{DependsOnVar}(y, x, f)}$$

DEPENDS-TRANSITIVE
$$\frac{\textsc{DependsOnVar}(y, x, f) \qquad \textsc{DependsOnVar}(x, z, f)}{\textsc{DependsOnVar}(y, z, f)}$$

CONTROL-DEPENDS
$$\frac{\begin{array}{c}\textsc{branchingNode}(n_1)\\ x \in RVars(n_1) \qquad \textsc{ControlDependent}(n_2, n_1) \qquad y \in WVars(n_2)\end{array}}{\textsc{DependsOnVar}(y, x, f)}$$

SUMMARY-DEPENDS
$$\frac{\begin{array}{c}\textsc{Callsite}(n, f, g)\\ \textsc{OutActual}(r, i, f, n) \qquad \textsc{OutFormal}(y, i, f) \qquad \textsc{InFormal}(x, j, f)\\ \textsc{DependsOnVar}(y, x, f) \qquad \textsc{InActual}(e, j, f, n) \qquad w \in RVars(e)\end{array}}{\textsc{DependsOnVar}(r, w, g)}$$

DEPENDS-NODE
$$\frac{x \in WVars(n) \qquad n \in N_f}{\textsc{DependsOnNode}(x, n, f)}$$

DEPENDS-NODE-TRANSITIVE
$$\frac{\textsc{DependsOnNode}(x, n, f) \qquad \textsc{DependsOnVar}(y, x, f)}{\textsc{DependsOnNode}(y, n, f)}$$

Figure 2: Inference rules for computing $\textsc{DependsOnVar}$ and $\textsc{DependsOnNode}$. The input is a program *Prog*.

Impact Analysis: Figure 3 describes a set of inference rules to compute the set of nodes that are impacted in either program. For now, we ignore the highlighted antecedents (we use them in § 4 where we describe how we incorporate change semantics). The rules take as input a program (either $Prog^1$ or $Prog^2$), the set of mapped nodes $\textsc{mapped}$, and precomputed relations $\textsc{DependsOnNode}$ and $\textsc{DependsOnVar}$ for the particular program. They produce the relations $\textsc{impactedNode}$, $\textsc{impactedVar}$, and $\textsc{ImpactedSumm}$ that are an upper bound on the set of impacted nodes, variables, and variable summaries, respectively. Next we explain the rules using our illustrative example.

The SYNT-CHANGED rule represents the source of any change impact, stemming from syntactic changes to the program. The next rules, prefixed by VAR-2 and NODE-2, propagate impact from variables to expressions or program nodes, and vice versa.

The next two rules propagate change impact for an output y of a procedure f, expressed by the $\textsc{ImpactedSumm}(y, f)$ predicate. For an output formal $y \in Out_f$, the summary (input-output dependency) may change either when (i) y depends on a variable updated at an unmapped node $n \in N_f$ (expressed by IMPACT-SUMMARY), or (ii) y depends on the return of a procedure g with an impacted summary (expressed by IMPACT-SUMMARY-PROP).

Next, the CALL-IMPACT rule says that an input formal x in f can be impacted if the corresponding actual argument e at a callsite is impacted, representing downward flowing impact where a caller impacting a callee. Alternatively, the RETURN-IMPACT rule considers the case when the variable summary for the corresponding output formal y is impacted, representing upward flowing impact.

$$\text{SYNT-CHANGED}\ \frac{n \notin \textsc{mapped}}{\textsc{impactedNode}(n)} \qquad \text{NODE-2-VAR}\ \frac{\textsc{impactedNode}(n) \quad x \in \mathit{WVars}(n)}{\textsc{impactedVar}(x)} \qquad \text{VAR-2-EXPR}\ \frac{\textsc{impactedVar}(x) \quad x \in \mathit{RVars}(e)}{\textsc{ImpactedExpr}(e)} \qquad \text{VAR-2-NODE}\ \frac{\textsc{impactedVar}(x) \quad x \in \mathit{RVars}(n)}{\textsc{impactedNode}(n)}$$

$$\text{IMPACT-SUMMARY}\ \frac{\textsc{OutFormal}(y, i, f) \quad \textsc{DependsOnNode}(y, n, f) \quad n \notin \textsc{mapped} \quad \neg\textsc{SummaryEquiv}(y, f)}{\textsc{ImpactedSumm}(y, f)}$$

$$\text{IMPACT-SUMMARY-PROP}\ \frac{\begin{array}{c}\textsc{OutFormal}(y, i, f) \\ \textsc{Callsite}(n, g, f) \quad \textsc{OutFormal}(x, j, g) \quad \textsc{ImpactedSumm}(x, g) \quad \textsc{OutActual}(w, j, g, n) \quad \textsc{DependsOnVar}(y, w, f) \quad \neg\textsc{SummaryEquiv}(y, f)\end{array}}{\textsc{ImpactedSumm}(y, f)}$$

$$\text{CALL-IMPACT}\ \frac{\textsc{Callsite}(n, f, g) \quad \textsc{InActual}(e, i, f, n) \quad \textsc{ImpactedExpr}(e) \quad \textsc{InFormal}(x, i, f) \quad \neg\textsc{PreEquiv}(x, f)}{\textsc{impactedVar}(x)}$$

$$\text{RETURN-IMPACT}\ \frac{\textsc{Callsite}(n, f, g) \quad \textsc{OutActual}(r, i, f, n) \quad \textsc{OutFormal}(y, i, f) \quad \textsc{ImpactedSumm}(y, f)}{\textsc{impactedVar}(r)}$$

$$\text{SUMMARY-IMPACT}\ \frac{\begin{array}{c}\textsc{Callsite}(n, f, g) \quad \textsc{OutActual}(r, i, f, n) \quad \textsc{OutFormal}(y, i, f) \\ \textsc{InFormal}(x, j, f) \quad \textsc{DependsOnVar}(y, x, f) \quad \textsc{InActual}(e, j, f, n) \quad \textsc{ImpactedExpr}(e) \quad \neg(\textsc{PreEquiv}(x, f) \land \textsc{SummaryEquiv}(y, f))\end{array}}{\textsc{impactedVar}(r)}$$

Figure 3: Inference rules for dataflow based change-impact analysis. The highlighted antecedents are relevant for change-semantics aware analysis.

Finally, SUMMARY-IMPACT considers impact which propagates *through* a callee. Here, g calls f with an impacted actual e for the formal input x of f. Since the formal output y of f depends on aforementioned impacted x, the impact flows back outwards into the output actual r in g.

Our analysis preserves context-sensitivity as it does not impact a return value simply because the corresponding output formal is impacted in some context.

The algorithm `DCIA` does the following:

(1) Takes as input $Prog^1, Prog^2$ and $\textsc{mapped}$.
(2) Applies the inference rules in Figure 3 on $Prog^i$ to generate $\textsc{impactedNode}^i$, $\textsc{impactedVar}^i$, $\textsc{ImpactedSumm}^i$ until a fix-point is reached.
(3) Returns the tuple $(\bigcup_i \textsc{impactedNode}^i, \bigcup_i \textsc{impactedVar}^i, \bigcup_i \textsc{ImpactedSumm}^i)$.

The following theorem states the soundness of the dataflow analysis `DCIA`.

Theorem 3.2 (Soundness). *Given two programs $Prog^1, Prog^2 \in$ Programs and $\textsc{mapped} \subseteq N$, (a)* `DCIA` *terminates, and (b) for any $n \notin \textsc{impactedNode}$, n is not an impacted node with respect to* $\textsc{mapped}$ *(according to Definition 3.1).*

Consider for example the changes in Figure 1 at line 22; the procedure `locale_ok` has an impacted summary because its return variable depends on a node that is syntactically changed, i.e., is not in $\textsc{mapped}$. This causes the line 6 and the variable `locale` to be marked as impacted because of the rule IMPACT-SUMMARY. Impacts are propagated interprocedurally by the rule CALL-IMPACT to all calls that take `locale` as an argument, i.e., `print_name`, `print_major_vers`, and `print_minor_vers`. Similarly, using the same rule, the body of `print_header` is impacted by the changed argument `'\n'` changed to the variable `line_delim` on line 4. The propagation through calls further impacts their entire body because of the data and control dependency on the impacted argument (by the rules NODE-2-VAR and VAR-2-NODE which propagate impact through both control- and data-dependency relying on the predicate $\textsc{DependsOnVar}$).

4 INCORPORATING CHANGE SEMANTICS

In this section, we make the `DCIA` algorithm *change-semantics aware*. In other words, the analysis takes into account also the exact semantics of the change, in addition to the set of nodes $\textsc{mapped}$ that may have been syntactically changed. We inject the change-semantics by leveraging equivalence relationships between variables and procedure summaries in the two programs $Prog^1$ and $Prog^2$.

Let us define the following semantic equivalences for a variable over $Prog^1$ and $Prog^2$.

Definition 4.1 ($\textsc{PreEquiv}$). $\textsc{PreEquiv}(x, f)$ holds for an input formal $x \in In_f$ if for all stores θ, and for every pair of traces $\tau^1 \doteq \tau^1_{main}(\theta)$ and $\tau^2 \doteq \tau^2_{main}(\theta)$, $(\tau^1|_{n^e_f})\rfloor_x = (\tau^2|_{\pi(n^e_f)})\rfloor_x$.

Intuitively, $\textsc{PreEquiv}(x, f)$ holds for an input formal x of f if any two executions starting from *main* on the same input θ call f with the same sequence of values of x. For the example in Figure 1 the equivalences that hold are $\textsc{PreEquiv}(delim, print\_header)$, $\textsc{PreEquiv}(locale, print\_name)$, $\textsc{PreEquiv}(locale, print\_major\_vers)$, and $\textsc{PreEquiv}(locale, print\_minor\_vers)$. In contrast, the equivalence $\textsc{PreEquiv}(delim, print\_minor\_vers)$ does *not* hold, because of different values for *delim* `'\n'` and `'\0'` respectively, at the callsite in $print\_product\_info$.

We define $Deps(y)$ as the set of variables x in either $Prog^1$ or $Prog^2$ such that $\textsc{DependsOnVar}(y, x, f)$. For two stores θ_1 and θ_2

defined over same set of variables, we denote $\theta_1 =_{Vars_1} \theta_2$ to mean $\theta_1(x) = \theta_2(x)$ for every $x \in Vars_1$.

Definition 4.2 (SummaryEquiv). SummaryEquiv(y, f) holds for an output formal $y \in Out_f$ if $(\theta_1, \theta_2) \in \Omega_f$ in $Prog^i$ and $\theta_1 =_{Deps(y)} \theta_3$, then $(\theta_3, \theta_4) \in \Omega_f$ is in $Prog^j$ $(j \neq i)$ and $\theta_2(y) = \theta_4(y)$.

Intuitively, if the versions of f are executed from stores θ_1 and θ_3 where $\theta_1 =_{Deps(y)} \theta_3$, then either both procedures do not terminate, or the value of y after executing f is identical on exit. In Figure 1, all procedures are equivalent except `print_product_info`, i.e., in this case SummaryEquiv(*line_delim*, *print_product_info*) does not hold since in one version the value of *line_delim* at the end of the execution is "\0" while in the other it is undefined.

Figure 3 with the highlighted parts provides a refinement to the dataflow analysis to incorporate change semantics. In addition to the mapped, the algorithm now takes as input pre-computed relations PreEquiv and SummaryEquiv. In this section, we assume an oracle that provides these relations; we provide one implementation later (§ 5.1). The highlighted facts strengthen the antecedent of a rule and prevent it from being applicable in some contexts. For example, the strengthened CALL-IMPACT prevents an input formal x from being impacted if PreEquiv(x, f) holds. Similarly, the strengthened IMPACT-SUMMARY prevents a summary for y from impact if we know that SummaryEquiv(y, f) holds. The strengthened SUMMARY-IMPACT is now applicable only when either (i) the formal x does not satisfy PreEquiv or (ii) the summary for y does not satisfy SummaryEquiv.

We denote the new change-semantics aware algorithm as *Semantic Dataflow-based Changed Impact Analysis* (`SEM-DCIA`).

Theorem 4.3 (Soundness). *Given two programs $Prog^1, Prog^2 \in Programs$, mapped, PreEquiv, and SummaryEquiv, (i)* `SEM-DCIA` *terminates, and (ii) for any $n \notin$ impactedNode, n is not an impacted node with respect to mapped (from Definition 3.1).*

4.1 Anytime Algorithm

The `SEM-DCIA` algorithm assumes an oracle to compute the PreEquiv and SummaryEquiv relations. Computing such equalities typically require constructing the product of the two programs $Prog^1$ and $Prog^2$ and inferring equivalence relations over the product program [29]. Such inference algorithms typically have high complexity and therefore it is wise to apply them prudently. In this section, we make a simple observation that allows us to interleave `SEM-DCIA` and inference of PreEquiv and SummaryEquiv in a single framework.

```
void main(int x) {          void f2(int x) {
- f1(x);                       f3(x+2);
+ f1(x+0);                  }
}                           ...
void f1(int x) {            void fn(int x) {
  f2(x+1);                  }
}
```

Figure 4: Motivating example for anytime algorithm.

To exploit the change semantics, it is often useful to apply equivalence relation inference only in the vicinity of actual syntactic changes. Consider the example in Figure 4 to make the intuition clear. Applying `DCIA` will result in impacting all the nodes in the program as follows. The modified call node for f_1 in *main* is not in mapped, which impacts input formal x of f_1. This in turn impacts the call to f_2 and so on. We can observe that PreEquiv and SummaryEquiv hold for each of the procedures because the change does not propagate outside the changed statement.

For Figure 4 it suffices to infer the equivalences on *main* while *abstracting* the rest of the procedures from the expensive equivalence analysis. Considering f_1 has all callsites inside *main* and that it does not have an impacted summary by rule IMPACT-SUMMARY after `DCIA` suffices to determine that PreEquiv(x, f_1) holds. This information can be fed to `SEM-DCIA` which will prune the impact for the input parameter of f_1 which will prune the remaining impacts when performing a pure dataflow analysis. Thus, we obtain a precise change-impact analysis by applying the equivalence inference only on a small subset of the procedures in the program. Similarly, in Figure 1 it suffices to analyze only the syntactically changed procedures and abstract away the others to obtain the most precise result; this is not the case in general because to infer the PreEquiv we need all call sites to be in scope, not only the syntactically changed procedures.

Algorithm 1: SEM-DCIA-ANYTIME

Input: $Prog^1, Prog^2 \in Programs$
Input: $Procs^{\Delta} \subseteq Procs$
Input: mapped $\subseteq N$
Output: $impNds \subseteq N$
begin
  $k \leftarrow 0$;
  $EQ \leftarrow (\emptyset, \emptyset)$;
  $(impNds, impVars, impSumms) \leftarrow$ `SEM-DCIA`$(Prog^1, Prog^2,$ mapped$, EQ)$;
  $Procs' \leftarrow Procs^{\Delta}$;
  while $Procs' \subset Procs$ **do**
    $prEQ \leftarrow \{(x, f) \mid x \in In_f$ and $x \notin impVars\}$;
    $smEQ \leftarrow \{(x, f) \mid x \in Out_f$ and $(x, f) \notin impSumms\}$;
    $EQ \leftarrow EQ + (prEQ, smEQ)$;
    $Procs' \leftarrow$ `ProcsWithin`$(Procs^{\Delta}, Prog^1, Prog^2, k)$;
    $Prog^1_k \leftarrow$ `AbstractProcs`$(Prog^1, Procs \setminus Procs')$;
    $Prog^2_k \leftarrow$ `AbstractProcs`$(Prog^2, Procs \setminus Procs')$;
    $EQ \leftarrow$ `InferEquivs`$(Prog^1_k, Prog^2_k, EQ)$;
    $(impNds, impVars, impSumms) \leftarrow$ `SEM-DCIA`$(Prog^1, Prog^2,$ mapped$, EQ)$;
    $k{+}{+}$;
  return $impNds$;

Algorithm 1 (`SEM-DCIA-ANYTIME`) provides an *anytime* algorithm that performs the integration. The algorithm takes as an additional input $Procs^{\Delta}$, the set of syntactically changed procedures. It outputs a set of nodes *impNds* that overapproximates the set of impacted nodes. We term the algorithm anytime [15, 46, 48] because the algorithm can be stopped at any time after the first call to `SEM-DCIA` to obtain a conservative bound for the impacted nodes.

The algorithm starts with invoking `SEM-DCIA` on the two programs with an empty set of equivalences in *EQ* (line 4); this is identical to calling `DCIA`. The return values provide a conservative measure on impacted variables, nodes and summaries respectively (Theorem 3.2). The algorithm implements a loop (line 6) where it increases the frontier of procedures $Procs'$ around $Procs^{\Delta}$ that are analyzed for inferring equivalences in `InferEquivs` (line 13). Lines 7 and 8 construct equivalences from the provably non-impacted variables and summaries. These equivalences are added to *EQ* in line 9. `ProcsWithin` returns all procedures that can reach or be reached from $Procs^{\Delta}$ within a call stack of depth k; k is incremented with each iteration of the loop. `AbstractProcs` abstracts all procedures outside $Procs'$; it only retains the knowledge of whether any procedure $f \in Procs'$ has additional call sites outside $Procs'$ - this determines whether PreEquiv can be inferred for a procedure. `InferEquivs` is invoked with a set of equivalences in *EQ* on the smaller programs $Prog^i_k$. The final call to `SEM-DCIA` is used to compute the more refined set of impacted variables, nodes and summaries based on the equivalences discovered from `InferEquivs`. The loop terminates when $Procs'$ consists of the entire program; at this point `InferEquivs` has already looked at the entire program and no new equivalences will be discovered in line 13.

Let us denote $\texttt{SEM-DCIA}_k$ as an instantiation of the algorithm `SEM-DCIA-ANYTIME` that terminated after the loop is executed exactly $k+1$ times. We also denote $\texttt{SEM-DCIA}_\infty$ if the loop terminates normally after $Procs'$ equals *Procs*.

Theorem 4.4 (Soundness). *Given two programs* $Prog^1, Prog^2 \in$ *Programs,* mapped*, and* $Procs^{\Delta}$*, if* $\texttt{SEM-DCIA}_k$ *terminates then for any* $n \notin impNds$*, n is not an impacted node with respect to* mapped *(according to Definition 3.1).*

5 IMPLEMENTATION AND EVALUATION

5.1 Implementation

We presented and evaluated our `SEM-DCIA()` analysis for C programs, but our analysis is implemented over the intermediate verification language Boogie [4]. We leverage SMACK [41] to convert LLVM bytecode to Boogie programs.

Differencing: For our initial implementation, we leveraged `diff` over C files to produce the source of changes, i.e., nodes not in mapped. However, `diff` does not satisfy the soundness criteria for diff (see Section 3.1) because of changes in macros, data structures, control-flow changes, etc.; we therefore conservatively consider all nodes in a changed procedure as sources of impacts. Note that because we operate on Boogie, macros are already expanded so changes in macros will be reflected in the resulting Boogie code. Although this can overapproximate the initial source of impact, the use of equivalences in `SEM-DCIA` allows us to prune the spurious impacts from escaping the syntactically-changed procedures; All our code and scripts are available in the SymDiff repository at: https://symdiff.codeplex.com/.

Inference: We used SymDiff to construct a product program and infer valid PreEquiv and SummaryEquiv. Given $Prog^1$ and $Prog^2$, SymDiff generates a product program $Prog^{1\times2}$ that defines a procedure $f^{1\times2}$ for every f and $\pi(f) \in Procs^i$. For the product program $Prog^{1\times2}$, one can leverage any of the (single program) invariant generation techniques to infer preconditions, postconditions (including two-state postconditions) on $f^{1\times2}$. Such invariants are *relational* in that they are over the state of two programs $Prog^1$ and $Prog^2$, and include equivalences relations such as PreEquiv (preconditions of $f^{1\times2}$) and SummaryEquiv (summary of $f^{1\times2}$). To ensure our inferred equivalences are valid we require the programs to be equi-terminating [23]; this is an area of future work – for now we assume that changes do not introduce non-termination. We modified SymDiff to add candidates for inferring summaries and take as input cheaply-inferred equalities from `DCIA`. More details can be found in our extended report [22].

Table 2: Summary of projects used as evaluation subjects

Project Name	# Version Pairs	SLOC min	SLOC max	LOC Changed min	LOC Changed max
flingfd	2	142	146	2	14
histo	8	617	624	1	6
mdp	91	135	1616	1	402
theft	2	1672	1838	2	328
tinyvm	61	425	903	1	328
print_tokens	5	478	480	1	8
print_tokens2	10	397	402	1	6
replace	32	509	516	1	15
schedule	9	290	294	2	4
space	38	6180	6205	1	42
tcas	41	136	140	2	16
tot_info	23	346	347	2	3

5.2 Evaluation

In this section we evaluate the effectiveness of our approach on GitHub projects with real program changes and standard benchmark programs with artificial changes. We show that our semantic based analysis, `SEM-DCIA` improves on `DCIA` by reducing the size of the impacted set, a proxy metric for the effort necessary to perform many software engineering tasks such as code review and testing.

We analyze 164 changes consisting of refactorings, feature additions, buggy changes, and bug fixes from 5 GitHub projects. We selected the projects based on popularity, size, active development, and compatibility with SMACK. The projects, number of versions used, their size in non-comment non-blank source lines of code (SLOC), and corresponding change sizes (in number of C source lines changed) are summarized in Table 2. Our subjects are C implementations of a virtual machine program (tinyvm), a histogram creator (histo), a markdown presentation tool (mdp), a file-descriptor management library (flingfd) and a test-generation library (theft). We include 6 standard benchmarks widely used by prior research [24]. These benchmarks consist of 158 manually introduced changes representing non-trivial and hard to detect bugs. Our projects are sized between 142 lines of source code and 6205 (SLOC). The changes in our projects vary in size between very small changes, consisting of single line changes and larger ones, consisting of over 400 lines (most of our changes are small).

For our experiments, we first compare `SEM-DCIA` against `DCIA` to study the impact of adding change-semantics to the impact analysis (§ 5.3). Next, we evaluate the cost-precision tradeoff of the anytime algorithm `SEM-DCIA-ANYTIME` (§ 5.4). Finally, we present several representative examples discovered while applying our tool (§ 5.5).

Table 3: Analysis results for different levels of precision. Time in seconds. (timeout = 1 hour)

Project	DCIA			SEM-DCIA$_0$				SEM-DCIA$_1$				SEM-DCIA$_\infty$			
Name	min	max	Time	min	max	Red	Time	min	max	Red	Time	min	max	Red	Time
flingfd	64	84	0.94	39	83	20.1%	8.92	14	70	47.3%	9.85	14	70	47.3%	10.44
histo	0	86	2.14	0	75	11.5%	19.43	0	65	28.6%	20.59	0	65	28.6%	24.92
mdp	0	465	28.16	0	330	1.5%	77.71	0	324	3.4%	100.68	0	283	6.5%	173.08
tinyvm	0	344	68.96	0	308	18.5%	158.33	0	298	23.2%	160.35	0	283	43.6%	169.05
theft	184	261	4.48	11	186	61%	38.45	11	185	62%	57.48	11	107	77%	289.75
print_tokens	151	153	2.22	69	137	19.37%	24.02	34	128	28.67%	58.23	34	128	28.67%	102.40
print_tokens2	155	158	1.46	80	129	30.36%	16.08	59	101	44.65%	24.42	55	100	45.66%	97.98
replace	75	195	4.96	74	194	2.08%	35.72	70	194	2.89%	92.72	65	174	9.41%	236.77
schedule	79	115	1.37	7	104	26.35%	13.73	7	87	40.58%	24.15	7	68	70.85%	30.83
space	20	2851	59.45	14	2816	31.87%	798.96	14	2816	36.71%	895.14	*n.a.*	*n.a.*	*n.a.*	*timeout*
tcas	1	49	0.66	0	49	9.24%	7.94	0	49	9.24%	8.63	0	49	9.24%	9.71
tot_info	103	104	6.39	31	102	18.65%	37.01	24	102	46.26%	74.99	12	77	56.50%	127.61

5.3 Change-Semantic Aware Analysis

Table 3 shows the results of running our SEM-DCIA analysis on our subjects. For each change, we measure the number of lines impacted by dataflow analysis (columns DCIA Impact) and also by SEM-DCIA (columns SEM-DCIA$_\infty$). The columns SEM-DCIA$_i$ denote various bounds for SEM-DCIA-ANYTIME and are discussed in § 5.4. We report for each project the minimum and maximum number of impacted lines (min, max), and for the SEM-DCIA() analysis we report also the average reduction of the size of the impacted set. Note that SEM-DCIA analysis always reports a subset of the set reported by the non-semantic analysis. We also report the average analysis time in seconds for all analyses.

Our evaluation shows that on average, the semantic-aware analysis reduces the size of the impacted set by 35%. The overhead of performing full semantic analysis on the entire program is on median 19x, ranging between 3x and 67x. While the semantic analysis results at ∞ level represent the most precise analysis our technique achieves, it is quite expensive. For example in the theft project the reduction achieved by SEM-DCIA$_\infty$ is 77% but with a 64x overhead. This motivates the need for an incremental analysis, whose results are obtained faster.

Imprecision: Our manual inspection of results reveals three broad classes for nodes classified as impacted: (i) nodes in syntactically changed procedures, (ii) SymDiff's inability to match loops as it relies on syntactic position in AST (this can be fixed by better matching heuristics), (iii) SMACK represents all aliased addresses accessing a field using a single map; writing to one location destroys equivalences on the map variables (need more refined conditional equivalences [26]).

5.4 Incremental Analysis

Table 3 shows the analysis results of varying the bound on k for the SEM-DCIA-ANYTIME. The first iteration SEM-DCIA$_0$ corresponds to semantically analyzing only the syntactically-changed procedures; the second iteration SEM-DCIA$_1$ corresponds to analyzing the procedures at distance at most one from the syntactically changed procedures (callers and callees). The results show that even SEM-DCIA$_0$ provides benefits, pruning the impacted set by 22% on average. The overhead is reduced compared to the full analysis (9x). The results show that the reduction in impact improves as the analysis scope (k) increases. For example, in the case of theft the improvement is from 61% (SEM-DCIA$_0$) to 77% (SEM-DCIA$_\infty$), at the cost of overhead increase from 8x to 64x.

Table 4: Analysis results for space

Analysis	Min	Max	Reduction	Time
DCIA	20	2851	*n.a.*	59.45
SEM-DCIA$_0$	14	2816	31.87%	798.96
SEM-DCIA$_1$	14	2816	36.71%	895.14
SEM-DCIA$_2$	14	2816	40.56%	1300.43
SEM-DCIA$_3$	14	2808	43.96%	1900.03
SEM-DCIA$_\infty$	*n.a.*	*n.a.*	*n.a.*	*timeout*

We find that the anytime analysis is most beneficial for cases where it is prohibitive to run the full algorithm because of time constraints. This is best illustrated for the case of space (we used a timeout of one hour). Table 4 shows the first four levels for space (two more iteration beyond the ones in Table 3); performing the analysis incrementally is still valuable even upto $k = 3$; the first iteration already provides big benefits on top of the non-semantic analysis, while the following iterations display a smooth improvement with each iteration. We believe this highlights the benefits of our anytime algorithm, giving the user control over the tradeoff between precision and analysis-time.

5.5 Representative Examples

Our inspection of the analysis results indicates that the improvement in precision in SEM-DCIA() comes from two fronts. First, it compensates for the price we paid for soundness by considering entire procedures as source of impact. The semantic analysis reduces the impacts for callers and callees transitively. Second, the reduction in impact happens from refactorings that a pure dataflow analysis cannot consider. We next show a few interesting patterns we discovered while applying the tool (for brevity we only describe the change briefly).

Variable Extraction: Figure 5 shows a refactoring to extract a constant to a variable. A non-semantic technique will create impacts in term_move_to through the first argument, since it will not be able to find that the value flowing into the first argument is the same in both versions and in all executions. Our SEM-DCIA technique will successfully prove the mutual precondition necessary to show

```
void draw_histogram(int data[], int len) {
  ...
+ int xbarw = 5;
  ...
  while (y--) {
-   term_move_to(x * 5 + xpad + 3,
+   term_move_to(x * xbarw + xpad + 3,
        y - 1 + h + ypad);
      ...
    }
}
```

Figure 5: Change illustrating an extract constant to variable in histo commit *c723a4*

```
-while (*c) {
+for (;*c;c++) {
  ...
  wprintw(window, "%c",  *c);
- c++;
}
```

Figure 6: Change illustrating a loop conversion in mdp commit *00c2ad*

```
- if (!strend || !strbegin) goto pp_ret;
+ if (!strend || !strbegin) return 0;
  if (!pFile) {
    ...
-   goto pp_ret;
+   return 0;
  }
  ...
- pp_ret: return 0;
+ return 0;
```

Figure 7: Change illustrating a goto-elimination refactoring in tinyvm commit *378cc6*

the equality in both versions, and hence cut impacts that would propagate through the first argument.

Loop Refactoring: Figure 6 shows a change from a while loop to a for loop. Remember that we extract loops as tail recursive procedures. Input-output equivalence checking would not prevent the impact of the argument `c` to the callee inside the loop–the body of the loop–(nor would dataflow analysis).

Control-Flow Equivalence: Figure 7 shows a change to replace a `goto` with `return` statements. This is a change in the project tinyvm. The `goto` statements were all redirecting control-flow to a `return` statement, so the developer replaced the `goto` with the target `return` statement. Our semantic technique successfully finds that the change does not produce impacts.

6 RELATED WORK

Our work is closely related to work aiming to support developers in evolution tasks through change-impact analysis, regression verification, and symbolic analysis.

Change impact analysis: Change Impact Analysis has been widely explored in static and dynamic program analysis context [10, 30, 32, 42, 45]. Most previous works perform the analysis at a coarse-grain level (classes and types) to retain soundness of analysis [1, 2, 31, 36] which can result in coarse results. JDiff [1] addresses some of the challenges of performing both a diff and computing a mapping between two programs in the context of Java object-oriented programs. Other techniques resort to dynamic information to recover from the overly-conservative dataflow analysis [2, 36]. Our goal is to improve the precision of CIA analysis by making it change-semantics aware using statically computed equivalence relations without sacrificing soundness.

Regression verification: Regression verification [20, 39] and its implementations [28] aim at proving summary equivalence interprocedurally, but does not help with the CIA directly as shown in § 1.1. The work by Bakes et al. [3] improves traditional equivalence checking by finding paths not impacted by changes through symbolic execution. The approach is non-modular (does not summarize callees), bounded (unrolls loops and recursion), and does not seek to improve the underlying change-impact analysis. The technique leverages CIA to avoid performing equivalence checking on non-impacted procedures (computed by standard dataflow analysis). These approaches are useful for equivalence-preserving changes; when the changes are non-equivalent they do not provide meaningful help for reducing code review or testing efforts. Our approach, on the other hand, refines the CIA and can be used in code review and regression testing. Besides, our approach retains modularity and is sound in the presence of loops and recursion. We leverage the product construction in SymDiff [29] that has been used for *differential assertion checking* (checking if an assertion fails more often after a change); however this work is limited as it requires the presence of assertions in the program. Our approach can also use other product construction techniques and relational invariant inference techniques as an off-the-shelf solver [7, 8, 11].

Symbolic Analysis: Person et al. use change-directed symbolic execution to generate regression tests [40]. Our technique can be used to prune the space for which regression tests need to be generated. In addition, there is research on relational verification using a product construction [7–9, 37], but most approaches are not automated and do not consider changes across procedure calls.

7 CONCLUSIONS

In this work, we formalize and demonstrate how to leverage equivalence relations to improve the precision of dataflow-based change-impact analysis and provide a time-precision knob, which is crucial for applying such analyses to large projects. Our work brings together program verification techniques (namely relational-invariant generation) to improve the precision of a core software engineering task, and can go a long way in providing the benefits of semantic reasoning to average developers.

ACKNOWLEDGMENTS

We thank Darko Marinov, Sasa Misailovic, August Shi, and the anonymous reviewers for their comments. Alex performed parts of this research while at Microsoft Research. He was also partially supported by the NSF Grant Nos. CCF-1421503 and CNS-1646305.

REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 2–13. IEEE Computer Society, 2004.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th international conference on Software engineering*, pages 432–441. ACM, 2005.

[3] J. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries. In *Model Checking Software*, pages 99–116. Springer, 2013.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects (FMCO)*, pages 364–387, 2006.

[5] M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69. Springer, 2004.

[6] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. In *International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.

[7] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM 2011: Formal Methods*, pages 200–214. Springer, 2011.

[8] G. Barthe, J. M. Crespo, and C. Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In *Logical Foundations of Computer Science*, pages 29–43. Springer, 2013.

[9] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN Notices*, volume 39, pages 14–25. ACM, 2004.

[10] H. Cai and R. Santelices. A comprehensive study of the predictive accuracy of dynamic change-impact analysis. *Journal of Systems and Software*, 103:248–265, 2015.

[11] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 169–180, 2012.

[12] J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 302–314, 2009.

[13] Coreutils paste.c commit. https://github.com/coreutils/coreutils/commit/8297568ec60103d95a56cf142d534f215086fe2b.

[14] Coreutils sort.c commit. https://github.com/coreutils/coreutils/commit/611e7e02bff8898e622d6ad582a92f2de746b614.

[15] T. L. Dean and M. S. Boddy. An analysis of time-dependent planning. In *AAAI*, volume 88, pages 49–54, 1988.

[16] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[18] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 234–245, 2002.

[19] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 211–222. ACM, 2015.

[20] B. Godlin and O. Strichman. Regression verification. In *DAC*, pages 466–471, 2009.

[21] P. A. Grillet. *Semigroups: an introduction to the structure theory*, volume 193. CRC Press, 1995.

[22] A. Gyori, S. K. Lahiri, and N. Partush. Interprocedural semantic change-impact analysis using equivalence relations. In *Technical Report*. http://arxiv.org/abs/1609.08734, 2016.

[23] C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebelo. Towards modularly comparing programs using automated theorem provers. In *International Conference on Automated Deduction (CADE)*, pages 282–299. Springer, 2013.

[24] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200. IEEE Computer Society Press, 1994.

[25] J.-M. Jezequel and B. Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

[26] M. Kawaguchi, S. Lahiri, and H. Rebelo. Conditional equivalence. Technical report, Microsoft Research, October 2010.

[27] J. Krinke. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 301–309. IEEE, 2001.

[28] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification (CAV)*, pages 712–717, 2012.

[29] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 345–355, 2013.

[30] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 308–318. IEEE, 2003.

[31] W. Le and S. D. Pattison. Patch verification via multiversion interprocedural control flow graphs. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1047–1058. ACM, 2014.

[32] S. Lehnert. A review of software change impact analysis. *Ilmenau University of Technology, Tech. Rep*, 2011.

[33] F. Logozzo, S. Lahiri, M. Fahndrich, and S. Blackshear. Verification modulo versions: Towards usable verification. In *Proceedings of the 35th conference on Programming Languages, Design, and Implementation (PLDI 2014)*. ACM SIGPLAN, June 2014.

[34] P. D. Marinescu and C. Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 716–726. IEEE Press, 2012.

[35] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201. ACM, 2014.

[36] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 128–137. ACM, 2003.

[37] N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 238–258, 2013.

[38] F. Pastore, L. Mariani, A. E. Hyvärinen, G. Fedyukovich, N. Sharygina, S. Sehestedt, and A. Muhammad. Verification-aided regression testing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 37–48. ACM, 2014.

[39] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM.

[40] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 504–515, New York, NY, USA, 2011. ACM.

[41] Z. Rakamaric and M. Emmi. SMACK: decoupling source language details from verifier implementations. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 106–113, 2014.

[42] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices*, volume 39, pages 432–448. ACM, 2004.

[43] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61, 1995.

[44] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, Apr. 1997.

[45] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53. ACM, 2001.

[46] Wikipedia. Anytime algorithm. https://en.wikipedia.org/wiki/Anytime_algorithm.

[47] W. Yang. Identifying syntactic differences between two programs. *Software: Practice and Experience*, 21(7):739–755, 1991.

[48] S. Zilberstein and S. Russell. Approximate reasoning using anytime algorithms. *Kluwer International Series in Engineering and Computer Science*, pages 43–43, 1995.