Prioritizing Change-Impact Analysis via Semantic Program-Dependence Quantification

Haipeng Cai, Raul Santelices, and Siyuan Jiang

***Abstract*—Software is constantly changing. To ensure the quality of this process, when preparing to change a program, developers must first identify the main consequences and risks of modifying the program locations they intend to change. This activity is called change-impact analysis. However, existing impact analysis suffers from two major problems: coarse granularity and large size of the resulting impact sets. Finer-grained analyses such as slicing give more detailed impact sets which, however, are also even larger in size.While various impact-set reduction approaches have been proposed at different levels of granularity, the challenge persists as very-large impact sets are still produced, impeding the adoption of impact analysis due to the great costs of inspecting those impact sets.**

To address these challenges, we present a novel dynamic-analysis technique called SENSA which combines sensitivity analysis and execution differencing. SENSA not only provides fine-grained (statement-level) impact sets but also *prioritizes* potential impacts via *semantic-dependence quantification* for program slices. We evaluated the benefits of impact prioritization using SENSA with respect to static and dynamic forward slicing via an extensive empirical study of open-source Java applications and three case studies. Our results show that SENSA can offer much better cost-effectiveness than slicing in assisting developers with impact inspection and fault cause-effect understanding.

***Index Terms*—Sensitivity analysis, execution differencing, impact prediction, impact prioritization, dependence quantification**

ACRONYMS

DEA	differential execution analysis
BFS	breadth-first search
LOC	lines of code
SENSA-Inc	SENSA using incremental modifications
SENSA-Rand	SENSA using random modifications

NOTATION

`prog`	example program used for illustration
P	a generic program
T	test suite for a program
C	a set of statements (to be changed)
I	a program input
L	ranking given by a prioritization strategy
A	actual-impact ranking
N	number of statements in a ranking
NR	number of modified executions per test input

I. INTRODUCTION

SUCCESSFUL modern software requires constant changes. This evolution process [1], however, poses serious risks to the qualify and reliability of software [2]. Thus, there is a crucial need for techniques that provide automated and effective support to analyze the effects of those changes. *Change-impact analysis* is such a technique. It identifies potential consequences of software changes and estimates what else needs be modified to accomplish those changes [3].

Two typical usage scenarios of change-impact analysis exist: *predictive* impact analysis which is applied before potential changes are designed, and *descriptive* impact analysis which is applied after actual changes have been made. In this paper, we target *predictive* impact analysis, which helps developers assess risks and budget of, as well as design and plan for, their changes early [4], [5]. Importantly, for predictive change-impact analysis (referred to as *impact analysis* for brevity hereafter), the actual changes to be made are still unknown.

While impact analysis is widely recognized as a critical step during software development [6]–[8], many challenges have been identified by developers that hinder its practical use [9]. The two greatest such challenges are the coarse granularity (e.g., method-level) and large sizes of the impact sets produced. Statement-level techniques such as slicing [10] provide finer-grained results, which can alleviate the first challenge but tend to suffer more from the second: the resulting impact sets potentially contain even larger number of entities than those produced by techniques working at coarser levels. On the one hand, developers have to inspect the impact sets to understand *all* potential consequences of changes [7]. On the other hand, inspecting large impact sets can be prohibitively expensive for developers, who generally have limited resources for impact analysis in practice [9].

A large amount of research has aimed to reduce the impact-set sizes by improving the precision of impact analysis, both at the statement level (e.g., [11]–[13]) and coarser levels (e.g., [14]–[16]). However, the result-size problem persists as large impact sets are still produced frequently by these techniques. Moreover, impact-set reduction techniques can have additional drawbacks such as sacrificing scalability and risking the loss of safety of the analysis result.

To address these challenges, we adopt an approach that is different from, yet complementary to, existing impact-set reduction techniques. Instead of reducing impact-set sizes, we *prioritize* entities (statements) in impact sets so that impacts of higher priority can be inspected earlier than others, with the priority of a statement measured by the *strength of impact* of the change on that statement. While finding all semantic dependencies of a statement s is an undecidable problem [17], our approach provides an effective under-approximation of those dependencies through computing a subset of all statements semantically affected by s and the strengths of those statements to be impacted.

The authors are with the Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, 46556, USA. E-mails: hcai@nd.edu, rsanteli@nd.edu, and sjiang1@nd.edu

Our new technique and tool, called SENSA,[1] combines *sensitivity analysis* [19] and *execution differencing* [20]–[24] to estimate those frequencies as impact strengths. Unlike a different approach by Masri and Podgurski [25], which observes information-flow strengths for existing executions, SENSA quantifies semantic dependencies between statements for a much larger set of executions derived from existing ones to explore the semantic dimensions of such dependencies. By systematically modifying program states (resulting values of statements) in a way similar to fuzzing [26], which uses invalid or random inputs to test programs against exceptions, SENSA is able to determine the effects of statements and find dependencies that could otherwise be missed.

As a dynamic analysis, SENSA can be applied only to statements executed at least once. However, the number of executions it considers is large given the many modified executions it created and analyzed. Also, dynamic analysis has proven quite useful when executions exist (e.g., [14], [23], [27]–[31]) or can be generated (e.g., [32]–[38]). The goal of SENSA is to answer questions on the existence and frequency of semantic dependencies for a *specific set* of runtime behaviors—those caused by the provided executions and alternative ones derived automatically from them.

To evaluate SENSA, we empirically compared the effectiveness of SENSA against that of static and dynamic forward slicing for *predictive impact analysis* on various candidate change locations across six Java subjects. For each location, we ranked statements by impact frequency (likelihood) computed by SENSA and that by breadth-first search as proposed by Weiser for slicing [10]. Then, we estimated the effort a developer would spend inspecting the rankings to find all statements impacted by actual changes (both bug-fixing and bug-introducing changes) in those locations for the same test suite. Our results indicate with statistical confidence that SENSA outperforms both forms of slicing at *predicting the actual impacts* of such changes by producing more accurate rankings describing the real dependencies on those locations.

To further demonstrate the benefits of prioritizing impacts with SENSA, we also performed three case studies to investigate how well SENSA and program slicing isolate the *cause-effect chains* that explain how bugs propagate to *failing points* (crash locations or outputs) and, thus, how those bugs can be fixed. To achieve this goal, first, we manually identified the statements that are impacted by each bug and that propagate the erroneous state to a failing point. Then, we computed how well SENSA and slicing isolate those event chains. Although we cannot draw general conclusions, we found again that SENSA was better than slicing for isolating the specific ways in which buggy statements make a program fail.

The main benefit of this work is the greater effectiveness of impact prioritization via semantic-dependence quantification compared to static and dynamic slicing, which do not address all semantic aspects of dependencies but only distinguish them via breadth-first traversals. With this new kind of quantitative information, developers can identify potential code-level impacts of program entities more effectively by focusing on the code most likely or most strongly affected in practice.Thus, other dependence-based software-engineering tasks might benefit too from this work.

The main contributions of this paper are:

- A novel approach for quantifying semantic dependence based on sensitivity analysis, execution differencing, and fuzzing working synergistically.
- A new technique and tool SENSA harnessing the dependence quantification to prioritize change impacts for reducing the cost of impact inspection.
- An empirical study that compares the cost-effectiveness of SENSA against slicing, and shows the superiority of SENSA for *predictive* prioritization of impacts.
- A set of case studies that illustrates the advantages of SENSA over slicing, as the benefits of prioritizing change effects over just predicting them, in isolating cause-effect chains that propagate bugs to failing points.

II. PROBLEM STATEMENT AND MOTIVATION

Today's impact-analysis techniques usually work at coarse granularities, such as methods and classes (e.g., [14], [27], [39]–[42]). While these techniques provide a first approximation of potential change impacts, they do not distinguish which *statements* in particular in the impacted methods or classes are actually responsible for the impacts. Moreover, they can miss code-level dependencies that are not captured by higher-level structural relationships among classes or methods [43], which may lead to false-negative dependencies. Therefore, coarse analyses can inaccurately report change impacts, which, according to a study by Rovegard and colleagues [9], is a critical issue that developers encountered in practice when using impact-analysis techniques.

Another issue with today's impact-analysis approaches is that they often produce very-large impact sets [14], [15], [44]–[47]. To understand the effects of potential changes, developers need to inspect the impact sets of those potential changes. One strategy is to check the entire impact set, akin to the "retest-all" strategy in regression testing [48], but this strategy incurs too much effort. At the method level, existing dynamic impact analysis can report over 60% false positives in their impact sets [49], whereas static impact-analysis techniques are even more imprecise [45]. Such imprecision tends to result in very large impact sets. For example, impact sets of a static impact analysis for WebKit can be as large as 41,000 methods [47]. Fully inspecting such enormous impact sets is impractical.

At the statement level, the forward version of *program slicing* [10], [50] could be an attractive option for fine-grained impact analysis (e.g., [44]), yet it can also be inaccurate. For instance, *static* slicing can report too many potentially-affected statements that are not truly affected [51] whereas *dynamic* slicing [28], [30], [52] gives smaller results but can still suffer from imprecision [22], [25], [53] in addition to low recall.

In an industrial study by Acharya and Robinson [44], the large impact-set sizes were one of the most critical problems of using impact analysis: The impact sets, as large as 343,758 lines of code, cannot possibly be fully explored by developers. Indeed, producing large impact sets, hence incurring excessive

[1] An earlier, shorter version of this paper appeared in [18]

```
   float prog(int n, float s)
   {
1:   int g = n;
2:   if (g ⩾ 1 && g ⩽ 6) {
3:      s = s + (7-g)*5;
4:      if (g == 6)
5:          s = s * 1.1;
      }
      else {
6:      s = 0;
7:      print g, " is invalid";
      }
8:   return s;
   }
```

Fig. 1: Example program used throughout this paper.

inspection efforts, is one of the main challenges to today's impact analysis, even more so as modern software is increasingly complex and software evolution is usually subject to tight schedules due to the volatility of requirements, technology, and knowledge [1]. In fact, a problem reported as even more critical than the coarse granularity is that the time developers can devote for impact analysis is quite restricted [9].

A possible solution to this challenge is impact-set reduction, akin to the test selection approach in regression testing [48]. Unfortunately, researchers have tried various such approaches but the resulting impact sets can be still quite large. However, we believe that *not all statements in the large impact sets are equally strongly impacted*, thus not all of them have always the same *priority* for inspection. For tasks such as change understanding and program comprehension, developers need to isolate the *key* relationships between components of a program that explain its behavior. In particular, when preparing to change a program, developers must first identify the *main* consequences and risks of applying the changes before properly designing and propagating those changes.

With these motivations, we propose to distinguish the *degree of influence* of statements in a program and separate affected statements by *likelihood* (or *strength*) of being truly affected so that users can decide which code to inspect first. This impact prioritization approach, similar in spirit to test prioritization in regression testing [54], has not yet been exploited for impact analysis. In the rest of this paper, we present how prioritizing impacts at the statement level can be effectively realized via semantic-dependence quantification to address the challenges to impact analysis we just discussed.

III. Background

This section presents core concepts of our approach and illustrates them using the example program of Figure 1. Program `prog` in this figure takes an integer `n` and a floating point number `s` as inputs, creates a local variable `g`, initializes `g` to the value of `n`, manipulates the value of `s` based on the value of `g`, and returns the value of `s`.

A. Syntactic and Semantic Dependencies

Syntactic program dependencies [17] are derived directly from the program's syntax. These dependencies are classified as control or data dependencies. A statement s_1 is *control dependent* [55] on a statement s_2 if a branching decision at s_2 determines whether s_1 necessarily executes. In Figure 1, for example, statement 3 is control dependent on statement 2 because the decision taken at 2 determines whether statement 3 executes or not. This dependence is *intra-procedural* because both statements are in the same procedure (function or method). Control dependencies can also be *inter-procedural* (across procedures) [56].

A statement s_1 is *data dependent* [57] on a statement s_2 if a variable v defined (written) at s_2 might be used (read) at s_1 and there is a *definition-clear path* from s_2 to s_1 in the program for v (i.e., a path that does not re-define v). For example, in Figure 1, statement 8 is data dependent on statement 3 because 3 defines `s`, 8 uses `s`, and there is a path $\langle$3,4,8$\rangle$ that does not re-define `s` after 3. Data dependencies can also be classified as intra-procedural or inter-procedural. The parameters of the example `prog`, however, are inputs and thus are not data dependent on any statement.

Semantic dependencies represent the actual behaviors that the program can exhibit, which syntactic dependencies can only over-approximate. Informally, a statement s is *semantically dependent* on a statement t if there is any change that can be made to t that affects the behavior of s. More formally, as defined by Podgurski and Clarke [17], a statement s_1 is semantically dependent on a statement s_2 in a program P if and only if:

1) $\exists i \in I$ where I is the input domain of the program,
2) $\exists c \in C$ where C is the set of all possible changes to the values or conditions computed at s_2, and
3) the occurrences or states of s_1 differ when P runs on input i with and without c applied to s_2.

For example, in Figure 1, statement 5 is semantically dependent on statements 1, 2, 3, and 4 because they could be changed so that the execution of 5 changes (e.g., not executing anymore) or the state at 5 (i.e., the value of variable s) changes. In this case, the semantic dependencies of statement 5 coincide with its direct and transitive syntactic dependencies.

However, in this example, if statement 1 just declares that g is an alias of n (i.e., if it were not an executable statement) and only values of n in [1..6] are valid inputs, the condition at statement 2 would always be true and, thus, statement 5 would *not* be semantically dependent on 2 despite its being transitively syntactically dependent on that statement.

B. Program Slicing

Program slicing [10] determines which statements in a program may affect or be affected by another statement. *Static* slicing [10], [50] identifies such statements for all possible executions whereas *dynamic* slicing [30] does this for a particular execution. (Joining the dynamic results of multiple executions is called *union slicing* [58], [59].) A (static or dynamic) *forward slice* from statement s is the set containing s and all statements directly or transitively affected by s along (static or dynamic) control and data dependencies. Because slicing is based on the transitive closure of syntactic dependencies, it attempts to (over-)approximate the semantic dependencies in the program.

For example, the *static* forward slice from statement 3 in

Figure 1 is the set {3,5,8}. We include statement 3 in the slice as it affects itself. Statements 5 and 8, which use `s`, are in the slice because they are data dependent on the definition of `s` at 3. Another example is the *dynamic* forward slice from statement 1 in `prog` for input $\langle n$=0, s=1$\rangle$, which is {1,6,7,8}. In this case, statement 2 uses `g` to decide that statements 6 and 7 execute next (i.e., 6 and 7 are control dependent on 2) and statement 8 is data dependent on 6.

The size of a static slice is generally large [51] but varies according to the precision of the underlying points-to analysis used. If we use a coarse points-to analysis in which a pointer can be any memory address, a forward slice from a statement s that defines a pointer p would include any statement that uses or dereferences any pointer (which may or may not point to the same address as p) if the statement is reachable from s. Yet, even when using a precise (and expensive) points-to analysis, static slices can still be quite large.

C. Execution Differencing (DEA)

Differential execution analysis (DEA), or simply *execution differencing*, is designed to identify the runtime *semantic dependencies* [17] of statements on changes. Although finding all semantic dependencies in a program is an undecidable problem, DEA techniques (e.g., [20]–[22], [60]) can detect such dependencies on changes when they occur *at runtime* to under-approximate (find a subset of) the set of all semantic dependencies in the program. Although DEA cannot guarantee 100% recall of semantic dependencies, it does achieve 100% precision. This is usually better than what dynamic slicing achieves [22], [25].

DEA executes a program before and after a change to collect and compare the execution histories of both executions [22]. The *execution history* of a program is the sequence of statements executed and, at each statement, the values computed or branching decisions taken. The differences between two execution histories reveal which statements had their *behavior* (i.e., occurrences and values) altered by a change, which is the conditions for semantic dependence [17].

To illustrate, consider input $\langle n$=2, s=10$\rangle$ for `prog` in Figure 1 and a change in statement 3 to `s=s`. DEA first executes `prog` *before* the change for an execution history of $\langle$1(2), 2(*true*), 3(35), 4(*false*), 8(35)$\rangle$ where each element $e(V)$ indicates that statement e executed and computed the value set V. DEA then runs `prog` *after* the change, obtaining the execution history $\langle$1(2), 2(*true*), 3(10), 4(*false*), 8(10)$\rangle$. Finally, DEA compares the two histories and reports 3 and 8, whose values changed, as the dynamic semantic dependencies on this change in statement 3 for that input.

IV. Technique

The goal of SENSA is, for a program P and an input set (e.g., test suite) T, to *detect* and *quantify* the effects on P of the *runtime behavior* of a statement C or any *changes* in C. To this end, SENSA combines sensitivity analysis [19] and execution differencing [22].

In this section, we first give a detailed overview of SENSA and use an example to illustrate the overall working of our technique and how it differs from other relevant techniques. Next, we describe the different strategies that SENSA current offers for modifying program states, which we call *modification strategies*. Finally, we formally present the SENSA technique including its process flow and analysis algorithm.

A. SENSA Overview

Every statement s in a program has a role in the program. This role is needed, for example, when s is considered for changes, thus *predictive* impact analysis should be performed for s to determine that role.Ideally, to find the role of s, one should identify all statements that *semantically* depend on s [17]: Semantic dependence considers *all possible changes* to the computations performed at s for all possible inputs to represent the *effects* of the behavior of s.

Unfortunately, computing semantic dependence is an undecidable problem, although for *individual* changes and executions. For impact analysis, DEA can tell which statements are dynamically impacted by a change. However, *before* developers can design and apply a change to a statement, they first need to know the *possible effects* of changing that statement.

To both *identify* and *quantify* the actual influences (the role) of a statement s in the program for a set of executions, SENSA uses sensitivity analysis on s. SENSA repeatedly runs the program while modifying the state of statement s and identifies in detail, using DEA, the impacted statements for each modification. Then, SENSA computes the *frequency* at which each statement is impacted (i.e., the sensitivity of those statements to s). These frequencies serve as estimates, for the executions and modifications considered, of the *likelihood* and (to some extent) the *strength* of the influences of s.

By design, the modifications made by SENSA are constrained to *primitive values* and strings (common objects) computed by statements. To determine the sensitivity of the program on a computation for any other object, such as a method call c to a data structure, the user must identify the statement(s) of interest in that operation, which compute the supported values that make up the data structure, and apply SENSA to that (those) statement(s) instead of c.[2]

We use the example program `prog` in Figure 1 again to illustrate how SENSA works for inputs (2,10) and (4,20). Suppose that a developer asks for the effects of line 1 on the rest of `prog`. SENSA instruments line 1 to invoke a state modifier and also instruments the rest of the program to collect the execution histories that DEA needs. The developer also configures SENSA to modify variable g in line 1 with values in the "valid" range [1,6].

For each input I, SENSA first executes `prog` without making changes to produce the original program (referred to as *baseline*) execution history for DEA. Then, SENSA re-executes `prog` on I multiple (five for illustration purpose here) times, once for each other value of g in the range of [1,6]. We list the execution histories for this example and test input (2,10) in Table I; the execution histories for the test input (4,20) are similar. Finally SENSA applies DEA to

[2]Naturally, SENSA can be extended in the future to modify all non-primitive values. How to make those changes useful and valid remains to be investigated.

TABLE I: Execution histories for `prog` with input (2, 10)

Run	Execution history
baseline	⟨1(2), 2(true), 3(35.0), 4(false), 8(35.0)⟩
modified #1	⟨1(1), 2(true), 3(40.0), 4(false), 8(40.0)⟩
modified #2	⟨1(3), 2(true), 3(30.0), 4(false), 8(30.0)⟩
modified #3	⟨1(4), 2(true), 3(25.0), 4(false), 8(25.0)⟩
modified #4	⟨1(5), 2(true), 3(20.0), 4(false), 8(20.0)⟩
modified #5	⟨1(6), 2(true), 3(15.0), 4(true), 5(16.5), 8(16.5)⟩

the execution histories of the baseline and modified runs of each test input, and computes the *frequency* (sensitivity) (i.e., the fraction of all executions) at which each statement was *impacted* by changing its state or occurrence.

A developer can use these frequencies directly or a ranking of affected statements based on their associated frequencies.In our example, the resulting ranking is ⟨{1, 3, 8}, {4, 5}, {2, 6, 7}⟩ where lines 1, 3, and 8 are tied at the top because their states (the values of g and s) change in all modified runs and, thus, their sensitivity is 1. Lines 4 and 5 come next with sensitivity 0.2 because line 4's state changes in one modified run and 5 executes for one modification on each input (when g changes to 6) but not reached in the baseline executions. Lines 2, 6, and 7 rank at the bottom because 2 never changes its state and 6 and 7 never execute.

In contrast, static forward slicing ranks statements by dependence distance from line 1. These distances are found by a breadth-first search (BFS) of the program-dependence graph—the inspection order suggested originally by Weiser [10]. Thus, the result for static slicing is the ranking ⟨{1}, {2, 3, 4, 7}, {5, 6, 8}⟩. For dynamic slicing, a BFS of the dynamic dependence graph [52], which is the same for both inputs, yields the ranking ⟨{1}, {2, 3, 4}, {8}⟩.

To illustrate the usefulness of these rankings in this example, consider their application to *predictive* change-impact analysis. Suppose that the developer eventually decides to change line 1 to `g = n + 2`. The *actual* set of impacted statements for this change and inputs is {1, 3, 4, 5, 8}. This is exactly the set of statements placed at the top two levels of the SENSA ranking. In contrast, static slicing predicts statement 2, from the four predicted statements, as the second most-likely impacted statement, but that statement is not really impacted. Static slicing also predicts statement 5 as one of the least impacted, even though this statement is actually impacted after making this concrete change.

Dynamic slicing, perhaps against intuition, performs even worse than static slicing in this example. The ranking for dynamic slicing misses the actually-impacted statement 5 and predicts statement 2, which is not really impacted, as the second most-impacted. Note that, in the context of this paper, a *forward* version of relevant slicing [11] would not perform better either, although in general it may achieve a higher recall, than forward dynamic slicing. In this example, the forward relevant slice is identical to the dynamic slice.

Naturally, the usefulness of SENSA depends on the executions and modifications chosen as well as application-specific aspects such as the actual change made to the statement analyzed by SENSA. If, for example, the range [0,8] is used instead of [1,6] to modify g in line 1, the sensitivity of statements 4 and 5 will be higher because they will not execute for some of the modifications. (The sensitivity for statement 3 will not change as it is always affected either by state changes or by not executing when g is not in [1,6].) Also, the sensitivity for 2, 6, and 7 will be non-zero because g can now be outside [1,6]. In this particular case, the SENSA ranking does not change but the frequencies do, and the developer's assessment of the possible impacts of line 1 might rely on those quantities.

Program `prog` is a very simple example that contains only eight statements. This program does not require much effort to identify, quantify, and rank potential impacts, regardless of the approach used. In a more realistic case, however, the differences in prediction accuracy between SENSA and both forms of slicing can be substantial, as our studies presented in Sections VI and VII indicate.

B. Modification Strategies

SENSA is a generic modifier of program states at given program locations. The technique ensures that each new value picked to modify the original value in a location is *different* to maximize diversity while minimizing bias. When SENSA runs out of possible values for a test case, it stops and moves on to the next test case.

Users can specify parameters such as the modification strategy to use to pick each new value for a statement. The choice of values affects the quality of the results of SENSA, so we designed two different strategies while making it straightforward to add other strategies in the future. The built-in modification strategies are:

1) *Random*: Picks a random value from a specified range. The default range covers all elements of the original value's type except for *char*, which only includes readable characters. For some reference types such as *String*, objects with random states are picked. For all other reference types, the strategy currently picks *null*. Instantiating objects with random states is left for future work.
2) *Incremental*: Picks a value that diverges from the original value by increments of i (default is 1.0). For example, for value v, the strategy picks v+i and then picks v–i, v+2i, v–2i, etc. For common non-numeric types, a similar idea is used. For example, for string *foo*, the strategy picks *fooo*, *fo*, *foof*, *oo*, etc.

For modifications that make a program run for a very long time or forever, SENSA *skips* them when the running time of the modified program exceeds 10 times the runtime of the original program. Modifications that cause early terminations do not need special treatment, though, since SENSA can work with them the same way as with normal executions.

Completeness. Although most heap object values are not *directly* currently supported, any supported value within a heap object can be modified by SENSA at the location where that value is computed. Thus, indirectly, SENSA can target any value in memory and track its changes via differencing.

As an example of other potential strategies to add, consider the one using values observed in the past at C to replace the value currently computed at C. First, the strategy would collect

all values observed at C for the test suite. Then, the strategy picks iteratively from this pool each new value to replace at C. Values picked by this strategy tend to be more meaningful to the program since they were computed at same location.

As we mentioned earlier, the way in which SENSA modifies program executions, especially when the *Random* strategy is adopted, is similar in spirit to fuzzing [26]. However, while fuzzing is a testing technique that inputs to the program anomalous data including invalid, unexpected, and random values, the values that SENSA generates for the modifications are not necessarily invalid or unexpected. In fact, SENSA tries to produce valid values by allowing users to specify the range of value modification and the increment, for the *Random* and *Incremental* strategy, respectively.

It is worth emphasizing that, despite of the modification strategy used, the changes SENSA made to a statement are strictly constrained to *the value* computed at that statement—SENSA does not change the statement itself. It is also noteworthy that SENSA modifications are different from the mutations adopted in mutation testing [61], although both are intended to change program states. First, SENSA neither changes operators in the program nor uses any mutant operators [62], as mutation analysis does. Second, SENSA guarantees producing and using different values, both from the original ones and from those used before during the same execution, for different modifications. In contrast, applying a mutation does not necessarily end up with any change to the program state from the original one (e.g., changing a logical operator at a predicate might not change the value computed there).

C. Formal Presentation

SENSA is a technique that, for a statement C (e.g., a candidate change location) in a program P with a test suite T (or, more generally, an input set), computes for each statement s in P a relevance value between 0 and 1. These values are estimates of the frequencies or sizes of the influences of C on each statement of P (or, more precisely, the static forward slice of C). Next, we present SENSA's process and algorithm.

1) Process: Figure 2 shows the diagram of the process that SENSA follows to quantify influences in programs. The process logically flows from top to bottom in the diagram and is divided into three stages: (1) *Pre-processing*, (2) *Runtime*, and (3) *Post-processing*. For clarity, computational steps are in rectangles while inputs and outputs are in parallelograms.

For the first stage, at the top, the diagram shows that SENSA inputs a *Program* and a *Statement*, which we denote as P and C, respectively. In this stage, SENSA instruments at C in program P a call to a *Runtime* module which executes in the second stage. Also, it instruments P to collect execution histories of the program for applying DEA later on, including values written to memory [22]. (Branching decisions are implicit in the sequences of statements in execution histories.) The result of this stage is the *Instrumented program*.

In the second stage, SENSA inputs an *Input set* (test suite) T and runs the instrumented program repeatedly while modifying the resulting value computed by C (*Run repeatedly with modified states*)per test case t in T. For every

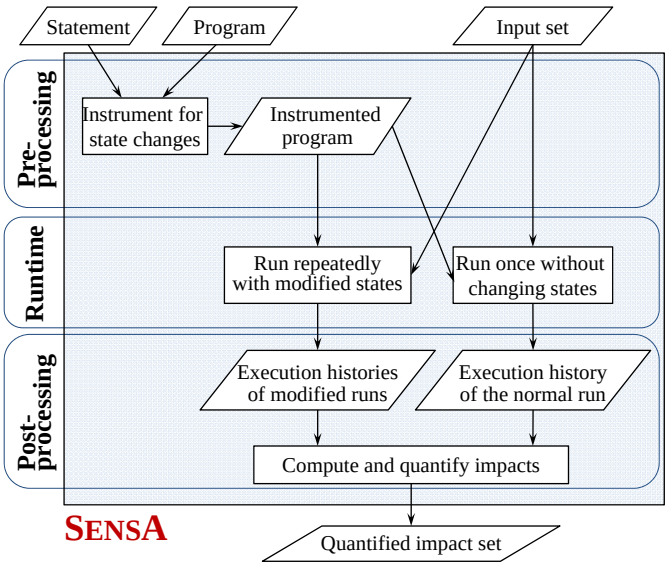


Fig. 2: The SENSA process for dependence quantification.

test t, SENSA also runs the program once without applying modifications (*Run once without changing states*) to produce the baseline execution-history for later comparisons. For the modified executions, the runtime module uses a user-specified *strategy* (see Section IV-B) and other parameters, such as a value range, to decide which value to use as a replacement of the value originally computed at statement C each time C is reached. Also, for all executions, the DEA-purpose instrumentation collects execution histories of the program with state-changes applied (*Execution histories of modified runs*) and the execution history of the original program with no changes applied (*Execution history of the normal run*) per test case.

In the third and last stage, SENSA executes DEA to compare each of the execution histories of all modified runs against the execution history of the normal run and, thus, finds out the statements affected by each modification applied to the program during the *Runtime* stage. In the meanwhile, for each test and modified execution, DEA accumulates the times each statement in the program appeared in the execution-history differences (i.e., affected by the modification), and then calculates, for all test cases and modifications, the frequency of each statement being impacted. This computation, also referred to as *influence quantification*, ends up with a set of impacted statements with their corresponding impact strengths quantified by the frequency. Finally, SENSA ranks all impacted statements based on such frequencies. This last step results in a *quantified impact set* as the eventual output of SENSA, where each statement is now associated with a rank indicating how strongly it is potentially impacted with respect to the impact strengths of other statements in that set.

2) Algorithm: Algorithm 1 formally describes how SENSA quantifies the influences of a statement C in a program P. The influenced statements found and ranked by SENSA are those in the forward static slice [50] of C in P, which are the only ones that could possibly be influenced by C. Therefore, to begin in the first stage, the algorithm computes this static

slice (line 1). At lines 2–4, SENSA initializes the map *influence* that associates to every statement in the slice its frequency and rank. Initially, these two values are 0 for all statements. Then, at line 5, the algorithm instruments P to let SENSA modify at runtime the values computed at C and to collect the execution histories for performing DEA subsequently.

For efficiency, SENSA instruments P only once to realize all the runtime modifications. To that end, it inserts at statement C probes invoking runtime monitors that generate different values for the defined variable or predicate at C. These monitors deal with the modification strategy and associated parameters (e.g., increment and value range) as configured by users, generate new values that are returned to replace the original value computed at C per the configuration, and also record values previously produced so that unique ones are applied at C in different executions. As such, modified executions are spawned from a single instrumented version P' of P, rather than by producing many different instrumented versions each producing one of those modified executions.

For the second stage, the loop at lines 7–12 determines, for all test cases, how many times each statement is impacted by the modifications made by SENSA. At line 7, SENSA executes t without modifications to obtain the baseline execution history. Then, lines 8–9 execute this same test for the number of times indicated by *NR* (short for *Number of Repetitions*) given by users as a parameter of the technique. Each time, SENSA modifies the value or branching decision computed at C with a different value.[3]

If *NR* is not specified, a default value of *NR*=20 will be used, so that at most 20 distinct modifications will be made at C, while constrained by the value range that the user provided and the data type of the value computed at C. The maximum number of modifications can be less if there are only a few distinct values (e.g., two for boolean types). Also, this default value was not determined arbitrarily but rather an empirical threshold identified through a preliminary experiment: At least for the subjects and changes studied, we tested with a larger range of values for this threshold in the preliminary experiment, and found that continuously increasing this threshold over the default value did not lead to increase in the cost-effectiveness of SENSA for any subject. In general, the user may also need to experiment with a few different values for *NR* so as to find the one that gives the best cost-effectiveness of the technique.

For the third stage, line 10 asks DEA for the differences between execution histories of each modified run and the baseline run. In lines 11–12, the differences found are used to increment the *frequency* counter for each statement affected by the modification. Then, the loop of lines 13–14 divides the influence counter for each statement by the total number of modified runs performed ($|T|$ gives the size of the input test set T), which normalizes this counter to obtain its influence frequency in the range of [0,1]. This influence frequency is used by SENSA as the measure of impact likelihood (strength)

[3]As discussed in the overview, modifications are limited to supported types of values computed at C. Any modifications involving unsupported values require applying SENSA not to C but to the statement(s) that compute the supported parts of such values.

Algorithm 1 : SENSA(program P, statement C, test suite T)

```
    // Stage 1: Pre-processing
 1: slice = STATICSLICE(P, C)
 2: influence = ∅     // map statement→(frequency,rank)
 3: for each statement s in slice do
 4:     influence[s] = (0, 0)
 5: P' = SENSA-INSTRUMENT(P, C)
    // Stage 2: Runtime
 6: for each test case t in T do
 7:     exHistBaseline = SENSA-RUNNORMAL(P', t)
 8:     for i = 1 to NR do
 9:         exHistModified = SENSA-RUNMODIFIED(P', t)
               // Stage 3: Post-processing
10:         affected=DEA-DIFF(exHistBaseline, exHistModified)
11:         for each statement s in affected do
12:             influence[s].frequency++
    // Stage 3: Post-processing (continued)
13: for each statement s in influence do
14:     influence[s].frequency /= NR×|T|
15: RANKBYFREQUENCY(influence)
16: return influence   // frequency and rank per statement
```

of a change on a statement affected by that change, and ranked in a non-ascending order (line 15) with higher (lower) value indicating stronger (weaker) impact.

V. SCOPE AND APPLICATIONS

As a dynamic analysis, SENSA requires the existence of at least one test case that covers the analyzed statement. However, test suites may not achieve 100% coverage of their programs. Therefore, SENSA is only applicable to covered statements or when new covering executions can be created.

As with any dynamic analysis, the results of SENSA are also subject to the quality and representativeness of the test cases and, in particular, those covering the analyzed statement C. The more behaviors are exercised for C, the more dependencies SENSA can find and quantify, and the more representative those behaviors are of the real usage of C, the more accurate the quantification will be.

The quality of SENSA's predictions is also a function of the accuracy of the modification strategies for modeling the effects of C's behavior. Intuitively, the more modifications are made to C, the more effects of C are reflected in the results. Therefore, the user may want to make these strategies modify C and re-execute the program as many times as it is practical according to that user's budget.

It is worth noting that, in a change-impact analysis scenario, the test suite for the subject will be used as the input set for SENSA. This test suite, perhaps with a few updates, will be used again after the changes. Thus, the change effects that the developer will experience will be subject to the same or similar runtime conditions as the one exploited by SENSA for predictive change-impact analysis.

SENSA has, potentially, a range of applications in software engineering. Virtually any task that uses program dependencies and slicing can benefit from semantic-dependence discovery and quantification. In this paper, we focus on the two applications that originally motivated our development of SENSA.

The first application is *predictive* change-impact analysis [3], [14]. Before designing and applying any changes, a developer identifies the location(s) that might need to be changed, and often the implications of changing that location(s) must be pondered early. Most impact-analysis techniques point the developer to the potentially-affected parts of the program which might need to be inspected, possibly be changed also, and later be tested against the changes made. Yet, those affected parts can be too large to fully inspect. Thus, SENSA quantifies those impacts to help the developers focus first on the most-likely impacted parts. We study this application in Section VI.

The second application is *failure comprehension*. After localizing a fault, to fix it properly, a developer might need to understand how that fault leads to a *failure point* somewhere else in the program, such as a failed assertion, a bad output, or the location of a crash. SENSA can isolate the most-likely ways in which that fault propagates to the failure point by giving high scores to the statements that participate in that propagation. Program slicing can report many ways (effects) in which a fault propagates to a failure point, but not all such effects are necessarily erroneous. SENSA can focus the developer better on the effects that matter (Section VII).

VI. STUDY: IMPACT PREDICTION AND PRIORITIZATION

To evaluate SENSA, we first studied its ability to *predict* the impacts of changes of a particular kind, *fault fixes*, in the context of the test suites of the programs containing those changes. We compared these predictions with those of *static* and *dynamic* slicing using Weiser's traversal of slices [10].[4] The rationale is that the more closely a technique approximates the actual impacts (including the ordering of impacts) that changes will have, the more effectively developers will focus their efforts to maintain and evolve their software. To this end, we formulated three research questions, two about effectiveness and the third about efficiency:

RQ1: How accurately does SENSA predict the real impacts of changes with respect to slicing?

RQ2: How accurately does SENSA predict these impacts, compared with slicing, under budget constraints?

RQ3: How expensive is it to use SENSA?

The first two questions address the comparative *effectiveness* of SENSA overall and per ranking-inspection effort (when users can only inspect a portion of the predicted ranking in practice), respectively. The third question targets the *practicality* of the technique.

A. Experimental Setup

We implemented SENSA in Java as an extension of our dependence analysis and instrumentation system DUA-FORENSICS [64], [65], which is built upon the Soot [66] Java bytecode analysis and manipulation framework. As such, SENSA works on Java bytecode as its input, without relying on accesses to the source code of programs. Both the SENSA tool and DUA-FORENSICS are available to the public for download.[5] DUA-FORENSICS also provides static and dynamic slicing and execution differencing for computing the actual impacts of changes. DUA-FORENSICS computes forward static slices by traversing data and control dependencies starting from the slicing criterion (change locations). Data dependencies and control flow are determined using context- and flow-insensitive points-to analysis. DUA-FORENSICS also performs forward dynamic slicing based on dynamic dependence monitoring. More details about DUA-FORENSICS can be found in [65]. To run our experiments, we used a Linux workstation with an 8-core 3.40GHz Intel i7-4770 CPU and 32GB DDR2 RAM.

TABLE II: Experimental subjects and their characteristics

Subject	Short description	LOC	Tests	Changes
Schedule1	priority scheduler	301	2,650	7
NanoXML-v1	XML parser	3,521	214	7
XML-security-v1	encryption library	22,361	92	7
JMeter-v2	performance tester	35,547	79	7
Ant-v2	project builder	44,862	205	7
PDFBox 1.1.0	tool for pdf files	59,576	32	7
			Total:	**42**

We studied six Java subjects of different types and sizes and seven changes (fault fixes) per subject, for a total of 42 changes. For most of these subjects (the largest five except PDFBox), seven is the maximal number of changes that our current implementation of SENSA supports with respect to the limited data types of variable it could modify (Section IV-A). Also, as we report average results over all studied changes per subject, we intended to avoid inconsistent numbers of individual data points across these subjects that would otherwise bring up possible biases in those results. Therefore, we chose seven changes consistently for all subjects (for subjects having more changes to use, we chose the first seven). Table II describes the subjects. Column *LOC* shows the size of each subject in non-comment non-blank lines of Java code. Column *Tests* shows the number of tests for each subject. Column *Changes* shows the number of changes we studied for each subject.

The first subject, Schedule1, is part of the Siemens suite that we translated from C to Java. This program can be seen as representative of small software modules. NanoXML is a lean XML parser with small memory footprint. XML-security is the XML signature and encryption component of the Apache project. JMeter is also an Apache application but for measuring the performance of software. Ant is a popular tool for building software projects. PDFBox is an Apache library for programming against PDF documents.

JMeter and Ant, in particular, exhibit some non-determinism (i.e., some behaviors vary from execution to execution for the same test input) due to their use of the system time and random number generators. To guarantee the reproducibility required by SENSA and DEA to find the impacts really caused by our changes, we manually *determinized* these subjects by ensuring that the same sequences of system-time and random

[4]Relevant slicing [11], [63] is an option in between for comparison, but a forward version must be developed first. We expect to do this in future work.

[5]http://nd.edu/~hcai/sensa/html

values were obtained before and after each change. To verify that we did not break the original program semantics of those subjects, at least for their test suites used in our studies, we re-run those test suites and found no differences in the outputs and test outcomes between the executions of the test cases before and after this manual determinization. To automate this manual process, at least for non-determinism caused by using system times or randomly generated values, a potential approach is to first utilize DEA to identify the sources of non-determinism (i.e., program locations producing different values between two executions of the same code and input), and then replace those values (in a way similar to which SENSA realizes runtime modifications) with the ones of the original types returned from an external library (like the monitors in SENSA). This library uses fixed initial values (e.g., starting times or randomization seeds) to generate consistent sequences of values to replace the original ones for any repetitions of program execution. While not a generic solution, such an approach can at least automate the determinization for particular cases and reduce manual efforts for others.

Table III lists the ids of the faults whose fixes we used as the changes for our experiment. The faults for all subjects except PDFBox were introduced by other researchers and contributed to the SIR repository [67]. For PDFBox, we obtained version 1.1.0 from its SVN repository[6] and we used a random number generator to select seven statements. For each statement, we randomly chose a mutation operator (and value when appropriate) from the *sufficient mutant operators* identified by Offutt and colleagues [62], and applied it to that statement. In short, these mutant operators attempt to mutate program states by replacing arithmetic, relational, or logical operators, or values, with alternatives of the same category or type as the original ones. Simulated faults injected through mutations as such have been shown to be effective [68], [69] and are actually used to produce many of the faults in the SIR repository too.More details on these PDFBox changes and the process we followed for generating them can be found on the SENSA project page.[5]

For each fault, the "fixed" (changed) program is the program *as is*: the one without the fault. Each fault fix modifies, adds, or deletes one to three lines of code. For Schedule1, v7 involves two methods so we chose v8 instead. (Future plans include studying SENSA on multiple changes.)

As for the test input data required by the dynamic analysis of SENSA, we utilized the input sets given by the provider of the subject programs. Specifically, for PDFBox, we used the test cases coming as part of the project package we checked out from its SVN repository; for other five subjects, we used test cases provided with the programs that we obtained all from the SIR repository. Finally, we adopted default values for SENSA parameters, including *NR*=20, increment i=1.0 for the *Incremental* strategy, and *default* data ranges (Section IV-B) for the *Random* strategy, consistently for all subjects. Also, as we mentioned before, modifications that led to abnormally long executions (10 times longer than normal ones) were discarded and not counted toward *NR*. For modifications leading to invalid values (e.g., those causing earlier termination of programs), SENSA simply handled the corresponding execution-histories as for those from normal values.

[6]http://svn.apache.org/repos/asf/pdfbox

TABLE III: Fault ids whose fixes used as changes in our study

Subject	Source	Fault ids
Schedule1	SIR	v1, v2, v3, v4, v5, v6, v8
NanoXML	SIR	v1s1, v1s2, v1s3, v1s4, v1s5, v1s6, v1s7
XML-security	SIR	v1s2, v1s3, v1s5, v1s14, v1s16, v1s17, v1s20
JMeter	SIR	v2s1, v2s2, v2s5, v2s6, v2s11, v2s13, v2s19
Ant	SIR	v2s3, v2s4, v2s5, v2s6, v2s7, v2s8, v2s15
PDFBox	SVN	Version 1.1.0 – s1, s2, s3, s4, s5, s6, s7

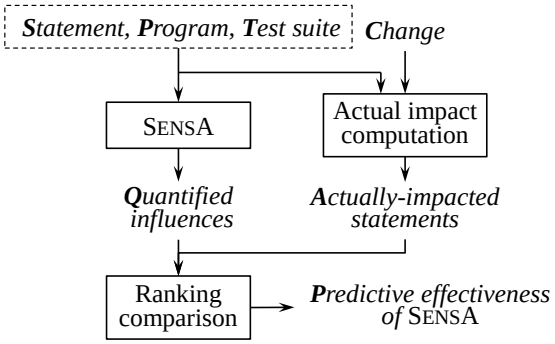


Fig. 3: Experimental process for *predictive* impact analysis, where the potential change location (given by the statement) is known to SENSA but the actual change and ground truth (actual impacts) to evaluate those predictions are not.

B. Methodology

Figure 3 shows our *experimental* process for SENSA. The inputs of SENSA are a program, a statement (e.g., a candidate change location), and the test suite of the program. SENSA quantifies the runtime influence of this statement on every other statement of the program. The output of SENSA that we use is the set of program statements ranked by decreasing influence. For tied statements (statements assigned with a same rank) in the ranking, the rank assigned to all of them is the average position in the ranking of those tied statements. To enable a comparison of this ranking with the rankings for static slicing, the SENSA ranking includes at the bottom, tied with influence zero, those statements in the static forward slice *not found* by SENSA to be affected.

Similarly, because forward dynamic slicing usually does not find all statements truly affected by potential change locations, our empirical approach also assigns to the statements in the static forward slice not found by dynamic slicing an influence of zero and adds them to the bottom of the dynamic slicing ranking to enable comparisons with SENSA and static slicing. Figure 4 illustrates the relationships among the three types of rankings for the example case discussed in Section IV-A.

To the right of the experimental process diagram (Figure 3), the *Actual impact computation* takes the same three inputs and a change for the selected statement. This procedure uses our execution-differencing technique DEA [22] to determine the *exact* set of statements whose behavior changes when running the test suite on the program before and after this change. It is crucial to note that this step of computing actual impacts with

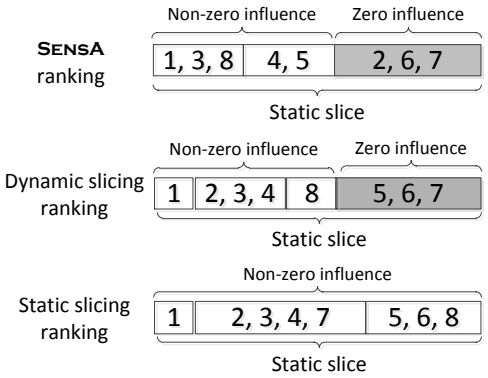


Fig. 4: An illustration of the three impact-ranking strategies and the relationships among them, using the example program and rankings of Section IV-A. Statements sharing the same rank are placed in one cell, and the ranks decrease from left to right. In each ranking, statements with *non-zero influence* are those found to be affected, while statements in the static slice but not found by the strategy are all assigned *zero influence* and appended to the bottom of the ranking. As such, each of these rankings includes the entire static slice.

DEA on both program versions (before and after the change is made) is used only for comparative evaluation purposes: SENSA, as a *predictive* impact-analysis technique, does not assume or use any knowledge about the actual changes.

The procedure *Ranking comparison* at the bottom of the diagram measures the *predictive effectiveness* of the SENSA ranking for the selected statement, when the actual change is not even known, by comparing this ranking with the ranking of the actually-impacted statements after the change is designed and applied to that statement. This *actual-impact* ranking is computed the same way as is the SENSA ranking (i.e., running DEA on the versions before and after the change), except that for the actual-impact ranking the modified program is the version in which the *actual* change has been applied.

The experimental process makes a similar comparison, not shown in the diagram, for the rankings obtained from *breadth-first searches* (BFS) of the dependence graphs for static and dynamic slicing. BFS is the traversal order of slices proposed by Weiser [10]. We should note that, while the semantics of slicing do not guarantee a stronger impact of the slicing criterion on statements closer to the top of the slice, as an impact-prioritization strategy, the ordering of statements in a slice ranking is the result of a BFS traversal of the dependencies of those statements on the criterion. Therefore, for developers who inspect the impacts in a forward direction, it is natural to compare the BFS ordering of slices to the impact-strength-based ranking of SENSA. For dynamic slicing, we join the dynamic slices for all executions that cover the selected statement. This is known as a *union slice* [59].

Ranking comparison computes the effectiveness of a ranking at predicting the actually-impacted statements by determining how high in the ranking those statements are on average. The rank of each impacted statement represents the effort a developer would invest to find it when traversing the ranking from the top. The more impacted statements are located near the top of the ranking, the more effective is the ranking at predicting the actual impacts that will occur in practice after making the change. The average rank of the impacted statements is the average *inspection effort*.

Ideal case. As a basis for comparison of the predictions given by the techniques we studied, we also computed the inspection effort for the *ideal* (best) scenario for each change. This ideal case corresponds to a ranking in which all statements impacted by the change are placed at the top of that ranking. No other ranking can make a better prediction. The ideal ranking is the actual-impact ranking which includes at the bottom all statements in the static forward slice that are not in the actual impact set. Those statements have zero influence and are tied at the bottom of the ideal ranking, as we do for SENSA and dynamic slicing for meaningful comparisons.

For RQ1, we computed, for all changes of each subject, the average inspection efforts for the entire rankings for SENSA, static slicing, and dynamic slicing. For each of these three impact-prioritization ranking strategies, we calculated this inspection effort for each change by dividing the sum of the ranks produced by the strategy for that change by the sum of the worst-case ranks (i.e., the length of the ranking given by the strategy) of all *actually-impacted* statements with respect to that change. Suppose L and A are the rankings given by a strategy and the actual-impact ranking for the change, respectively, the inspection effort for L is computed as

$$\frac{\sum_{\text{statement } s \in A} \text{the rank of } s \text{ in } L}{|A| \times |L|} \quad (1)$$

where $|A|$ and $|L|$ are the numbers of elements in rankings (sets of ordered statements) A and L, respectively. Then, we report such ratios as percentages for each change and compute the average over all seven changes per subject.

Example. Consider again the example program discussed in Section IV-A which has 8 lines of code that are all in the static slice. For SENSA, lines 1, 3, and 8 are tied at the top of the ranking with average rank (1+2+3)/3 = 2 and lines 4 and 5 are tied at average rank 4.5. SENSA does not detect differences in lines 2, 6, and 7, which get tied at the bottom with rank 7. If a change is actually made that impacts lines 1–4 and 8, the *average inspection effort* of using the predicted SENSA ranking (before making the change) is the average rank of those lines divided by the static slice size, or (2+2+2+4.5+7)/(5×8) = 43.75%. For static and dynamic slicing, the average efforts are computed similarly but with respect to the actual impacts.

For RQ2, we computed, for each ranking, the percentage of actually-impacted statements found in each fraction from the top of the ranking, for fractions $\frac{1}{N}$ to $\frac{N}{N}$ where N is the size of the ranking. The result for each change in each subject is a set of two-dimensional points, with each point (x, y) representing that $y\%$ of the actually-impacted statements can be found by inspecting the top $x\%$ of the statements in the ranking. Let L be the ranking for that change, to obtain those points, the experiment process traverses L to calculate y for each x when incrementing x by $\frac{1}{N}$.

To *merge such points across all changes* per subject for

comparison, we first linearly interpolated the data points for each change with an interval of 0.1% (i.e., to have 1,000 points for each change in the subject) and then computed the average per point for all seven changes. We report, visually using a plot of those points, the average cost-effectiveness curve per subject. Note that the plots are *usually not smooth curves*, even for the ideal case, because the size of L can be much larger than that of the corresponding actual impact set after all zero-influence static-slice statements are added to L.

For RQ3, we measured the execution time costs of the subjects for their provided test suites and the time each phase of SENSA takes on that program and test suite.

Test suite choice. By design, the test suites we use to compute the SENSA and dynamic-slicing predictions *before* making changes are the same as we used to find the actual impacts (the ground truth) *after* making the changes. To the casual reader, using the same test suite to obtain both the prediction and ground truth might seem biased. However, we chose to do so because developers will normally use the entire test suite of the program for SENSA *before* they decide their changes and then the same (occasionally updated) test suite to observe the actual impacts *after* defining and applying those changes. Therefore, using the same test suite before and after is not only appropriate, but necessary for evaluation.

Reverse analysis. In this study, we primarily looked at how well SENSA and the baseline (slicing) techniques operated on bug-fixing changes (or referred to as *original changes*) to understand various factors behind corresponding results.We found that, because these changes are fault fixes and the predictive techniques were applied to the *faulty* versions of the programs, for many of these faults the executions covering them stopped shortly afterwards, typically with an unhandled exception. Static slicing does not use these executions, but SENSA and dynamic slicing do. As a consequence, SENSA often had less data available than usual. In many cases, SENSA could not "guess" the modifications that would fix the fault and therefore missed the impacted statements that execute only when the fault is fixed (i.e., after the real change is applied).

This problem does not occur if SENSA analyzes executions that proceed normally after the candidate change location, which is the case for most other impact-prediction scenarios. Therefore, we also studied the *reverse* versions of the bug-fixing changes (i.e., bug-introducing changes, or referred to as *reverse changes*) where the fixed program is used instead by SENSA. Although the reverse changes might not be as common and representative of real changes, they nevertheless correspond to modifications that can occur in software. More importantly, studying these reverse changes provides insights on how SENSA would compare to slicing when executions continue normally after the analyzed location. Yet, since we focus on the original changes and used the reverse ones mainly for complementary analyses, and the detailed results for the latter would be shown similarly as those for the former, we give brief analyses in text only, and summary results just for RQ1 as an example, for reverse changes.

C. Results and Analysis

TABLE IV: Inspection efforts for original (bug-fixing) changes

Subject	Ideal (best) case	Average effort			
		Static slicing	Dynamic slicing	SENSA Rand	SENSA Inc
Schedule1	47.90	50.14	48.34	48.01	48.01
NanoXML	8.84	22.71	27.09	20.27	22.37
XML-security	5.00	31.94	45.37	13.15	21.49
JMeter	0.15	9.26	24.65	7.50	7.51
Ant	3.21	39.16	41.55	29.84	23.76
PDFBox	2.72	39.77	47.51	34.73	34.73
average	**11.30**	**32.16**	**39.08**	**25.59**	**26.31**
standard dev.	17.26	16.52	14.47	20.92	21.86
			p-value w.r.t. static slicing:	**3.70E-03**	**1.21E-02**

1) RQ1: Overall Effectiveness:

Data. Table IV presents the average inspection effort per subject (seven changes each), using Equation 1 to obtain the cost per change, and the average effort and standard deviation for all 42 changes. As explained before, the units are percentages. For the *Ideal* case, the effort is an absolute value representing the minimum possible effort: the best possible prediction. For each of *Static slicing*, *Dynamic slicing*, and SENSA with *Random* (SENSA-*Rand*) and *Incremental* (SENSA-*Inc*) strategies, the table shows the average effort required to find *all* actual impacts in their respective rankings.

For example, for XML-security, the best position on average in the ranking for all statements impacted by the changes is 5.0% of the static forward slice. On top of this, static and dynamic slicing add 26.94% and 40.37% average inspection effort, for a total of 31.94% and 45.37%, respectively. These extra efforts can be seen as the *imprecision* of the techniques. Meanwhile, SENSA-*Rand* and SENSA-*Inc* impose 8.15% and 16.49% extra effort, respectively, over the ideal case, which is considerably less than slicing.

Analysis. The *Ideal case* results indicate that the number of statements impacted in practice by the changes in our study, as a percentage of the total slice size, decreased with the size of the subject—from 47.90% in Schedule1 down to 5% or less in the four largest subjects. This phenomenon can be explained by two factors. First, the larger subjects consist of a variety of loosely coupled modules and the changes, which are thus more scattered, can only affect smaller fractions of the program. Second, static slicing will find connections among modules rarely, if ever, exercised at runtime: Aliasing in larger object-oriented subjects is a major reason for the imprecision of slicing. For Schedule1, however, there is little use of pointers, and most of the program executes and is impacted on every test case and by every change. In fact, an average of 97.8% of the statements in the slices for Schedule1 were impacted. These factors explain the much greater inspection efforts needed by all techniques in this subject.

Remarkably, for all subjects but Schedule1, dynamic slicing produced worse predictions than static slicing. This counterintuitive result is explained by the divergence of paths taken by executions before and after the changes. Because of these divergences, dynamic slicing missed many impacted statements that were not dynamically dependent on the input statement before the change but became dynamically dependent after the change. SENSA did not suffer so much from this problem

TABLE V: Inspection efforts for reverse changes

Subject	Ideal (best) case	Average effort			
		Static slicing	Dynamic slicing	SENSA Rand	SENSA Inc
Schedule1	47.90	50.15	48.30	48.05	48.03
NanoXML	8.84	22.64	20.66	11.23	12.24
XML-security	5.00	31.91	45.16	9.02	9.33
JMeter	0.15	9.26	20.46	8.63	9.34
Ant	3.21	37.27	33.45	15.94	15.32
PDFBox	2.72	39.77	42.60	28.60	28.58
average	**11.30**	**31.83**	**35.11**	**20.24**	**20.47**
standard dev.	17.26	16.40	17.11	20.35	20.10
p-value w.r.t. static slicing:				**7.16E-06**	**1.48E-05**

because its modifications altered some paths to approximate the effects that any change could have.

For Schedule1, the inspection effort for both variants of SENSA was, on average, 0.11% more than the ideal, which is very close to the ideal result. After SENSA, dynamic slicing also did well at predicting change impacts with only 0.44% extra effort, whereas static slicing was the worst predictor with 2.24% extra effort. Considering the high minimum effort for this small subject, however, these differences in effort are rather small in absolute terms. For NanoXML, the ideal average effort is much lower at 8.84%. Thus, impact prediction can be much more effective in this case. For this subject, SENSA-*Rand* was the best variant of SENSA, requiring 2.44% less effort on average than static slicing. SENSA-*Inc*, however, was only slightly better than static slicing, by less than 0.5%.

For XML-security, in contrast, both variants of SENSA (especially SENSA-*Rand*) were considerably more effective than static slicing, which required more than six times the ideal effort to isolate all impacts, whereas SENSA-*Rand* required less than three times the least possible effort. For JMeter, however, SENSA-*Inc* was almost as good as SENSA-*Rand*, both outperforming over static slicing. Although the difference was not large, the absolute levels of effort below 10% make that difference important.

For Ant, SENSA-*Inc* was the best variant with a considerable decrease of 15.4% in effort compared to static slicing. Finally, for PDFBox, both variants of SENSA reduced the average effort with respect to static slicing by about 5%.

Importantly, on average for all 42 changes, both variants of SENSA outperformed static slicing, with SENSA-*Rand* requiring 6.57% less effort than static slicing and 13.49% less than dynamic slicing to capture actual impacts. The standard deviation for both variants of SENSA, however, is greater than for both forms of slicing, which suggests that they are less predictable than slicing.

Statistical significance. To assess how conclusive is the advantage of both variants of SENSA over static slicing (the best-performing form of slicing here), we applied to our 42 data points a Wilcoxon signed-rank one-tailed test [70] which makes no assumptions about the distribution of the data. Both p-values, listed in the last row of Table IV, are less than .02, indicating that the superiority of both variants of SENSA, especially SENSA-*Rand*, is statistically significant.

Reverse analysis. Table V shows the results per subject and overall for the reverse changes. For these changes, the predictions of SENSA, especially SENSA-*Inc*, improve considerably. Dynamic slicing also improves but remains ineffective. Static slicing changes little as it does not depend on runtime data. The *Ideal* case is the same because execution differencing is symmetric. In all, SENSA outperforms static slicing with even stronger statistical significance for these changes.

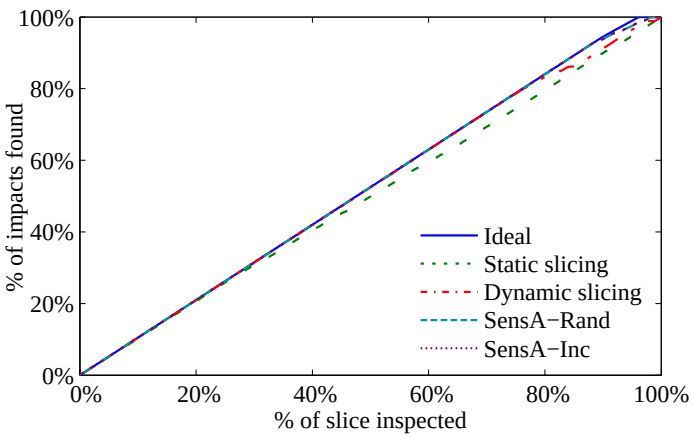
Conclusion. For these subjects and changes, with statistical significance, SENSA is more effective on average than slicing techniques at *predicting* the statements that are later impacted when changes are made. These results highlight the imprecision of static and dynamic slicing for predicting impacts, contrary to expectations, and the need for a technique like SENSA to detect semantic dependencies. SENSA is particularly superior when longer executions are available. In all, SENSA can save developers a substantial amount of effort for identifying the potential consequences of changes.

2) RQ2: Effectiveness Distribution: Developers might not always be able to examine the entire rankings produced by SENSA. In such cases, they can focus only on a fraction of each ranking, normally those statements at the top. Thus, developers can prioritize their inspection efforts by analyzing as many highly-ranked impacts as possible within their budget. To understand the effects of such prioritizations, we investigated the effectiveness of each portion of the SENSA and slicing rankings starting from the top.

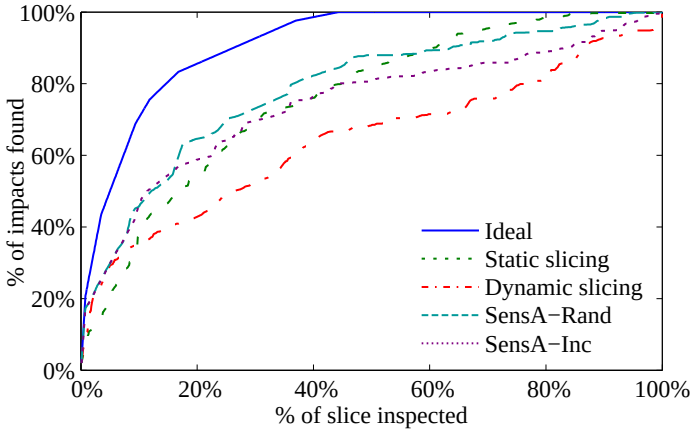
Data. Figure 5 shows, for each subject and all fault fixes in that subject, the average cost-effectiveness curves of five examination orders: the ideal (best possible) ranking, SENSA with *Random* and *Incremental* strategies, and static and dynamic slicing. The horizontal axis indicates the fraction of the ranking examined from the top of the ranking that predicts impacts. The vertical axis indicates the percentage of actually-impacted statements found within that fraction of the ranking, on average for all changes in the subject. Note that the results in Table IV are the average of the Y values for the corresponding rankings in these figures.

Analysis. The *Ideal* curves in these graphs provide detailed insights on how cost-effective impact prediction techniques can aspire to be. For all subjects except Schedule1, this curve rises sharply within the first 10% of the ranking. These curves are not straight lines because they are the average curves for all changes in each subject and the actual impacts for these changes (which define *Ideal*) vary in size. Only for Schedule1, the *Ideal* curve is mostly a straight line because the sets of actual impacts have almost the same size.

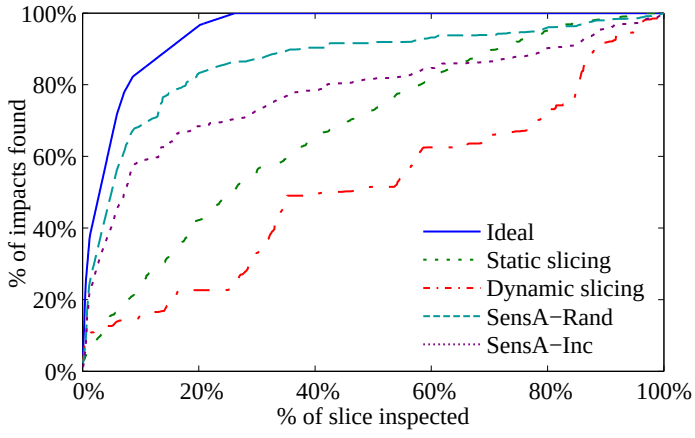
At the beginning, the curves for SENSA, especially SENSA-*Rand* for NanoXML and XML-security and SENSA-*Inc* for Ant, grow faster than those for slicing. For Schedule1, because of the high baseline (ideal) costs, all curves are very close. For that subject, the SENSA curves overlap with the ideal curve for about 90% of the ranking, whereas the dynamic-slicing curve breaks from them at about 75%. This contrast means that SENSA correctly predicts virtually all impacts for inspection budgets of 90% or less, whereas dynamic slicing has the same benefit for budgets up to 75%. Static slicing, however, always stays slightly below the ideal.



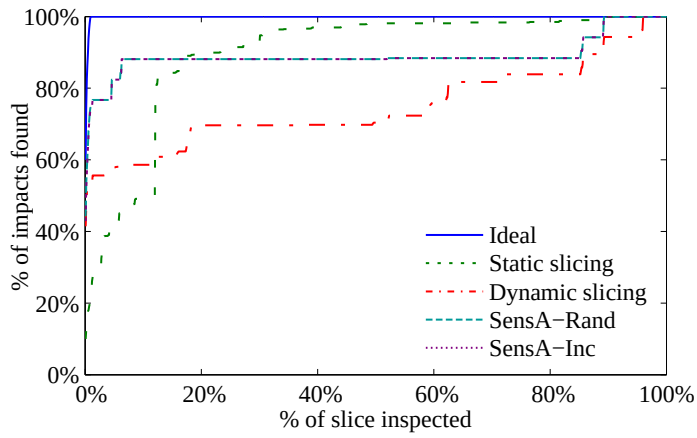
(a) Schedule1



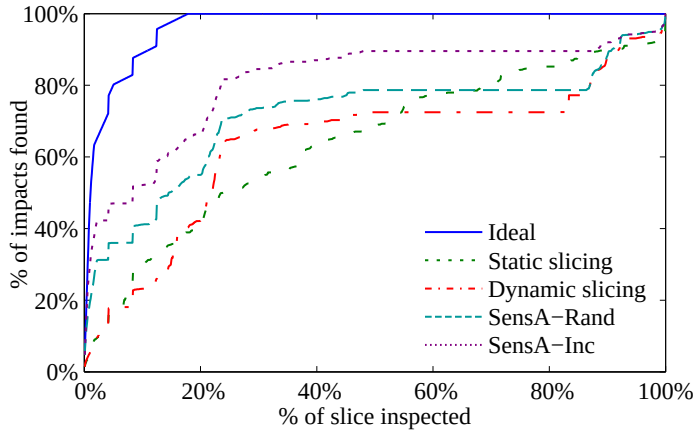
(b) NanoXML



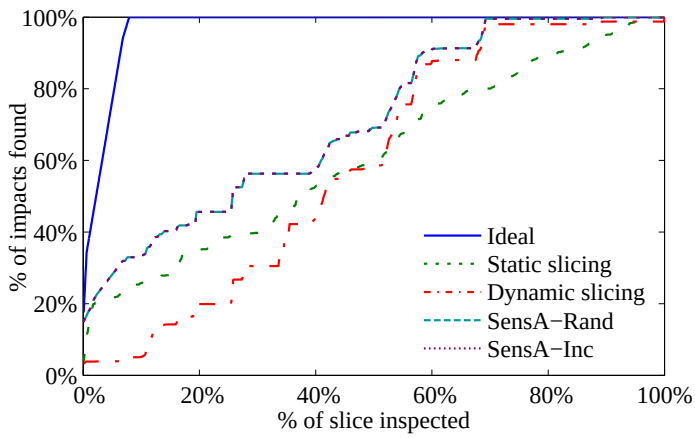
(c) XML-security



(d) JMeter



(e) Ant



(f) PDFBox

Fig. 5: Impacted code found per inspection effort for the six Java programs we studied, using different impact ranking (prioritization) strategies: the ideal case, static slicing, dynamic slicing, SENSA-*Rand* (SENSA with the *Random* strategy), and SENSA-*Inc* (SENSA with the *Incremental* strategy). In each diagram, the x axis shows the percentages of statements in a ranking that need be inspected, forwardly from the top, to reach the percentages of statements, shown on the y axis, in the associated actual-impact ranking. For each subject and ranking strategy, the curve displayed is the average of the curves for all changes for that subject, which are merged after each was normalized to 1,000 data points using linear interpolation. Because the sizes of all the rankings are the same per change as those of the corresponding static slices, which are usually much larger than the sizes of the corresponding actual-impact sets, the curves may not be smooth, even in the ideal cases.

TABLE VI: Average time costs of SENSA in seconds

Subject name	Normal run	Static analysis	Instrumented run	Influence ranking
Schedule1	186.0	6.1	4,756.8	1,054.1
NanoXML	15.4	16.5	773.1	10.0
XML-security	65.2	179.3	343.9	20.8
JMeter	40.5	604.6	2,967.8	0.3
Ant	75.2	942.9	439.0	7.0
PDFBox	5.4	504.0	1,094.0	7.7

The results for the other five subjects, which are more representative of modern software, indicate important cost-effectiveness benefits of SENSA with respect to slicing, up to about 60% of the ranking for NanoXML and 75% for PDFBox. For XML-security, SENSA-*Rand* outperforms slicing at 80% and SENSA-*Inc* does so by 65%. For Ant, SENSA-*Inc* and SENSA-*Rand* are more cost-effective than static slicing, up to about 85% and 60% of the ranking, respectively.

JMeter is, to some extent, an exception. Both variants of SENSA outperform static slicing in JMeter only up to 18% of the ranking. Beyond those points, static slicing becomes more competitive than SENSA up to 90% of the ranking. Yet, the superiority of SENSA for JMeter is substantial before this 18% point, which explains the overall advantage of SENSA reported in Table IV. More importantly, SENSA outperforms slicing considerably where we believe it matters the most, which is at the top of the ranking.

Reverse analysis. For the reverse changes, we found that the cost-effectiveness for the top portions of the SENSA rankings was even greater than for the original changes. This was expected given the greater overall superiority of SENSA for the reverse changes revealed in Table V. For NanoXML, XML-security, Ant, and PDFBox, SENSA was always better than static slicing from the top of the ranking to the point where static slicing reaches 100% effectiveness. For JMeter, SENSA was more cost-effective than static slicing up to about 24% of the respective rankings and by a greater amount than for the original changes.

Conclusion. For these subjects, changes, and test suites, SENSA is not only more effective than slicing overall (i.e., at 100% of the ranking), as Tables IV and V show, but also this superiority is concentrated at the top portions of the resulting rankings, which are the areas likely to be inspected first. Therefore, SENSA is especially more cost-effective than slicing when users are constrained by a budget which forces them to prioritize their inspection by some metric (e.g., influence).

3) RQ3: Computational Costs: Provided that the existing executions cover the statements to be analyzed, the other major factor that affects the practicality of SENSA is its computational cost.

Data. To study the cost factor, we collected the time SENSA took on the 42 changes in our experimental environment (Section VI-A) using the respective test suites. Table VI first shows the average time in seconds it took to run the *entire* test suite for each subject without instrumentation (column *Normal run*). The next three columns report the average time taken by each of the three stages of SENSA per subject.

Analysis. First, the pre-processing stage (column *Static analysis*) performs static slicing, which is needed by our experiment and also necessary for SENSA to instrument the program, as well as for dynamic slicing and DEA.As expected, this time grows with the size and complexity of the subject, where the three largest subjects (35-65K LOC) dominate. The average costs per subject were all less than 16 minutes, though, which we consider reasonably acceptable for an unoptimized prototype tool.

The runtime stage (column *Instrumented run*) of SENSA repeatedly executes 20 times (the default value for the parameter *NR*) those test cases that cover the candidate change location. In contrast with the first stage, the cost of the runtime stage is proportional to the number of test cases that cover those locations. The number of test cases available for our subjects is inversely proportional to the subject size (Table II), which explains the cost distribution seen in the table. The average costs for *Instrumented run* range from 6 to 79 minutes, which might or might not be acceptable for a developer depending on the circumstances. Finally, the costs of the third stage for all subjects except Schedule1 (for which a disproportionate total of 2650 test cases exist) are quite small as this stage simply reads the runtime data, computes frequencies, and ranks the semantically-dependent statements.

With the per-stage cost SENSA incurred put together, the total running time of our technique for one potential change (location) is 96 minutes in the worst case (for Schedule1) and 38 minutes on average over the six subjects. In contrast, worst normal runtime of the original programs is three minutes (also for Schedule1) and the mean over all subjects is about one minute. From these numbers, we can see that SENSA may cause considerable overhead relative to the normal running times. However, it is crucial to note that the predominant part of such overhead comes from the runtime stage, which can be significantly optimized by running multiple modifications and multiple test cases in parallel. Although our current prototype does not yet implement the parallelization, such optimizations can be easily added. Another important reduction in the overhead can be achieved by using fewer test cases: only those of interest or necessary for specific usage scenarios.

Reverse analysis. The results for the reverse changes (not shown here), which had longer executions, were not very different. We found that, for these changes, SENSA was no more than 1.3 times costlier than for the original changes. These results suggest that longer executions do not seem to cause an explosion in the computational cost of SENSA.

Comparisons to baselines. In comparison to static slicing, SENSA apparently costs more since computing the static slice is part (the first step) of the static analysis stage of SENSA, as seen in Figure 2 and Algorithm 1. At the same time, however, static slicing also constantly dominates the entire static-analysis cost of SENSA as shown in Table VI. Thus, the running times of the other two stages of SENSA approximately constitute its additional cost over static slicing. In addition, static slicing does not rely on the availability and quality of test inputs as SENSA does as discussed earlier in Section V. Nevertheless, like any static analysis, static slicing

can suffer from overly large analysis results with respect to all possible program inputs. And static slicing itself does not quantify and prioritize impacts as SENSA is able to. Therefore, SENSA would still be a better (more cost-effective) option for developers interested in examining impacts for concrete program behaviours with respect to specific inputs.

Like other dynamic analysis, dynamic slicing suffers from the same constraints regarding program inputs as SENSA. However, just as static slicing, the dynamic slicing technique itself does not separate impacts by relevance either. Also, our empirical data (not shown here) suggest that using dynamic slicing for impact prediction would be much less practical than SENSA just in terms of its efficiency: in our experimental environment, dynamic slicing was constantly much more expensive than SENSA, on average incurring over an hour for static analysis and about five hours for runtime alone, and total costs of over ten hours for the entire analysis for one potential change (location). The worst-case cost of dynamic slicing was even as large as over a day (on XML-security and JMeter). Although a well-optimized version of dynamic slicing may incur lower costs than we had experienced in our study, the generally heavyweight nature of the fine-grained dependence analysis required by dynamic slicing for providing results comparable to ours implies more difficult efficiency challenges to such slicing techniques in practice.

Conclusion. The observed costs are encouraging for the practicality of SENSA for three main reasons. First, we think that developers using our non-optimized prototype can in many cases accept to get the impact predictions from SENSA if they are provided budgets within the time frames that all subjects but JMeter exhibit. (For Schedule1, a smaller subset of its large test suite can be used.) Second, the cost-benefit ratio of prioritizing the inspection efforts with SENSA can be even smaller for developers inspecting larger impact sets, and the overhead can be more acceptable when the impact sets are too large to be fully inspected. Third, our prototype implementation can be significantly optimized by parallelizing the large number of runs made by SENSA, to the extent that even JMeter might be analyzable at reasonably low costs.

D. Threats to Validity

The main *internal* threat to the validity of our study is the potential presence of implementation errors in SENSA for sensitivity analysis and the underlying DUA-FORENSICS [65] for execution differencing and slicing. Although SENSA is a research prototype developed for this work, we have tested and used it for more than two years already. Meanwhile, DUA-FORENSICS has been in development for many years [64] and has matured considerably. Another internal threat is the possibility of procedural errors in our use of SENSA, DUA-FORENSICS, and related scripts in our experimental process. To reduce this risk, we tested, manually checked, and debugged the results of each phase of this process.

The main *external* threat to the validity of our study and conclusions about SENSA is that our set of subjects, changes, and test suites might not represent the effects of similar changes (bug fixes) in other software. Nevertheless, we chose our subjects to represent a variety of sizes, coding styles, and functionality to achieve as much representativeness as possible. The SIR subjects, in particular, have been used extensively in other experiments conducted by the authors and by many other researchers. Moreover, all subjects but Schedule1 are *real-world* open-source programs.

Importantly, we do *not* claim that our results generalize to *all kinds* of changes in software. We only studied changes that represent *bug fixes* and small corrections as a first demonstration of SENSA. These changes are commonly available for experimentation, yet may not be commonly found in other sources such as code repositories. To mitigate this threat, we examined different dimensions of our approach with respect to not only the bug-fixing changes but also their reverse versions. The reverse changes, to some extent, may represent a different type of changes—those that introduce bugs. Nevertheless, it will be of interest to study with even more types of changes, such as feature-adding ones, and real changes developers actually made in practical software evolution process,which we intend to explore more broadly in future work on impact analysis. In all, this study on bug-fixing changes highlights one of the many potential applications of SENSA. We presented and studied a second application in Section VII.

Another threat to external validity related to the above one comes from the limitation of our results to changes at single locations. It is possible that SENSA may exhibit different characteristics when working on multiple-location changes, including changes to multiple locations within single methods and those to either single or multiple locations across multiple methods. As one approach to accommodating such changes, SENSA itself can be extended to directly work on them by applying multiple modifications at the same time during runtime and enhancing DEA to support more generic execution-history differencing. An alternative way could be to break one large multiple-location change into a set of smaller (single-location) changes, apply the present tool to each separately, and then synthesize the results. The latter adaptation may not be viable if change interactions [22] have to be considered, though.

For *construct* validity, one threat can be our choice of ground truth (the *actual* impacts of changes) and the method to compute it. We used execution differencing (DEA) to find, for a test suite, which statements behave differently in response to changes in another statement. DEA, like SENSA, works at the *statement* level, unlike repository-mining methods which are coarser and possibly noisier. Also, the actual impacts found via DEA are a subset of *all impacts* a change can have, so we chose subjects for which reasonable test suites exist. Moreover, we used the same test suites for SENSA as for DEA so that its predictions apply to the same runtime profile.

Another construct threat is the *metric* we used to compare the effectiveness of SENSA over slicing. Intuitively, this metric correlates with the benefits that developers experience when using a predictive technique that places actual impacts higher in a ranking. However, the fidelity of this metric and the usefulness of SENSA for this task can only be, ultimately, studied on software developers in real production environments.

Lastly, a *conclusion* threat is the appropriateness of our statistical analysis and our data points. To minimize this threat, we used a well-known test that makes *no assumptions* about the distribution of the data [70], [71]. Another issue could be the use of an equal number of changes per subject. For each of the larger SIR subjects, however, seven was the largest number of faults we could use and we deemed inadequate to study less than seven changes in smaller subjects. Moreover, SENSA outperformed slicing by greater margins for larger subjects. Thus, we expect the statistical significance will increase if we drop changes from small subjects. Finally, as mentioned earlier, we compared SENSA to static slicing regarding the cost-effectiveness in the context of impact inspection and concluded about the advantage of SENSA in that regard; yet, the performance of SENSA is subject to the availability of quality execution sets required by the dynamic analysis in SENSA while static slicing does not have such constraints. Thus, static slicing might be a preferable option over SENSA in practice when high-quality inputs are not available.

VII. CASE STUDIES: FAILURE ANALYSIS

To further investigate the effectiveness of SENSA, we studied how well it predicts the *cause-effect chains* that link faulty code to a particular failure. Unlike impact analysis, whose goal is to identify *all impacted code* for maintenance, this task looks for the specific *subset of all effects* of a fault that made a program fail.

To that end, we conducted three short case studies of fault cause-effect isolation. After a (possibly) faulty statement is identified during *fault localization* [12], [72], a developer must decide *how* to fix it. This decision requires understanding how the fault really affects the *failing point* (a failed assertion, a bad output, or crash location). However, not all effects of a fault are necessarily erroneous or are responsible for the failure. The interesting behavior is the chain of events that propagates this fault to the failing point. Unlike Zeller's approach that *finds* cause-effect chains based on differencing program states between passing and failing runs [73], SENSA inputs a change (location) and *prioritizes*, in addition to finding, the effects of that change to help inspect and understand them.

We performed our case studies on the first fault provided at SIR [67] for each of these three subjects: NanoXML, XML-security, and JMeter. For each fault, we identified the *first* failing point (one statement in each case) where the fault is manifested. Given the faulty statement, we *manually* identified the sequence of all statements that propagates the fault to the failing point. We discarded affected statements that did not participate in this propagation to the failing point. All statements are Java bytecode instructions in a readable representation [66].

Given a chain of events (the set of propagating statements) and the bug fix provided with the subject, we computed how close to the top the chain is in the rankings computed by SENSA versus by forward static and dynamic slicing from the fault. While forward slicing is not typically used for debugging, these small case studies are intended for investigating the effectiveness of SENSA against both slicing techniques and in assisting with fault understanding, which may help with debugging, rather than performing debugging or fault-localization tasks directly. Specifically, we calculated the average rank of the statements in this chain in each ranking to estimate how well those rankings highlight the effects of the fault that actually cause the failure.

Although three case studies are insufficient for statistically-significant conclusions (the manual effort for these studies is large), these cases shed light on the workings of the three techniques that we studied for this application. Next, we present our results and analysis for these brief case studies.[7]

A. NanoXML

Fault v1s1 in NanoXML is located in a condition for a *while* loop that processes the characters of the DTD (Document Type Definition) of the input XML document. The execution of this fault by some test cases triggers a failure by failing to recognize the end of the DTD, which then causes an unhandled exception to be thrown when parsing the next section of the document by using the method that parses DTD. The bug and its propagation mechanism are not easy to understand because the exception is not thrown immediately after the execution of the faulty statement. After exhaustive inspection, we manually identified the 21 Java Jimple [66] statements that constitute the entire cause-effect chain from the fault to the failure.

All 21 statements that cause the failure, which will help design the bug fix, are placed by SENSA-*Rand* in the top 18.54% of the ranking and by SENSA-*Inc* in the top 23.14%, whereas static and dynamic slicing place them in the top 33.53% and 26.28% of their rankings, respectively. Thus, with either modification strategy, SENSA isolated the entire cause-effects chain better than both forms of slicing.

The average inspection effort for finding these statements, computed with the method of Section VI-B, is 7.08% for SENSA-*Rand*, 12.49% for SENSA-*Inc*, 8.86% for static slicing, and 7.44% for dynamic slicing. Therefore, for this particular fault, dynamic slicing was, comparatively, much more effective than for most impact-analysis cases studied in Section VI. Nevertheless, SENSA-*Rand* was still slightly better than dynamic slicing, close to the favorable trend observed for SENSA-*Rand* on faulty programs (Section VI-C1).

B. XML-security

Fault v1s2 in XML-security is revealed by only one of the 92 test cases available. This test fails because of an assertion failure caused by an unexpected value. Manually tracing the execution backwards from that assertion to the fault location reveals that the fault caused an incorrect signature on the input file via a complex combination of control and data flows. The complete sequence of events for the failure trace contains more than 200 Jimple statements. Yet, many of those statements are in helper functions that, for practical purposes, work as atomic operations. Therefore, we skipped those functions to identify a more manageable and focused cause-effect chain that can be understood more easily.

[7] All data details are available at http://nd.edu/~hcai/sensa/casestudies

For the reduced chain of 55 statements, SENSA places 36 of them at the top 1% of its ranking and 86.3% of those top 1% statements are in the chain. In sharp contrast, for the top 4% of its ranking, static slicing only discovers 9 of those statements. The cost of inspecting the entire sequence using SENSA is 6.6% of the slice whereas static slicing requires inspecting 33.15% of the slice and dynamic slicing needs 17.9%.

The entire forward static slice consists of 18,926 Jimple statements. Thus, users will be able to find the entire chain within the first 1,255 statements in the SENSA ranking. Using static and dynamic slicing, however, users would need to inspect 6,274 and 3,388 statements, respectively, to fully examine the same chain. Thus, for this fault, a developer using our technique can find the effects that make the assertion fail much faster than if using slicing.

C. JMeter

For this case study, we chose again the first fault provided with JMeter (v2s1) and we picked, from all 79 test cases, the one that makes the program fail for that fault. The failing point is an assertion by the end of that test. Despite the much larger size of both the subject and the forward slice from this fault than those in previous two cases, the fault-propagation sequence consists of only four statements.

Static slicing ranks two of those statements in the top 1% and the other two statements farther away. SENSA, in contrast, places the entire sequence on its top 1%, making it easier to distinguish the failure-related statements for this fault from the other, non-failing effects of it. The inspection of the entire failure sequence using static slicing would require a developer to go through 2.6% of the forward slice, or 848 statements. For SENSA, this cost would be only 0.1%, or 32 statements.

As we observed for dynamic slicing in Section VI, considering the effects of fixing this fault, dynamic slicing would cost much more than SENSA and static slicing to identify the fault-propagation sequence. To find all the four statements in the sequence, users would have to traverse 12.6% of the static slice, which corresponds to 4,094 statements. Once again, this case shows the advantage of SENSA over both forms of slicing, dynamic slicing in particular, in isolating the failure cause-effect sequence.

VIII. RELATED WORK

In [74], we outlined an early version of SENSA and we showed initial, promising results for it when compared with the predictions from breadth-first traversals of static slices [10], [12]. In this paper, we expanded our presentation of SENSA, its process, algorithm, and modification strategies. Moreover, we extended our experiments from four to six Java subjects, included dynamic slicing in our comparisons, and added three case studies of cause-effects isolation using SENSA.

A few other techniques discriminate among statements within slices. Two of them [13], [75] work on dynamic backward slices to estimate influences on outputs, but do not consider impact influences on the entire program. These techniques could be compared with SENSA if a backward variant of SENSA is developed in the future. Also for backward analysis, thin slicing [12] distinguishes statements in slices by pruning control dependencies and pointer-based data dependencies incrementally as requested by the user. Our technique, instead, can be used to automatically estimate the influence of statements in a static slice in a safe way, *without dropping* any of them, to help users prioritize their inspections.

In [25], [76], Masri and Podgurski followed an information-theoretic approach to measure the strength of dynamic information flow between variables through dynamic data and control dependencies. Their concept of dynamic flow strength is similar to the impact strength we used in SENSA for semantic-dependence quantification. However, although their approach can also be employed for quantifying dynamic dependence, we designed a more comprehensive approach to measuring dependence strengths by using fuzzed program executions and execution differencing, in contrast to using existing executions only as in [25], [76]. On the one hand, fuzzing helps SENSA alleviate the drawbacks of using a limited set of executions, which may not represent well the usage pattern of the analyzed parts of the program. On the other hand, we could enhance in future work the approach of Masri and Podgurski via fuzzing.

Program slicing was introduced as a backward analysis for program comprehension and debugging [10]. Static forward slicing [50] was then proposed for identifying the statements affected by other statements, which can be used for change-impact analysis [3]. Unfortunately, static slices are often too big to be useful. Our work alleviates this problem by recognizing that not all statements are equally relevant in a slice and that a dynamic analysis can estimate their relevance to improve the effectiveness of the forward slice. Other forms of slicing have been proposed, such as dynamic slicing [30], union slicing [58], [59], relevant slicing [11], [63], deletion-based slicing approaches [77]–[79], and the already mentioned thin slicing [12], all of which produce smaller backward slices but can miss important statements for many applications. Our technique, in contrast, is designed for forward analysis and does not trim statements from slices but scores them instead.

Dynamic impact analysis techniques [14], [27], [42], which collect execution information to assess the impact of changes, have also been investigated. These techniques, however, work at a coarse granularity level (e.g., methods) and their results are subject strictly to the available executions. Our technique, in contrast, works at the statement level and analyzes both the available executions and, in addition, multiple variants of those executions to predict the impacts of changes. Also, our technique is *predictive*, unlike others that are only descriptive [22], [42] (using knowledge of the changes already made).

Mutation testing is a specific form of sensitivity analysis that simulates common programming errors across the entire program [61], [80]–[82]. Its purpose is to assess the ability of a test suite to detect errors by producing different outputs. This approach is related to testability-analysis approaches, such as PIE [83], which determine the proneness of code to propagate any errors to the output so they can be detected. Similar to these approaches, SENSA modifies program points to affect executions but it focuses on points of interest to the user

(e.g., candidate change locations) and analyzes not only the influences on outputs but also the influences on all statements.

Many fault-localization approaches [12], [72], [84]–[86], although not directly related to SENSA, share a common aspect with our work: they assess their effectiveness in terms of the inspection effort for finding certain targets in the program. For fault localization, those targets are faults, whereas in our work they are the influences of a statement. This effort is often measured as the percentage of the program that must be inspected to reach those targets. Nevertheless, as demonstrated in a few other relevant techniques [87]–[89], impact analysis can be immediately applied in fault localization and understanding. We have preliminarily explored in this line through several case studies. It will be rewarding to further investigate potential benefits of SENSA in debugging and testing more broadly in the future.

Fuzzing is a well-known technique that has been widely used in software security testing and quality assurance [26], especially for detecting software vulnerability and reliability issues [90]. However, to the best of our knowledge, fuzzing has not been exploited for dependence quantification or impact analysis before. Yet, while SENSA shares the spirit of fuzzing, it does not necessarily use invalid or unexpected values.

IX. CONCLUSION AND FUTURE WORK

Two main challenges faced by existing *predictive* impact-analysis techniques are the coarse granularity and large size of the impact sets they produce. To address both challenges, we presented a new technique and tool called SENSA that works at the statement level by offering a fine-grained impact analysis and prioritizing impacts based on semantic-dependence quantification. This approach helps overcome the drawbacks of large impact sets without pruning statements.

Our studies suggest that SENSA outperforms the two main alternative fine-grained approaches, static and dynamic slicing, for predicting and prioritizing the impacts of candidate change locations. We conclude that prioritizing the effects (impacts) of potential changes via sensitivity analysis, powered by fuzzing, and execution differencing is more effective than slicing techniques in assisting developers with inspecting possible change effects and isolating fault cause-effect chains.

Our immediate plan is to expand our studies to subjects and changes of other types and sizes to further generalize our results and characterize the best and worst conditions for the use of SENSA. We are also developing a visualization for quantified dependencies to improve our understanding of the approach, to enable user studies, and to support other researchers. Using this tool, we will study how developers take advantage in practice of quantified slices.

Slightly farther in the future, we foresee adapting SENSA to quantify dependencies for other key tasks, such as debugging, comprehension, mutation analysis, interaction testing, and information-flow measurement. More generally, we see SENSA's scores as abstractions of program states as well as interactions among such states. These scores can be expanded to multi-dimensional values and data structures to further annotate slices. Such values can also be simplified to discrete sets as needed to improve performance.

REFERENCES

[1] V. Rajlich, "Software evolution and maintenance," in *Proceedings of the Conference on the Future of Software Engineering*, 2014, pp. 133–144.

[2] P. Jönsson and M. Lindvall, "Impact analysis," in *Engineering and managing software requirements*. Springer, 2005, pp. 117–142.

[3] S. A. Bohner and R. S. Arnold, *An introduction to software change impact analysis*. Software Change Impact Analysis, IEEE Comp. Soc. Press, pp. 1–26, Jun. 1996.

[4] V. Rajlich, "A model for change propagation based on graph rewriting," in *Proceedings of IEEE International Conference on Software Maintenance*, Sep. 1997, pp. 84–91.

[5] ——, *Software Engineering: The Current Practice*. Chapman and Hall/CRC, Nov. 2011.

[6] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 492–501.

[7] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: an exploratory study in industry," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 51:1C–51:11.

[8] C. R. de Souza and D. F. Redmiles, "An empirical study of software developers' management of dependencies and changes," in *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 241–250.

[9] P. Rovegard, L. Angelis, and C. Wohlin, "An empirical study on views of importance of change impact analysis issues," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 516–530, 2008.

[10] M. Weiser, "Program slicing," *IEEE Trans. on Softw. Eng.*, 10(4):352–357, Jul. 1984.

[11] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London, "Incremental regression testing," in *Proceedings of IEEE Conference on Software Maintenance*, Sep. 1993, pp. 348–357.

[12] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *Proceedings of ACM Conference on Programming Language Design and Implementation*, Jun. 2007, pp. 112–122.

[13] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," in *Proceedings of ACM Conference on Programming Language Design and Implementation*, 2006, pp. 169–180.

[14] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *Proc. of Int'l Conf. on Softw. Eng.*, May 2003, pp. 308–318.

[15] B. Breech, M. Tegtmeyer, and L. Pollock, "Integrating influence mechanisms into impact analysis for increased precision," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, 2006, pp. 55–65.

[16] H. Cai and R. Santelices, "Diver: Precise dynamic impact analysis using dependence-based trace pruning," in *Proceedings of International Conference on Automated Software Engineering*, 2014, pp. 343–348.

[17] A. Podgurski and L. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 965–979, 1990.

[18] H. Cai, S. Jiang, R. Santelices, Y. jie Zhang, and Y. Zhang, "SENSA: Sensitivity analysis for quantitative change-impact prediction," in *Proceedings of IEEE International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 165–174.

[19] A. Saltelli, K. Chan, and E. M. Scott, *Sentitivity Analysis*. John Wiley & Sons, Mar. 2009.

[20] K. J. Hoffman, P. Eugster, and S. Jagannathan, "Semantics-aware Trace Analysis," in *Proceedings of ACM Conference on Programming Language Design and Implementation*, Jun. 2009, pp. 453–464.

[21] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Sieve: A tool for automatically detecting variations across program versions," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2006, pp. 241–252.

[22] R. Santelices, M. J. Harrold, and A. Orso, "Precisely detecting runtime change interactions for evolving software," in *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, Apr. 2010, pp. 429–438.

[23] W. N. Sumner, T. Bao, and X. Zhang, "Selecting peers for execution comparison," in *Proceedings of ACM International Symposium on Software Testing and Analysis*, Jul. 2011, pp. 309–319.

[24] X. Zhang and R. Gupta, "Matching execution histories of program versions," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, 2005, pp. 197–206.

[25] W. Masri and A. Podgurski, "Measuring the strength of information flows in programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 19, no. 2, pp. 1–33, 2009.

[26] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

[27] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proc. of Int'l Conf. on Softw. Eng.*, May 2005, pp. 432–441.

[28] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, and B. Korel, "Theoretical foundations of dynamic program slicing," *Theor. Comp. Sci.*, vol. 360, no. 1, pp. 23–41, 2006.

[29] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[30] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, no. 3, pp. 155–163, 1988.

[31] R. Santelices and M. J. Harrold, "Demand-driven propagation-based strategies for testing changes," *Software Testing, Verification and Reliability*, vol. 23, no. 6, pp. 499–528, 2013.

[32] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of ACM International Symposium on Software Testing and Analysis*, Jul. 2010, pp. 49–60.

[33] M. Bohme, B. Oliveira, and A. Roychoudhury, "Partition-based regression verification," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, May 2013, pp. 302–311.

[34] N. Gupta and Z. Heidepriem, "A new structural coverage criterion for dynamic detection of program invariants," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, Oct. 2003, pp. 49–58.

[35] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, Apr. 2010, pp. 137–146.

[36] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, May 2007, pp. 75–84.

[37] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, "Carfast: Achieving higher statement coverage faster," in *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, Nov. 2012, pp. 35:1–35:11.

[38] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux, "eXpress: Guided path exploration for efficient regression test generation," in *Proceedings of ACM International Symposium on Software Testing and Analysis*, Jul. 2011, pp. 1–11.

[39] L. C. Briand, Y. Labiche, and G. Soccar, "Automating impact analysis and regression test selection based on UML designs," in *Proceedings of International Conference on Software Maintenance*, 2002, pp. 252–261.

[40] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, May 2012, pp. 430–440.

[41] M. Petrenko and V. Rajlich, "Variable granularity for improving precision of impact analysis," in *Proc. of IEEE Int'l Conference on Program Comprehension*, May 2009, pp. 10–19.

[42] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs," in *Proc. of ACM Conf. on Obj. Oriented Prog. Syst., Lang., and Appl.*, Oct. 2004, pp. 432–448.

[43] R. Vanciu and V. Rajlich, "Hidden dependencies in software systems," in *Proc. of IEEE Int'l Conference on Software Maintenance*, Sep. 2010, pp. 1–10.

[44] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 746–755.

[45] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *Proc. of 9th European Softw. Eng. Conf. and 10th ACM SIGSOFT Symp. on the Foundations of Softw. Eng.*, Helsinki, Finland, september 2003, pp. 128–137.

[46] A. Orso, T. Apiwattanapong, J. B. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *Proc. of 26th IEEE and ACM SIGSOFT Int'l Conf. on Softw. Eng. (ICSE 2004)*, Edinburgh, Scotland, may 2004, pp. 491–500.

[47] L. Schrettner, J. Jász, T. Gergely, Á. Beszédes, and T. Gyimóthy, "Impact analysis in the presence of dependence clusters using static execute after in webkit," *Journal of Software: Evolution and Process*, 2013.

[48] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *Software Engineering, IEEE Transactions on*, vol. 22, no. 8, pp. 529–551, 1996.

[49] H. Cai, R. Santelices, and T. Xu, "Estimating the accuracy of dynamic change-impact analysis using sensitivity analysis," in *Proceedings of International Conference on Software Security and Reliability*, 2014, pp. 48–57.

[50] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. on Prog. Lang. and Systems*, 12(1):26-60, Jan. 1990.

[51] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 2, 2007.

[52] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proceedings of ACM Conference on Programming Language Design and Implementation*, Jun. 1990, pp. 246–256.

[53] S. Jiang, R. Santelices, M. Grechanik, and H. Cai, "On the accuracy of forward dynamic slicing and its effects on software maintenance," in *Proceedings of IEEE International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 145–154.

[54] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.

[55] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Trans. on Prog. Lang. and Systems*, 9(3):319-349, Jul. 1987.

[56] S. Sinha, M. J. Harrold, and G. Rothermel, "Interprocedural control dependence," *ACM Trans. Softw. Eng. Method.*, vol. 10, no. 2, pp. 209–254, 2001.

[57] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools (2nd Ed.)*, Sep. 2006.

[58] R. J. Hall, "Automatic extraction of executable program subsets by simultaneous dynamic program slicing," *Automated Software Engineering*, vol. 2, no. 1, pp. 33–53, 1995.

[59] A. Beszedes, C. Farago, Z. Mihaly Szabo, J. Csirik, and T. Gyimothy, "Union slices for program maintenance," in *Proceedings of IEEE International Conference on Software Maintenance*, Oct. 2002, pp. 12–21.

[60] W. N. Sumner and X. Zhang, "Comparative causality: Explaining the differences between executions," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, May 2013, pp. 272–281.

[61] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[62] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99–118, 1996.

[63] T. Gyimóthy, A. Beszédes, and I. Forgács, "An efficient relevant slicing method for debugging," in *Proceedings of joint European Software Engineering Conference and ACM International Symposium on the Foundations of Software Engineering*, Sep. 1999, pp. 303–321.

[64] R. Santelices and M. J. Harrold, "Efficiently monitoring data-flow test coverage," in *Proc. of Int'l Conf. on Automated Softw. Eng.*, Nov. 2007, pp. 343–352.

[65] R. Santelices, Y. Zhang, H. Cai, and S. Jiang, "DUA-Forensics: A fine-grained dependence analysis and instrumentation framework based on soot," in *Proceeding of ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis*, Jun. 2013, pp. 13–18.

[66] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "Soot - a java bytecode optimization framework," in *Cetus Users and Compiler Infrastructure Workshop*, Oct. 2011.

[67] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Emp. Softw. Eng.*, vol. 10, no. 4, pp. 405–435, 2005.

[68] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *27th International Conference on Software Engineering*, 2005, pp. 402–411.

[69] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 654–665.

[70] R. E. Walpole, R. H. Myers, S. L. Myers, and K. E. Ye, *Probability and Statistics for Engineers and Scientists*. Prentice Hall, Jan. 2011.

[71] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Journal of Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, May 2014.

[72] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2005, pp. 273–282.

[73] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2002, pp. 1–10.

[74] R. Santelices, Y. Zhang, S. Jiang, H. Cai, and Y. jie Zhang, "Quantitative program slicing: Separating statements by relevance," in *Proceedings of IEEE/ACM ICSE - NIER*, May 2013, pp. 1269–1272.

[75] T. Goradia, "Dynamic impact analysis: A cost-effective technique to enforce error-propagation," in *Proceedings of ACM International Symposium on Software Testing and Analysis*, Jul. 1993, pp. 171–181.

[76] W. Masri and A. Podgurski, "An empirical study of the strength of information flows in programs," in *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, 2006, pp. 73–80.

[77] D. Binkley, N. Gold, M. Harman, J. Krinke, and S. Yoo, "Observation-based slicing," RN/13/13, UCL, Dept. of Comp. Sci., Jun. 2013.

[78] H. Cleve and A. Zeller, "Finding failure causes through automated testing," in *Proceedings of International Workshop on Automated Debugging*, 2000, pp. 254–259.

[79] R. A. DeMillo, H. Pan, and E. H. Spafford, "Critical slicing for software fault localization," in *Proceedings of ACM International Symposium on Software Testing and Analysis*, Jan. 1996, pp. 121–134.

[80] H. Agrawal, R. A. Demillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of mutant operators for the C programming language," Software Engineering Research Centre, Tech. Rep., Mar. 1989.

[81] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978.

[82] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, Jun. 2005.

[83] J. Voas, "PIE: A dynamic failure-based technique," *IEEE Trans. on Softw. Eng.*, vol. 18, no. 8, pp. 717–727, Aug. 1992.

[84] R. Abreu, P. Zoeteweij, and A. J. C. V. Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of Testing: Academic and Industrial Conference - Practice and Research Techniques*, Sep. 2007, pp. 89–98.

[85] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 528–545, 2010.

[86] V. Debroy and W. E. Wong, "A consensus-based strategy to improve the quality of fault localization," *Software: Practice and Experience*, vol. 43, no. 8, pp. 989–1011, 2013.

[87] X. Ren, O. C. Chesley, and B. G. Ryder, "Identifying failure causes in java programs: An application of change impact analysis," *Software Engineering, IEEE Transactions on*, vol. 32, no. 9, pp. 718–732, 2006.

[88] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, 2011, pp. 23–32.

[89] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013, pp. 765–784.

[90] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in *Proceedings of International Conference on Software Testing, Verification and Validation*, 2011, pp. 427–430.