Estimating the Accuracy of Dynamic Change-Impact Analysis using Sensitivity Analysis

Haipeng Cai, Raul Santelices
University of Notre Dame
Notre Dame, Indiana, USA
E-mail: hcai@nd.edu, rsanteli@nd.edu

Tianyu Xu
Fudan University
Shanghai, China
E-mail: xty213@gmail.com

***Abstract*—The reliability and security of software are affected by its constant changes. For that reason, developers use *change-impact analysis* early to identify the potential consequences of changing a program location. *Dynamic* impact analysis, in particular, identifies potential impacts on concrete, typical executions. However, the *accuracy* (precision and recall) of dynamic impact analyses for predicting the *actual* impacts of changes has not been studied. In this paper, we present a novel approach based on *sensitivity analysis* and execution differencing to estimate, for the first time, the accuracy of dynamic impact analyses. Unlike approaches that only use software repositories, which might not be available or might contain insufficient changes, our approach makes changes to *every* part of the software to identify actually-impacted code and compare it with the predictions of dynamic impact analysis. Using this approach *in addition* to changes made by other researchers on multiple Java subjects, we estimated the accuracy of the best method-level dynamic impact analysis in the literature. Our results suggest that dynamic impact analysis can be surprisingly inaccurate with an average precision of 47–52% and recall of 56–87%. This study offers insights to developers into the effectiveness of existing dynamic impact analyses and motivates the future development of more accurate analyses.**

Keywords*—*software evolution; change-impact analysis; dynamic analysis; empirical studies; sensitivity analysis; execution differencing

I. Introduction

Modern software is increasingly complex and changes constantly, which threatens its reliability and security. Failing to identify and fix defects caused by software changes can have enormous costs in economic and human terms [1]–[3]. Therefore, it is crucial to provide developers with effective support to identify dependencies in code and deal with the impacts of changes that propagate via those dependencies. Specifically, developers must understand the risks of modifying a location in a software system *before* they can budget, design, and apply changes there. This activity, called *(predictive) change-impact analysis* [4]–[6], can be quite challenging and expensive because changes affect not only the modified parts of the software but also other parts where their effects propagate.

A practical approach for assessing the effects of changes in a program is *dynamic* impact analysis [6]–[12]. This approach uses runtime information such as profiles and traces to identify the entities that might be affected by changes under specific conditions—those represented by the test suite for that program. The resulting *impact sets* (affected entities) of dynamic approaches are smaller than those obtained by static analyses as they represent the effects of changes for the test suite used. For scalability, most dynamic impact analyses operate on *methods* as the entities that can be changed and be impacted by changes [6]–[13]. At the statement level, dynamic slicing [14]–[16], in its forward version, can be used for impact analysis in greater detail but at a greater computational cost.

Despite their attractiveness, however, dynamic impact analyses have not been evaluated for their ability to identify or predict the *actual* impacts that changes have on software. The only exception is the CHIANTI approach [6], but the study of this approach only evaluates which *test cases* are affected by changes—which is the purpose of CHIANTI—and not which *code* is affected by them. The rest of the literature only presents experiments that compare the sizes (relative precision) of dynamic impact sets and the relative efficiency of the techniques [7]–[12] without considering how closely those impact sets approximate the real impacts of changes.

To address this problem, in this paper, we first introduce a novel approach for estimating the *accuracy* (precision and recall) of dynamic impact analyses. The approach uses SENSA, a sensitivity-analysis approach we recently developed [17], [18]. We adapted SENSA for making large numbers of random changes efficiently across the software and running dynamic impact analysis on those change locations. These random changes approximate dynamically the concept of *semantic* (real) dependence [19] as a base of comparison for dynamic impact analysis. Although random changes do not necessarily represent typical changes, the semantic dependencies they find can help identify *deficiencies in precision and recall* of dynamic impact analyses *across the entire software*.[1]

To find the *ground truth*—the code actually impacted by changes—our approach uses *execution differencing* [20]–[22] on the program before and after each change is applied to determine which code behaves differently (really affected). By design, we use the *same* test suite as the dynamic impact analysis so that we can assess the accuracy of that analysis under the same runtime conditions. The code found to behave differently by changing a location is, by definition of semantic dependence [19], truly dependent on that location. The similarities and differences between this ground truth and the impact sets indicate how accurate the impact analysis can be.

Using this approach and, *in addition*, changes (bug fixes) from the SIR repository [23], we present a thorough study of

[1]Repositories, if available, provide typical but more limited selections of changes. Nevertheless, we plan to incorporate such changes in future studies.

the accuracy of dynamic impact analysis on multiple Java subjects. For dynamic impact analysis, we chose the most precise and efficient technique from the literature: PATHIMPACT [7] with *execute-after-sequences* (EAS) [11]. We call this technique PI/EAS. (Another analysis, INFLUENCEDYNAMIC [12], marginally improves the precision of PI/EAS but at a greater expense.) For each method in a subject, we obtained the impact set predicted by PI/EAS and computed its precision and recall with respect to the ground truth.

The results of our study are rather surprising. For most subjects, the average *precision* of the impact sets is only about 50%. In other words, roughly, only one in two methods identified by PI/EAS was actually impacted by our changes. Moreover, for most subjects, the average *recall* of PI/EAS was far from perfect, revealing that dynamic impact analyses can also miss many real impacts. These findings suggest that developers should not expect a good accuracy from existing dynamic impact analyses and that there is plenty of room for improving the ability of these techniques to predict impacts.

Our study also showed that, often, the precision was high and the recall was low or vice versa. We hypothesized and confirmed that, when the program execution is shorter *before* a change (when predictive impact analysis is performed) than after a change, runtime effects are missed (e.g., many methods execute only in the changed program). Interestingly, the precision in such cases is greater than usual, suggesting that methods in dynamic impact sets are more likely to be truly impacted if they execute relatively soon after the change.

In all, the contributions of this paper are:

- A novel approach for evaluating the accuracy of dynamic change-impact analysis techniques with respect to the actual impacts of (random or real) changes
- An implementation of this approach for Java programs that performs massive numbers of changes without creating explicit copies of the program
- A study on multiple Java subjects that estimates, for the first time, the accuracy of one of the most precise dynamic impact analyses known
- An investigation finding that, for programs whose executions end early, many effects can be missed by impact analysis even though their precision increases

Section II provides the necessary background, including a working example. Section III analyzes the qualities of PI/EAS that affect its accuracy. Then, Section IV presents our approach for estimating the accuracy of dynamic impact analyses and Section V presents our study using this approach. Finally, Section VI discusses related work and Section VII concludes.

II. EXAMPLE AND BACKGROUND

Figure 1 shows an example program P used in this paper. In P, the entry method `M0` in class `C` calls methods `M1` and `M4` in classes A and B, respectively. `M1` receives two integers from `M0` and passes the sum to `M2`, which conditionally calls `M5`. Then, `M0` calls `M4`, which sets variable `t` that `M5` may read.

A. Dynamic Impact Analysis

Dynamic impact analysis uses execution information to compute change-impact sets (e.g., for a test suite) that estimate the impacts that any changes in a set of program locations might have on the entire program (at least for the executions considered). Developers typically apply the analysis on the unmodified program to find the potential impacts of changing that program. Thus, we call this analysis *predictive*.

Of the existing predictive dynamic impact analyses in the literature, PATHIMPACT [7] with *execute-after sequences* (EAS) [11], which together we call PI/EAS, has shown almost the best precision and about the best efficiency compared with the alternatives [7], [8], [10]–[12]. Only one other technique, INFLUENCEDYNAMIC [12], has shown a marginally better precision than PI/EAS but at a much greater cost.

PI/EAS computes the impact set of a method m based on the method-execution order of the program. The idea of PI/EAS is that any method called or returned into after m starts executing might be impacted by a program state modified at m. The original version, PATHIMPACT, first collects the trace of events of *entering* and *exiting* each method. Then, PATHIMPACT responds to a query for the impact set of a method m by finding in the trace the set of all methods entered or returned into after entering m. The set includes m itself.

To illustrate, Figure 1 on the bottom left shows an example trace for program P, where `r` indicates a method return, `x` is the program exit, and a method name (e.g., `M1`) represents the entry to that method. For a candidate change location `M2`, for example, PATHIMPACT first traverses the trace forward and identifies `M5` and `M3` as impacted because they are entered after `M2`. In this traversal, PATHIMPACT also counts two unmatched returns. Then, a backward traversal from `M2` finds the two matching methods `M1` and `M0` for those return events. The resulting impact set is {`M0,M1,M2,M3,M5`}. PATHIMPACT repeats this process for all occurrences of the candidate method in all traces considered and reports the union of the sets.

EAS improves PATHIMPACT to obtain the same results for a much lower cost. Instead of using traces, EAS only keeps track of the first and last time each method is entered or returns. From this, we can infer the execution order of all methods and, thus, their dynamic impacts. To illustrate, Figure 1 on the bottom right shows the *first* and *last* values within square brackets for the methods of P. A "timer" starts at 0 and is incremented on each event. The first event for `M2` occurs at time 2 when it is entered. `M4` is not executed so its registers are uninitialized. All other methods execute after time 2, so the impact set of `M2` is, again, {`M0,M1,M2,M3,M5`}. For another example, the impact set for `M3` is {`M0,M1,M3`} because only the last events for `M0` and `M1` occur after time 6.

B. Execution Differencing

Differential execution analysis (DEA) [20]–[22] identifies *semantic dependencies* [19] at runtime of statements on changes—statements *truly affected* by those changes. Formally, a statement s is semantically dependent on a change c and a test suite T if, after change c is made, the *behavior* of s (the values computed by s or executions of s) changes when running T [21]. The approach requires the execution of the program before and after the change under the same conditions for all sources of non-determinism to ensure that a difference in the behavior of s is, in fact, caused by the change.

```java
1 public class A {
2     static int M1(int f, int z) {
3         M2(f+z);
4         return new B().M3(f,1); }
5     void M2(int m) {
6         if (m > 0)
7             C.M5(); }}
```

```java
8 public class B {
9     public static int t=0;
10    int M3(int a, int b) {
11        int n = b*b - a;
12        return n; }
13    static void M4() {
14        t = 10; }}
```

```java
15 public class C {
16     public static boolean M5() {
17         return B.t > 10; }
18
19     public static void M0() {
20         if (A.M1(4,-3) > 0)
21             B.M4(); }}
```

PathImpact trace: M0 M1 M2 M5 r r M3 r r r x

EAS first-last events: M0[0,8] M1[1,7] M2[2,4] M3[6,6] M4[-,-] M5[3,3]

Fig. 1: The example program P, an example PathImpact runtime trace, and the corresponding EAS *first* and *last* events.

Although finding all semantic dependencies in a program is an undecidable problem, DEA detects at least a subset of those dependencies at runtime. To do that, DEA compares the execution histories of a program *before* and *after* a change is made. An *execution history* is the ordered sequence of statements executed and the values computed by them. The differences between histories mark the statements whose behaviors change and are thus semantically dependent on the change.

To illustrate, consider in Figure 1 the execution of P starting at `M0` and a change in line 6 to `if (m<0)`. DEA first executes P *before* the change to obtain the execution history

⟨20(*false*), 3(1), 6(*true*), 7(), 17(*false*), 4(-3), 11(-3), 12(-3)⟩

Each element $s(v)$ in this sequence indicates that statement s executes and computes value v. DEA then runs the *changed* P and obtains the execution history

⟨20(*false*), 3(1), 6(*false*), 4(-3), 11(-3), 12(-3)⟩.

Finally, DEA compares the two histories and reports statements 6, 7, and 17 as truly affected at runtime (i.e., semantically dependent on the change) because 6 computes a different value and 7 and 17 execute only before the change is made.

To study method-level impact analyses, we adapted DEA to report all methods containing at least one affected statement. We call this variant mDEA. For our example change, mDEA finds for `M2` the actually-impacted set {`M2,M5`} because those methods contain the affected statements 6, 7, and 17.

III. Analysis of PI/EAS

A. Precision

PI/EAS relies solely on runtime execution orders to identify, for a method m, its dynamic impact set. At a first glance, the technique is safe as only methods that execute after m can be affected by m. PI/EAS also produces smaller impact sets than approaches based on runtime coverage [8] for almost the same cost [11]. However, not all methods executed after m are necessarily affected by m, so PI/EAS can be quite imprecise.

For the example of Figure 1, PI/EAS predicts that the dynamic impact set of `M2` is I={`M0,M1,M2,M3,M5`}. However, after our example change in line 6 (see Section II-B), the only truly-affected methods are `M2` and `M5`. Thus, the *predictive* precision of PI/EAS in this case is only $|I \cap M|/|I| = 2/5 = 40\%$. The imprecision is caused by the limited effects of the change, which prevents 7 from executing and from calling `M5` but has no other consequence. Of course, for other changes, the precision can reach 100%. Therefore, we must empirically measure the precision of PI/EAS to draw any conclusion.

In general, given a set of executions, PI/EAS can produce large and potentially imprecise impact sets for a method m in a program. This problem occurs when one or more executions continue for a long time after the first occurrence of m and a large number of methods are called or returned into during that process, but only a small portion of those methods are dynamically dependent on m. In some cases, the execution of the program at m goes deep into a call structure but, because of modularity, a change in m propagates only to some of those calls. Similarly, m might be called when the call stack is long, making PI/EAS mark all methods in that stack as impacted even if many of them are completely unrelated in reality.

B. Recall

Naturally, no method that can only execute and return before another method m is called for the first time can be affected by the behavior of m. Therefore, in a *descriptive* sense, PI/EAS has 100% recall. For example, for the program of Figure 1, PI/EAS reports all methods but `M4` as possibly impacted by `M2`, which has 100% recall because `M4` does not execute and, thus, cannot be dynamically impacted. For another example, the impact set for `M3`, which is {`M0,M1,M3`}, also has 100% recall in a descriptive sense because `M2` and `M5` are no longer executing while or after `M3` executes.

However, developers normally need to identify to know not only the effects of a method on a single version of the program but also the impacts that changing that method can have on the entire program, even *before* changes are designed and applied. This task is called *predictive* impact analysis. Any method m' that a dynamic analysis does not report as potentially impacted by a method m might be actually impacted by a change in m that affects the control flow of the program so that m' executes after the changed m. In consequence, for *predictive* purposes, the recall of a dynamic impact analysis can be less than 100%.

For example, if the expression `b*b-a` in line 11 in our example changes to `a*a-b`, the value returned by `M3` and `M1` becomes 15 instead of -3. Thus, the expression at line 20 now evaluates to $true$ and line 21 executes, calling (and impacting) `M4`. However, the dynamic impact set for `M3` does not include `M4`. The change modifies the control flow of P so that `M4`, which did not execute before, now executes after `M3`. Thus, the recall of PI/EAS for this example change is 75%.

C. Accuracy

To be useful, in addition to efficient, an approach such as dynamic impact analysis must be accurate. Typically, neither precision nor recall measures alone are enough, but a good

balance is desired. For instance, on one hand, PI/EAS might achieve 100% recall for a method m if all methods execute after m but only a few of them is truly impacted, which yields a low precision. On the other hand, the program might always end at a method m so that PI/EAS predicts an impact set $\{m\}$ for m with 100% precision but, after changes to m, many methods might execute after m so yielding a low recall. Therefore, in this paper, we also use an *F-measure* [24] to estimate the balance of PI/EAS. We use the first such measure:

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall}$$

To illustrate, consider again the example change of statement 6 in method `M2` to `if (m<0)`. As we saw in Section III-A, PI/EAS produces an impact set of {`M0,M1,M2,M3,M5`} for `M2`, whereas the actual impact set is {`M2,M5`}. Thus, precision is 40% and recall is 100%, whereas the accuracy is $2 \times (0.4 * 1.0)/(0.4 + 1.0) = 57.1\%$. For another example, for the change in the expression in statement 11 in method `M3` from `b*b-a` to `a*a-b`, (see Section III-B), the precision is 100% and recall is 75%, for an accuracy of 85.7%.

D. Exception Handling

The PI/EAS approach, as published [11], can suffer from unpredictable results in the presence of unhandled exceptions that can make the runtime technique miss *return* or *returned-into* events. To process such events, PI/EAS assumes that an exception raised in a method m is caught in an existing *catch* or *finally* block in m before m exits or in the method that called m when m raised that exception. However, this assumption does not hold for many software systems, including some of those studied in this paper.

If neither method m nor the (transitive) callers of m handle an exception thrown in m, the *returned-into* events for m and all methods in the call stack that do not handle the exception will be missed. As a result, those methods will not be added to the resulting impact set. To illustrate, in Figure 1, if an exception is raised in `M3`, it will not be handled. Thus, the *last* records for `M1` and `M0` will not be updated to reflect that they were returned into after `M3` exited abnormally, and the impact set for `M3` will miss `M1` and `M0`.

For this paper, we decided to fix this problem by developing an improved version of PI/EAS that accounts for unhandled exceptions. Our design captures all return or returned-into events by wrapping all methods in special try-catch blocks. Those blocks catch unhandled exceptions, process the events that would otherwise be missed, and re-throw those exceptions. In the rest of this paper, whenever we mention PI/EAS, we will be referring to our fixed design.

IV. Experimental Approach

To support studies of the *predictive* accuracy of dynamic impact analyses such as PI/EAS, we designed an approach that (1) systematically applies impact analysis to a large number of candidate change locations throughout the program, (2) changes those locations one at a time, and (3) compares the predicted impact sets with the actual impacts found by MDEA.

A. Process

Figure 2 outlines our experimental approach. The process uses a *changer* module that, for each change location (statement) in a set C for program P, performs a number of changes in that location to produce one version of the program per change. For greater realism, each changed program version is treated as the *unchanged* (base) program for predictive impact analysis and the original P is treated as the "fixed", changed version. In other words, the changes can be seen as bug fixes.

The changer first instruments P at the locations C to produce a large number N of base (unchanged) versions of P called $P_{MI}1$ to $P_{MI}N$. Then, at runtime, the instrumentation in P invokes the changer for the points in C to produce the N base versions, one at a time, across which the C locations (statements) are distributed. A *change strategy* is provided for customization. By default, this strategy is *random*, which replaces the values or control-flow decisions computed at each change point with random values.

The new values that replace the original values for each execution of P are stored so that each base execution of P can be reproduced. Unlike other similar tools, to speed up the process by avoiding disk-space blowup, our system uses only two versions of the program: the original P and the instrumented P controlled at runtime by the changer. At runtime, using the test suite provided with P, the approach applies dynamic impact analysis to each of the N base versions to obtain the dynamic impact set of each method that contains a change location. Then, MDEA is applied to that version and P with the same test suite to compute the actual impacts. We *deliberately* use the same test suite so we can compare predicted and actual impacts under the same runtime conditions.

In the last step, the system compares the dynamic impact set of each changed method against the ground truth calculated by MDEA to determine the predictive precision, recall, and accuracy (F1) of that impact set. The last step, on the right of Figure 2, computes statistics on these accuracy values for the final report for the subject P.

B. Generation of Base Versions

At the core of our approach is the creation of N base versions from P. Each of those versions has one change in one statement and the subject of study is the reverse of that change that "fixes" it to obtain P. To make the study thorough and representative of the entire program, the system selects the change set C to cover as many methods of P as possible. Then, the impact set for each such method is compared against the actual impacts of *all* changes in C located in that method.

To create the changer, we adapted SENSA [17], [18], our sensitivity-analysis technique and tool for Java bytecode. SENSA can modify values of primitive types and strings in assignments and branching conditions. (Other statements not supported by SENSA are normally affected directly by those supported by SENSA.) Therefore, the change set C is selected from all statements to which SENSA is *applicable*. We call a method with no applicable statements *non-applicable*.

The goal of our approach is to change every applicable statement in the program at least once. However, this can be impractical for large subjects. Therefore, our system chooses

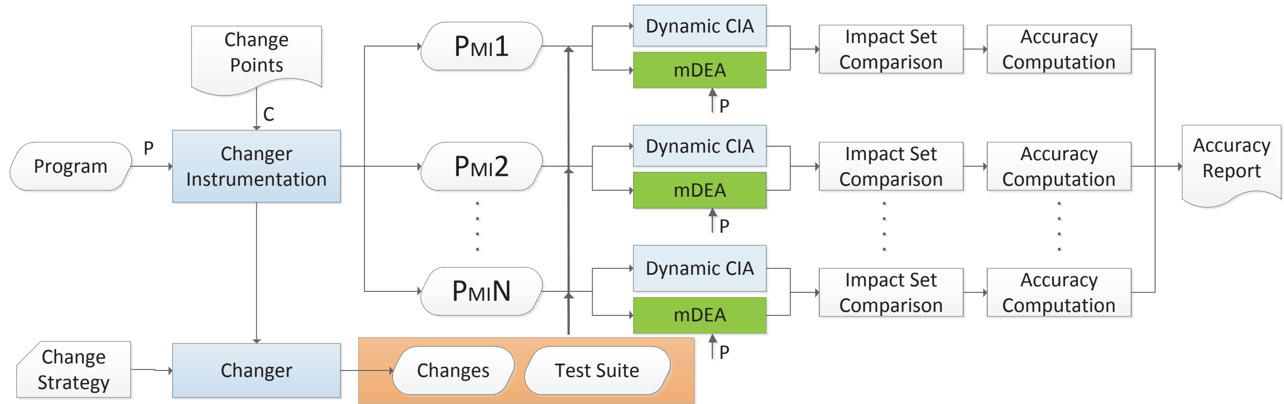

Fig. 2: Process for estimating the accuracy of dynamic impact analyses through sensitivity analysis and execution differencing.

a well-distributed subset of those statements according to per-method limits L and L_{max} which default to 5 and 10, respectively. For each method m and applicable-statements set A_m in m, the change locations set C_m for m is A_m if $|A_m| \leqslant L$. Otherwise, the size of C_m is limited to $min(|A_m|, L_{max})$ to ensure that at most L_{max} locations are used in m. In that case, to evenly cover m, the system splits the method into L_{max} segments of (almost) equal length. For each segment i of consecutive applicable statements in positions $\lceil max(0, i-1) \times |A_m|/L_{max} \rceil$ to $\lceil max(1, i) \times |A_m|/L_{max} \rceil$, the system randomly picks for C_m one statement in that segment.

V. Empirical Studies

We now present our studies of the predictive accuracy of PI/EAS using our SensA-based approach and SIR changes.

A. Experimental Setup

For our studies, we selected six open-source Java subjects, some of them well known, and the test suites provided with them. Table I lists these subjects with brief descriptions and their statistics, including their sizes in non-comment non-blank lines of Java source code (*#LOC*), the total number of methods (*#Methods*), and the number of test cases (*#Tests*).

The first five subjects came from the SIR repository for software reliability studies [23]. When applicable, the subject name includes the SIR version. Schedule1 is representative of small modules for specific tasks. NanoXML is a lean and efficient XML parser. Ant is a popular cross-platform build tool. XML-security is the encryption component of the Apache project. JMeter is an Apache application for performance assessment of software. The last subject is ArgoUML, an UML modeling tool. We chose its repository revision *r3121*—the largest the current implementation of SensA can handle.

The subjects JMeter, Ant, and ArgoUML exhibit some non-determinism (a few different behaviors for the same test inputs) due to their use of the system time and random number generators. To guarantee the reproducibility required by mDEA to find the impacts really caused by our changes, we made these subjects *deterministic* by ensuring that the same sequences of system-time and random values were used before and after each change. To check that we did not accidentally

TABLE I: Experimental subjects and their characteristics

Subject	Description	#LOC	#Methods	#Tests
Schedule1	priority scheduler	290	24	2650
NanoXML-v1	XML parser	3521	282	214
Ant-v0	Java build tool	18830	1863	112
XML-security-v1	encryption library	22361	1928	92
JMeter-v2	performance test tool	35547	3054	79
ArgoUML-r3121	UML modeling tool	102400	8856	211

break those subjects (at least for their test suites in our studies) we re-run those test suites and found no outcome differences.[2]

We implemented our approach in Java as described in Section IV as well as PI/EAS according to Section II with our fix from Section III-D. We built this infrastructure on top of our analysis framework DUA-Forensics [25], which includes mDEA, and our SensA tool [17]. Also, to separately analyze the effects of changes that shorten executions considerably, our implementation classifies changes that decrease the number of PI/EAS events to 50% or less as *shortening* (S) and the rest as *normal* (N).

B. Study I: SensA Changes

For this study, we used our full approach described in Section IV. Table II summarizes the results of this study. For each subject, the table reports the average precision, recall, and accuracy for all changes in that subject. Since the data points were collected per change, methods that contain more changes are better represented in those results. This is appropriate because they contain more code that developers could change. The first column names the subject and the second column is *Coverage*—how much of the program is truly studied—which corresponds to the percentage of all statements located in methods executed and changed at least once.

The row for each subject has three sub-rows, each named by the type of base program in the third column: *All* (both normal and shortening), *N* (normal only), and *S* (shortening only). The extent of the changes made to each subject per category is indicated by columns *#Changed Statements* for the total number of executed and changed statements and *#Changed Methods* for the number of methods containing at

[2]For the future, we plan to implement automated determinization.

TABLE II: Accuracy of PI/EAS for SENSA changes, for all changes and their two subsets: normal (N) and shortening (S).

Subject	Coverage	Change Type	#Changed Statements	#Changed Methods	PI/EAS I.S. Size	Actual I.S. Size	#FP	#FN	Precision		Recall		Accuracy (F1)	
									mean	conf. range	mean	conf. range	mean	conf. range
Schedule1	82%	**all**	**46**	**12**	**16.1**	**13.4**	**4.5**	**1.8**	**0.73**	**[0.59, 0.87]**	**0.90**	**[0.81, 0.99]**	**0.72**	**[0.59, 0.85]**
		N	12	6	16.7	5.8	11.0	0.2	0.33	[0.09, 0.56]	0.99	[0.95, 1.00]	0.43	[0.17, 0.69]
		S	34	12	15.9	16.0	2.2	2.4	0.87	[0.77, 0.97]	0.87	[0.75, 0.99]	0.83	[0.72, 0.93]
NanoXML-v1	85%	**all**	**379**	**129**	**74.6**	**54.2**	**39.3**	**18.8**	**0.46**	**[0.40, 0.52]**	**0.73**	**[0.68, 0.79]**	**0.40**	**[0.34, 0.46]**
		N	181	78	81.2	18.0	64.6	1.4	0.24	[0.16, 0.31]	0.95	[0.91, 0.98]	0.27	[0.20, 0.34]
		S	198	107	34.1	82.2	3.3	51.3	0.73	[0.66, 0.81]	0.31	[0.25, 0.36]	0.41	[0.35, 0.47]
Ant-v0	77%	**all**	**437**	**121**	**21.8**	**78.2**	**9.2**	**65.7**	**0.67**	**[0.62, 0.73]**	**0.53**	**[0.48, 0.58]**	**0.39**	**[0.35, 0.44]**
		N	381	102	17.6	27.6	8.8	18.8	0.65	[0.59, 0.71]	0.59	[0.54, 0.64]	0.43	[0.38, 0.48]
		S	56	22	49.9	422.4	12.4	384.8	0.81	[0.70, 0.91]	0.11	[0.05, 0.17]	0.14	[0.09, 0.20]
XML-security-v1	80%	**all**	**1405**	**297**	**149.1**	**208.1**	**53.8**	**112.8**	**0.7**	**[0.67, 0.73]**	**0.42**	**[0.39, 0.45]**	**0.40**	**[0.38, 0.43]**
		N	843	218	127.2	112.9	77.3	63.0	0.56	[0.52, 0.60]	0.38	[0.35, 0.42]	0.31	[0.28, 0.34]
		S	562	122	182.0	350.9	18.6	187.6	0.91	[0.89, 0.93]	0.47	[0.43, 0.52]	0.54	[0.50, 0.58]
JMeter-v2	78%	**all**	**1439**	**401**	**81.6**	**51.6**	**54.5**	**24.4**	**0.42**	**[0.39, 0.44]**	**0.58**	**[0.56, 0.61]**	**0.38**	**[0.36, 0.40]**
		N	1241	357	78.7	32.5	57.1	10.8	0.38	[0.35, 0.41]	0.60	[0.58, 0.63]	0.37	[0.35, 0.40]
		S	198	82	99.9	171.3	38.4	109.7	0.66	[0.60, 0.72]	0.44	[0.37, 0.51]	0.42	[0.36, 0.48]
ArgoUML-r3121	70%	**all**	**1239**	**421**	**81.6**	**51.4**	**56.9**	**26.6**	**0.39**	**[0.36, 0.41]**	**0.65**	**[0.63, 0.67]**	**0.37**	**[0.35, 0.39]**
		N	1043	371	77.3	33.3	58.8	14.8	0.34	[0.32, 0.37]	0.68	[0.65, 0.71]	0.35	[0.34, 0.37]
		S	196	70	104.9	147.6	46.9	89.7	0.62	[0.55, 0.70]	0.49	[0.43, 0.54]	0.44	[0.38, 0.49]
Overall (all subjects)	73%	**all**	**4945**	**1381**	**46.7**	**47.8**	**25.0**	**26.8**	**0.52**	**[0.50, 0.53]**	**0.56**	**[0.55, 0.58]**	**0.39**	**[0.38, 0.40]**
		N	3701	1132	38.2	26.4	25.8	11.6	0.43	[0.41, 0.45]	0.59	[0.58, 0.61]	0.35	[0.34, 0.37]
		S	1244	415	80.0	135.6	21.0	87.4	0.79	[0.77, 0.81]	0.44	[0.41, 0.47]	0.47	[0.45, 0.50]

least one changed statement. Note that the sums of numbers of methods for categories *N* and *S* can be greater than for *All* because some methods contain both *N* and *S* changes.

Next, the table shows the accuracy results per subject and change category, starting with the average number of impacted methods found by PI/EAS (*I.S.* means *impact set*) and the average number of actually-impacted methods identified by MDEA (*Actual I.S. Size*). The next two columns show the average number of false positives (*#FP*) and false negatives (*#FN*) for PI/EAS with respect to the actual impacts. Finally, the last three columns show the average resulting precision, recall, and accuracy (*F1*) of PI/EAS for the subject and change category. Each of those columns presents the mean and its 95% confidence interval (*conf. range*) obtained via the *non-parametric* Vysochanskij-Petunin inequality [26], which makes *no assumptions* about the distribution of the data.

To illustrate, consider the results for JMeter, for which 78% of its code was in methods that contained one or more changes, which are those analyzable by PI/EAS and MDEA. Of the 1439 statements on which changes were studied, distributed over 401 methods, 198 of them, distributed over 82 methods, contained changes that shortened the executions of the base program to less than half. On average for JMeter, the PI/EAS impact set had 81.6 methods, the actual impacts were 51.6, and the false positives and negatives of PI/EAS were 54.5 and 24.4 methods, respectively. The average precision of PI/EAS was 0.42 with 95% confidence that its real value is not outside the range [0.39, 0.44]. Recall and accuracy are presented similarly.

The last row presents the overall results for *all* changes in all subjects, so that each change has the same weight in those results. Thus, subjects with more changes (column *#Changed Statements*) have a greater influence in those results. Overall, the changed methods covered 73% of the code even though these methods were only a fraction of all methods in the subjects (see Table I). This means that the methods that never executed or for which SENSA was not applicable were much smaller than the average. In total, the study spanned almost 5000 changes. About 3 in 4 of them were *normal*.

For all changes, on average, the precision of PI/EAS was .52, its recall was .56, and its accuracy was only .39. The non-parametric statistical analysis shows with 95% confidence that these values are no farther than 2 percent points from the real value. (Naturally, for individual subjects which have fewer data points, the confidence ranges are wider). These numbers indicate that only a bit more than one in two methods reported by PI/EAS are actually impacted by those changes. Also, almost one in two methods truly impacted were missed by PI/EAS for a low recall. On average, the PI/EAS impact sets were about the same size as the actual impact sets, but the large numbers of false positives and false negatives led to this low accuracy. Thus, we can conclude with high statistical confidence that, at least for SENSA changes, the accuracy of PI/EAS is low. Hence, for many practical scenarios, dynamic impact analyses appear to be in need of considerable improvements.

For a more detailed view of the distribution of the accuracy of PI/EAS, Figures 3–5 present box plots for the precision, recall, and accuracy (F1) of all subjects for changes in *all*, *N*, and *S* categories, respectively. Each box plot consists of the minimum (lower whisker), the 25% quartile (bottom of middle box), the 75% quartile (top of middle box), and the maximum (upper whisker) of the three metrics, respectively. The medians are shown as central marks within the middle boxes. The vertical axis of each box plot represents the values of the metrics—precision, recall, and accuracy—for all changes in the corresponding subject.

For Schedule1, the simplest subject, the precision, recall, and accuracy are the highest of all subjects. This can be explained by the shorter executions and smaller number of methods in Schedule1, which makes any change likely to truly impact the methods executed after it. The box plots for Schedule1 also show the concentration on the top of the accuracy values for its 46 changes. NanoXML also had a high recall, possibly for the same reasons as Schedule1, but its precision was low—less than half the methods that execute after the change were truly impacted. This low precision suggests that NanoXML performs a larger number of independent tasks (so that changes to one task do not affect the other tasks).

For the largest four subjects, the average recall was much lower, ranging from .42 to .65, suggesting that changes in them have greater effects on their control flow because false-negative methods not covered after executing change locations in base versions do execute in changed versions. In other words, there seem to be many methods that execute under specific conditions satisfied only in changed program versions. As for precision, Ant and XML-security had a greater value than NanoXML and closer to Schedule1, suggesting that the degree of propagation of the effects of changes in those subjects is high, possibly by performing sequences of tasks that feed into each other. The largest two subjects had the lowest precisions, suggesting that their internal tasks are less coupled.

When considering the N and S categories separately, the changes in N usually have higher recalls than changes in S. This result was expected, as *normal* base versions of the subjects execute more methods and, therefore, the larger impact sets of PI/EAS should include more actual impacts. At the same time, the precision for S was greater than for N, which we also expected because the shorter executions analyzed by PI/EAS correspond to methods executed soon after each change, which we speculated would be more likely related to the changed method and, thus, actually impacted.

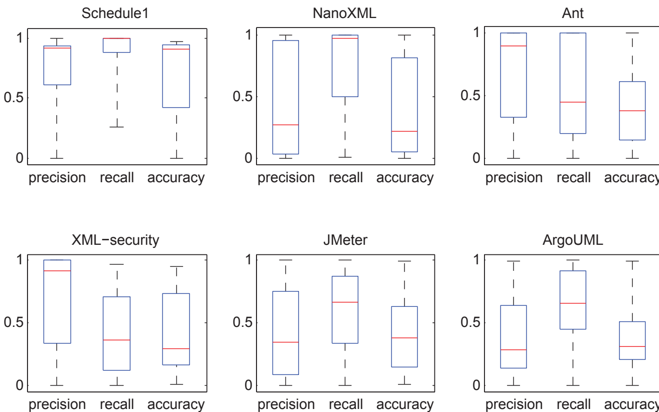

Fig. 3: Distribution of accuracy of PI/EAS for *all* changes.

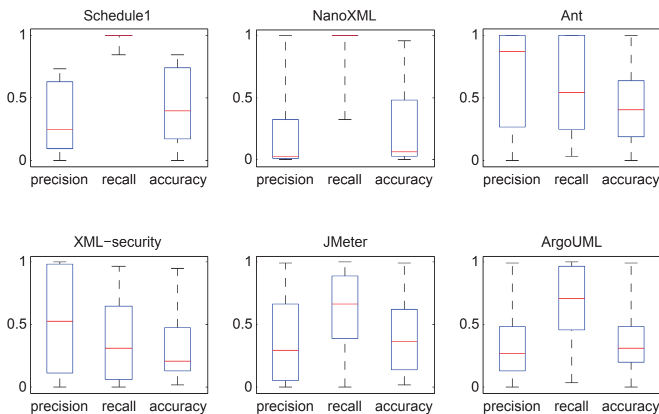

Fig. 4: Distribution of accuracy of PI/EAS for *normal* (non-shortening) changes.

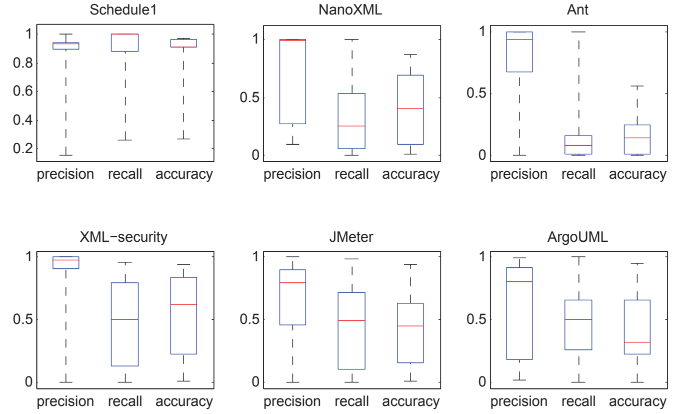

Fig. 5: Distribution of accuracy of PI/EAS for *shortening* changes.

The recall trend for N and S, however, does not apply to XML-security, where the recall is lower for N than for S. To understand this phenomenon, we manually examined the source code and executions of this subject by randomly picking five change points of type N with a recall below .0001. These changes, by definition of N, had traces of similar length before and after each change. However, the traces diverged for the most part after each change, making the actual impacts very different from the predicted impact sets. In contrast, for the changes of type S in this subject, the recall was greater, suggesting that in reality these changes did not affect the control flow of the program so dramatically.

C. Study II: SIR Changes

To contrast and complement our SENSA-based results, we performed a second study on four subjects from Table I also available at the SIR repository [23]: Schedule1, NanoXML, XML-security and JMeter. Our choice was guided by the availability of faults (real or artificial) submitted by other researchers to this experimental repository. The changes in this study are the fixes of those faults. We adapted our experimental infrastructure so that the change points in our approach are the locations of these faults and the *changer* (Figure 2) uses the fixes of these SIR faults instead of random changes.

For the three largest subjects, the maximum number of SIR changes usable for our approach was exactly seven. Thus, for each subject, we chose seven changes and run PI/EAS on the faulty versions to predict the impact set of the corresponding bug fixes. These changes (fixes) usually involve one or a few statements. For Schedule1, more changes were available but we chose the first seven to prevent this subject from having a disproportionate weight in the overall results. For these 28 changes, we compared the PI/EAS impact sets with those of MDEA to determine the predictive accuracy of PI/EAS.

Table III, similar to Table II, shows the results of the study. We did not classify the SIR changes into the N and C categories because each of those categories did not have enough changes to produce confidence ranges narrow enough to be meaningful. The PI/EAS and actual impact set sizes were similar to those in Table II for all subjects except JMeter, suggesting that the SIR changes on JMeter were less similar the SENSA ones for this subject, although the ratio of predicted to actual impacts did not differ much.

TABLE III: Second study of the accuracy of PI/EAS using SIR changes.

Subject	#Changes	#Changed Methods	PI/EAS I.S. Size	Actual I.S. Size	#FP	#FN	Precision		Recall		Accuracy (F1)	
							mean	conf. range	mean	conf. range	mean	conf. range
Schedule1	7	4	18.0	10.1	7.9	0.0	0.56	[0.38,0.74]	1.00	[1.00,1.00]	0.71	[0.57,0.85]
NanoXML-v1	7	7	96.7	30.1	66.9	0.3	0.39	[0.01,0.80]	0.99	[0.98,1.00]	0.48	[0.09,0.88]
XML-security-v1	7	6	189.7	159.4	119.3	89.0	0.53	[0.13,0.92]	0.64	[0.24,1.00]	0.40	[0.18,0.61]
JMeter-v2	7	7	37.1	17.4	22.6	2.9	0.40	[0.05,0.75]	0.84	[0.52,1.00]	0.43	[0.12,0.74]
Overall (all subjects)	**28**	**24**	**85.4**	**54.3**	**54.1**	**23.0**	**0.47**	**[0.30,0.64]**	**0.87**	**[0.72,1.00]**	**0.51**	**[0.36,0.66]**

For Schedule1, NanoXML, and JMeter, the average precision of PI/EAS was even lower for SIR changes than for SENSA changes—by 2 to 17 percent points. These numbers are actually closer to the category N of SENSA changes, suggesting that, for most SIR changes, the changed subject still runs for long enough for PI/EAS to identify as impacted larger numbers of unaffected methods. Interestingly, the bounds of the confidence ranges of precision resemble the average precisions for categories N and S in that first study.

Recall for SIR changes, in contrast, was much higher on average per subject than the recalls for all SENSA changes. Similar to the case of precision, though, the recalls for SIR changes were closer to those for the category N of changes in the first study. However, this difference is still considerable and suggests that, if it is not caused by the smaller number of data points, the recall of PI/EAS might not be as bad as we found in out first study. However, recall is still far from perfect. In that 13% of false negatives there can be many important impacts missed by users of PI/EAS. This increase in recall, however, is unable to make up for the lower precision—the accuracy (F1) per subject is almost the same in both studies.

D. Implications of the Results

The goal of our studies was to assess the practical accuracy of the best dynamic impact prediction technique in the literature, which is critical to client applications such as regression testing and maintenance—needed to ensure reliability and security over time. Despite differences mostly in recall in both studies, the results show that the predictive accuracy of PI/EAS can be surprisingly low. Although we cannot generalize the magnitude of the accuracy values obtained using SENSA and SIR changes, the numbers cast serious doubts on the effectiveness and practicality of PI/EAS. Thus, more accurate dynamic techniques are needed to fully exploit the knowledge that program executions can provide.

From these results we conclude that, at least for the subjects and types of changes considered, the precision of PI/EAS can indeed suffer because this technique is quite conservative—it assumes that all methods executed during or after the change are *infected* [27], [28] (affected) by the change. In practice, however, methods execute for many purposes and the order of execution does not always imply dependence.

Moreover, while PI/EAS was supposed to be safe relative to the execution traces utilized [11], our studies revealed that PI/EAS can even suffer from low recall. However, this drawback of dynamic impact analysis has not been noticed or reported prior to our studies in this paper. The recall problem is particularly important for changes for which the base (unchanged) program has smaller executions, such as when there is a crashing bug that needs fixing. Therefore, developers who consider using PI/EAS may first want to determine whether a change would lengthen the execution of the program. If so, PI/EAS should probably be avoided.

E. Threats to Validity

The main *internal* threat to the validity to our observations and conclusions is the possibility of implementation errors in our infrastructure, especially in the new modules PI/EAS and MDEA. However, both are built on top of our analysis and instrumentation framework DUA-FORENSICS, which has been in development for many years [25], [29] and has matured considerably. Moreover, DUA-FORENSICS has been carefully tested over time. Another internal threat is the potential for procedural errors in our use of the infrastructure, including custom scripts for experimentation and data analysis. To reduce this risk, we tested, inspected, debugged, and manually checked results from all phases of our process.

Another *internal* threat is the determinization process for Ant, JMeter, and ArgoUML. Therefore, we inspected and validated the determinizations by checking that they did not affect the execution behavior or semantics of the original programs—at least for the test suites on which our studies are based. We compared the results of all test cases for the determinized and the original versions of each of these subjects and found no differences. Furthermore, to confirm that we did not miss other sources of non-determinism, we run the determinized programs multiple times and used MDEA to determine any differences among them. We found no such differences. Of course, many programs are almost impossible to determinize but this is a limitation of the approach and not of our studies. This limitation might affect the programs we and other researchers and developers might study in the future.

The main *external* threat to our studies is the representativeness of the changes (bug fixes) that we used, in particular the random modifications inserted by our system and the changes corresponding to their fixes. Yet, these changes directly implement the concept of right-hand-side function replacements to show semantic dependencies [19], which is ultimately what dynamic impact analysis looks for. Moreover, we studied a very large number of those changes distributed evenly across every subject.

Another *external* threat is the representativeness of our selection of subjects with respect to software in general. The representativeness of the test suites is also a possible threat with respect to typical test suites found in other subjects. To limit these threats, we chose our subjects of different sizes, coding styles, and functionality to maximize variety in our studies within our available resources. We also chose the subjects based on the comprehensiveness of their test suites.

Finally, all subjects except for Schedule1 are applications and libraries used around the world.

The main *construct* threat to the validity of our studies lies in the design of our approach and the ability of random modifications and their fixes to produce similar effects to other changes that can be made to software. While actual changes made by developers might be found in source-code repositories, our design ensured that *all* parts of the program were studied. (In a repository, only a few parts are changed between two versions of a program.) More importantly, we wanted primarily to determine the presence or absence of inaccuracies in impact analysis which, if found, suggest similar problems for all kinds of changes. Also, we made sure that the modifications had a real effect, even if just local, on the program at runtime. These (bug) fixes represent at least a subset of all software changes that developers can make.

Also, as mentioned in Section IV-B, our changes were constrained to the value types that SENSA currently can change, which are primitive types and strings. Nevertheless, the SENSA-based changes were evenly spread, covered a large portion of each subject, and allowed our study to include most non-trivial methods and thus include most parts a developer who considers changing might run predictive impact analysis on. Moreover, the seeded faults in our second study were taken from the SIR repository and have been used in many studies. Those faults, whose fixes we used as changes, can be considered as representative of many real faults.

Another *construct* threat can be the method we used to determine the actual impacts of changes as the ground truth. We used execution differencing with MDEA to find, for each test suite and change, which methods behave differently in response to that change by identifying the underlying statement-level impacts using DEA. The statements found by DEA, however, are only a subset of *all* effects that the changes may have because the test suites might not be representative enough. To minimize this threat, we used subjects for which reasonably-sized test suites are provided and we studied dynamic impact analysis constrained to those same test suites so that the same operational profile is studied.

Finally, a *conclusion* threat is the appropriateness of our statistical analysis and our data points. To minimize this threat for our two studies, we used a *non-parametric* analysis [26] that computes confidence ranges without making any assumptions about the distribution of the data. This is the safest way to statistically analyze any data set. In addition, we studied the precision and recall metrics as well as the F1 accuracy metric. We included the quartile distributions of all the data points for the first study. Also, for diversity of the data, we distributed the change points evenly across each subject.

VI. RELATED WORK

Since PATHIMPACT was introduced by Law and Rothermel [7], dynamic impact analyses have been refined and studied but only for their relative precision in terms of the sizes of their impact sets and their relative efficiencies [10]–[13]. In this paper, we estimated for the first time the actual precision and recall of the best existing such analysis to inform developers on their accuracy and motivate researchers to improve them.

Shortly after PATHIMPACT, which is based on compressed traces, another dynamic impact analysis called COVERAGEIMPACT was introduced by Orso and colleagues [8], which uses cheap information in the form of runtime coverage to obtain impact sets. The authors of both techniques later empirically compared the precision and efficiency of PATHIMPACT and COVERAGEIMPACT [10] and concluded that COVERAGEIMPACT is considerably less precise than PATHIMPACT although it is cheaper.

Later, the Apiwattanapong and colleagues developed the concept of execute-after sequences (EAS) to perform PATHIMPACT using an execution-length-independent amount of runtime data that was shown to be almost as cheap to obtain as the data for COVERAGEIMPACT [11]. However, the precision of the resulting technique, which we call PI/EAS, was evaluated only in terms of its impact-set sizes against COVERAGEIMPACT. In this paper, in contrast, we evaluated the accuracy of PI/EAS for predicting *actual* impacts of changes.

To improve the precision of PATHIMPACT, Breech and colleagues conceived INFLUENCEDYNAMIC [12] which adds influence mechanisms to PATHIMPACT. However, the evaluation of INFLUENCEDYNAMIC found negligible improvements over PATHIMPACT while no clear variant of EAS exists for INFLUENCEDYNAMIC to reduce the cost of tracing (dependent on execution length). Their evaluation also studied only the relative sizes of the impact sets, unlike our studies in this paper.

Hattori and colleagues formally discussed the accuracy of impact analysis [30]. However, instead of *dynamic* impact analyses, they examined the accuracy of a class-level *static* impact analysis tool introduced in that same paper based on call-graph reachability with different depth values. In contrast, in this paper, we studied *dynamic* impact analysis and we did so at the method level, which is more informative. We also used DEA for comparison instead of coarse repository data.

The MDEA technique we used for comparison is adapted from execution differencing approaches such as DEA [21] and Sieve [20]. These two approaches are *descriptive* impact analyses because they describe the effects of changes as observed at runtime before and after those changes have been applied. In contrast, in this paper, we evaluated the best *predictive* dynamic impact analysis known that can answer impact queries earlier than descriptive ones—when only potential change locations (methods) are known.

Other related dynamic impact analyses exist, such as CHIANTI [6], which is descriptive. CHIANTI compares two program versions and their dynamic call graphs to obtain the set of changes between versions and map them to affected test cases. Our studies, however, focuses on *predicted* impacts on code. Yet another technique by Goradia [31] uses dynamic impacts, but as part of an iterative process that weighs statements for better dynamic backward slicing for debugging. In all, before this paper, there have been no other accuracy studies of predictive dynamic impact analyses [5].

VII. CONCLUSION AND FUTURE WORK

In this paper, we presented a novel approach for evaluating dynamic impact analyses which we applied to PI/EAS in particular, the best in the literature. Using this approach, we

performed the first study of the predictive accuracy of dynamic impact analyses. Concretely, we estimated the accuracy (precision and recall) of PI/EAS for thousands of injected changes using our SENSA tool and 28 changes made by other researchers on Java software. The results of our studies indicate that PI/EAS can suffer from low precision or low recall, or both, and an even lower accuracy in most cases.

The accuracy levels observed for this dynamic impact analysis are likely to be lower than what developers would expect and desire. The knowledge gained in this paper therefore gives a note of caution to developers on the usefulness of dynamic impact analysis but can also inform them on how to understand and use the corresponding impact sets.

The results also motivate the need for new techniques and improvements that should be sought by researchers to provide developers with more effective alternatives that take full advantage of the attractiveness of a dynamic approach to impact analysis for representing impacts that occur in practice, in contrast with the conservative results of static analyses.

Next, we will expand our studies of PI/EAS and other predictive impact analyses to more subjects and to a greater variety of change types, including changes made by developers as found in source-code repositories. Moreover, we are developing a new technique for considerably improving the precision of dynamic impact analysis without adding runtime overhead, which usually dominates the cost, by performing a one-time static analysis of method-level dependencies. We believe that such an approach can overcome at least part of the precision problem of predictive dynamic impact analysis.

ACKNOWLEDGMENT

This work was partially supported by ONR Award N000141410037 to the University of Notre Dame.

REFERENCES

[1] J.-L. Lions, "ARIANE 5 Flight 501 Failure: Report by the Inquiry Board," European Space Agency and Centre National d'Etudes Spatiales, Tech. Rep., Jul. 1996.

[2] R. V. Carlone, M. Blair, S. Obenski, and P. Bridickas, "Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia," U.S. Government Accountability Office, Tech. Rep., Feb. 1992.

[3] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage, "When Private Keys Are Public: Results From the 2008 Debian OpenSSL Vulnerability," in *Proceedings of the ACM Conference on Internet Measurement*, Nov. 2009, pp. 15–27.

[4] S. A. Bohner and R. S. Arnold, *An introduction to software change impact analysis*. In Software Change Impact Analysis, Bohner & Arnold, Eds. IEEE Computer Society Press, pp. 1–26, Jun. 1996.

[5] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, 2012.

[6] 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.

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

[8] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging Field Data for Impact Analysis and Regression Testing," in *Proc. of joint European Software Engineering Conference and ACM Int'l Symp. on the Foundations of Software Engineering*, Sep. 2003, pp. 128–137.

[9] J. Law and G. Rothermel, "Incremental Dynamic Impact Analysis for Evolving Software Systems," in *Proceedings of the 14th International Symposium on Software Reliability Engineering*, 2003, pp. 430–441.

[10] A. Orso, T. Apiwattanapong, J. B. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *Proc. of Int'l Conf. on Softw. Eng.*, pp. 491–500.

[11] 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.

[12] B. Breech, M. Tegtmeyer, and L. Pollock, "Integrating influence mechanisms into impact analysis for increased precision," in *Proc. of IEEE Int'l Conf. on Software Maintenance*, 2006, pp. 55–65.

[13] ——, "A comparison of online and dynamic impact analysis algorithms," in *Proc. of European Conf. on Software Maintenance and Reingenering*, 2005, pp. 143–152.

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

[15] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990, pp. 246–256.

[16] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *Proc. of International Conference on Software Engineering*, May 2003, pp. 319–329.

[17] H. Cai, S. Jiang, Y. jie Zhang, Y. Zhang, and R. Santelices, "SENSA: Sensitivity analysis for quantitative change-impact prediction," University of Notre Dame, Tech. Rep. ND-CSE-13-04, April 2013.

[18] R. Santelices, Y. Zhang, S. Jiang, H. Cai, and Y. jie Zhang, "Quantitative Program Slicing: Separating Statements by Relevance," in *Proc. of IEEE/ACM Int'l Conf. on Software Engineering – New Ideas and Emerging Results track*, May 2013, pp. 1269–1272.

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

[20] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Sieve: A Tool for Automatically Detecting Variations Across Program Versions," in *Proc. of IEEE/ACM Int'l Conf. on Automated Software Engineering*, Sep. 2006, pp. 241–252.

[21] R. Santelices, M. J. Harrold, and A. Orso, "Precisely detecting runtime change interactions for evolving software," in *Proc. of Int'l Conf. on Softw. Testing, Verif. and Valid.*, Apr. 2010, pp. 429–438.

[22] W. N. Sumner and X. Zhang, "Comparative Causality: Explaining the Differences Between Executions," in *Proc. of IEEE/ACM Int'l Conf. on Software Engineering*, May 2013, pp. 272–281.

[23] 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.

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

[25] R. Santelices, Y. Zhang, H. Cai, and S. Jiang, "DUA-Forensics: a fine-grained dependence analysis and instrumentation framework based on soot," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program analysis*, 2013, pp. 13–18.

[26] D. Vysochanskij and Y. I. Petunin, "Justification of the 3σ rule for unimodal distributions," *Theory of Probability and Mathematical Statistics*, vol. 21, no. 25-36, 1980.

[27] 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.

[28] J. Voas, "PIE:A Dynamic Failure-Based Technique," *IEEE Trans. on Softw. Eng.*, 18(8):717–727, Aug. 1992.

[29] 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.

[30] L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Damasio, "On the Precision and Accuracy of Impact Analysis Techniques," in *IEEE/ACIS Int'l Conf. on Computer and Information Science*, May 2008, pp. 513–518.

[31] T. Goradia, "Dynamic impact analysis: a cost-effective technique to enforce error-propagation," in *ISSTA 93*, Jul. 1993, pp. 171–181.