Hybrid Program Dependence Approximation for Effective Dynamic Impact Prediction

Haipeng Cai

Abstract—Impact analysis determines the effects that program entities of interest, or changes to them, may have on the rest of the program for software measurement, maintenance, and evolution tasks. Dynamic impact analysis could be one major approach to impact analysis that computes smaller *impact sets* than static alternatives for concrete sets of executions. However, existing dynamic approaches often produce impact sets that are too large to be useful, hindering their adoption in practice. To address this problem, we propose to exploit *static* program dependencies to drastically prune false-positive impacts that are not exercised by the set of executions utilized by the analysis, via hybrid dependence approximation. Further, we present a novel dynamic impact analysis called DIVER which leverages both the information provided by the dependence graph and method-execution events to identify *runtime method-level* dependencies, hence dynamic impact sets, much more precisely without reducing safety and at acceptable costs. We evaluate DIVER on ten Java subjects of various sizes and application domains against both arbitrary queries covering entire programs and practical queries based on changes actually committed by developers to actively evolving software repositories. Our extensive empirical studies show that DIVER can significantly improve the precision of impact prediction, with 100–186% increase, with respect to a representative existing alternative thus provide a far more effective option for dynamic impact prediction. Following a similar rationale to DIVER, we further developed and evaluated an online dynamic impact analysis called DIVERONLINE which produces impact sets immediately upon the termination of program execution. Our results show that compared to the offline approach, for the same precision, the online approach can reduce the time by 50% on average for answering all possible queries in the given program at once albeit at the price of possibly significant increase in runtime overhead. For users interested in one specific query only, the online approach may compute the impact set for that query during runtime without much slowing down normal program operation. Further, the online analysis, which does not incur any space cost beyond the static-analysis phase, may be favored against the offline approach when trace storage and/or related file-system resource consumption becomes a serious challenge or even stopper for adopting dynamic impact prediction. Therefore, the online and offline analysis together offer complementary options to practitioners accommodating varied application/task scenarios and diverse budget constraints.

Index Terms—Static program dependence, dynamic analysis, impact prediction, online impact analysis, precision, efficiency

1 INTRODUCTION

MODERN software is increasingly complex and undergoing constant changes rapidly. Those changes, while necessary for any successful software system, also pose serious risks to its quality and reliability. Therefore, it is crucial to analyze and evolve these systems with respect to the changes efficiently and effectively. A key activity in this evolution process is *impact analysis* [1], [2], [3], [4], [5], [6], [7], [8], [9], which identifies the effects that program entities of interest or changes to those entities can have on the rest of the software. Impact analysis can be performed in different scopes, from specifications and architecture to test cases and source code [10]. Further, concerning source code, the analysis can be performed at various levels of granularity, ranging from component and class to method and statement. Also, impact-analysis techniques can be developed with either static or dynamic approaches, or combinations of both.

Different approaches to impact analysis provide different tradeoffs in accuracy, costs, and other qualities for computing *impact sets* (i.e., potentially affected entities). Static analysis can produce safe but overly-conservative impact sets [11], [12], [13]. In addition, textual and repository analysis can capture complementary developer intents from source code [2], [6], [8]. Dynamic impact analysis, in contrast, uses runtime program information such as coverage [14] or execution traces [15] to produce smaller and more focused impact sets than static analysis but only for particular inputs, hence the corresponding operational profiles, of the program [5], [16], [17]. Nevertheless, users looking for or utilizing *actualized* program behaviors, as represented by a set of executions, would prefer obtaining the execution-specific impact sets [15], making dynamic impact analysis an attractive option.

Different dynamic impact analyses also provide different cost-effectiveness tradeoffs, although most of them work at the method level. For example, COVERAGEIMPACT [14] is based on runtime coverage and ignores execution order, which makes it very efficient but also very imprecise [5]. Another technique, PATHIMPACT [15], is more precise by using execution order but is less efficient because it requires whole-program method-execution tracing [5]. For intermediate tradeoffs, optimizations of PATHIMPACT have been proposed [18], [19], [20], including an incremental version [21]. The most efficient one that preserves the precision of PATHIMPACT is based on *execute-after sequences* (EAS) [18]. We call this optimized technique PI/EAS.

Unfortunately, existing approaches including PI/EAS can still produce too many false-positive impacts [22]. An alternative would be forward dynamic slicing [23], [24], [25], but this technique works at the statement level which would have to be applied to all locations in a method, making it excessively expensive for a method-level impact analysis. At the method

• *The author is with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, 99163, USA. E-mail: hcai@eecs.wsu.edu.*

Manuscript received May 8, 2015; revised Jan 28, 2017.

level, hybrid techniques combining static and dynamic analysis, such as INFLUENCEDYNAMIC [16], have been attempted to attain better effectiveness (precision) [26], [27], [28]. However, these techniques improve precision over PI/EAS only marginally yet at a much greater cost [16]. Another hybrid technique combines runtime coverage with text analysis [3], but it works at the change-request level while relying on various sources of information that might not always be available, such as change requests and changed code entities that exhibit certain patterns, repository commits/logs and code comments amenable to data mining, natural language processing, and/or information retrieval techniques, and well-documented bug reports, etc. These data are typically only available if the development team followed good software practices. Also, it remains unclear how this technique [3] performs with more precise dynamic data.

When impact analysis is used for changes, there are two usage scenarios [1]: *predictive* impact analysis (e.g., [15], [18]) before changes are designed and *descriptive* impact analysis (e.g., [29], [30]) after changes have been made. The former type *predicts* the impacts of changing code to help developers assess risks, budget, design, and propagate changes [31], [32]. The latter type *describes* impacts after changes are applied [7], [33]. In this work, we focus on *predictive* impact analysis, which is also applicable to dependence-based tasks other than change-impact analysis.

For a predictive method-level dynamic impact analysis that relies on no any other information than the concrete set of executions that developers work on (thus are always available), PI/EAS remained the most precise technique at its level of efficiency in the literature, to the best of our knowledge. However, in an average case, at least half of the methods reported as impacted by this technique are false positives [22], [34].

In this paper, to address the imprecision of dynamic impact analysis at acceptable costs, we leverage *static* program dependencies and *dynamic* method-execution events to develop a novel hybrid dynamic impact analysis technique, called DIVER [35]. For one thing, DIVER exploits the static dependencies to guide the computation of dynamic impacts so as to improve the precision of impact prediction. At the same time, it leverages an approximation of the traditional finest-grained (heavy) static dependence model [11] while utilizing the light form of dynamic information (the method events) to achieve practical efficiency. The goal of this approximate dependence model is to provide crucial information to dynamic impact analysis that helps it discard spurious impacts that it would otherwise report. Unlike call graphs, this approximation captures interprocedural data and control dependencies among methods, like the *system dependence graph* (SDG) [11] did but much more lightly. For dynamic impact analysis, the static dependencies are computed only once for the entire program to support any number of impact-set queries afterward.

DIVER incurs the same runtime costs as PATHIMPACT but can be much more precise. PATHIMPACT uses the execution order of methods to identify all possible runtime impacts of a method m, which can be quite imprecise because, in general, not all methods executed after m are affected by m. DIVER, however, applies static dependencies to shrink the *impact set* (the set of potentially impacted methods) to only those methods that could have depended on m at runtime. Since it prunes, from impact sets computed purely based on the execution order, only methods that are definitely not exercised by the dynamic method events, DIVER improves precision over PATHIMPACT *without reducing recall*. [1] Naturally, applying the static dependence information increases the cost of querying the impact set of a method, but this cost per *query* (the method for which impacts are to be queried) can remain acceptable and multiple queries can be processed in parallel.

We implemented DIVER and applied it to ten Java subjects of up to 356K lines of code, and compared the results with those of PI/EAS as the baseline. We also implemented PI/EAS for Java with a technical improvement that addresses the recall drawbacks of its original implementation as presented in [18] in the presence of unhandled exceptions. Our implementation of both DIVER and the improved PI/EAS is available to the public for download. [2]

To thoroughly evaluate our technique, we computed impact sets of both every single method of each subject as a query (*single-method query*) and groups of methods where those in each group together as a single query (*multiple-method query*), using DIVER and PI/EAS separately. Furthermore, to estimate how effectively DIVER would perform in contrast to PI/EAS in real-world usage scenarios, we extracted queries, both single- and multiple-method ones, from three actively evolving open-source repositories (*repository query*) and compared the impact sets between the two techniques for those queries. At smaller scales, we also verified that DIVER improves precision without penalizing recall relative to PI/EAS through two validation studies.

Intuitively, an *online* dynamic impact analysis [19], [20] would have the advantage of answering queries earlier than DIVER as the analysis produces impact sets immediately when the program execution finishes. Another merit of online analysis is that it does not incur the storage and time costs of tracing as incurred by DIVER. (Accordingly, a dynamic impact analysis like DIVER and PI/EAS which records traces at runtime and computes impact sets after program execution is referred to as an *offline* dynamic impact analysis.) Thus, based on the core technique of DIVER, we further developed an online dynamic impact prediction technique DIVERONLINE for Java and evaluated it against the same ten subjects used in the DIVER evaluation. With a similar rationale for propagating impacts based on method-execution events, DIVERONLINE decomposes the DIVER impact-computation algorithm into three run-time monitors that each handles one type of method-execution events. At the core of DIVERONLINE, the online algorithm processes an event and discards it immediately afterward, thus it eliminates the need for tracing hence the cost of trace storage and time for trace processing. DIVERONLINE shares the static-analysis phase with DIVER, yet it moves the offline, impact-analysis phase in DIVER to the runtime phase where DIVERONLINE carries out the online impact-computation algorithm. We explored the pros and cons empirically of online versus offline dynamic impact prediction by comparing DIVER with DIVERONLINE in terms of efficiency. To accommodate users of varied demands, DIVERONLINE may work in two different modes: *All-in-One* computing impact sets of all possible queries in a given program, and *One-by-One* computing the impact set for a specific query only. The implementation of DIVERONLINE also is freely available in the DIVER package.

Our results confirm the imprecision of PI/EAS, whose impact sets often contain *hundreds* of methods. DIVER, in contrast, computed impact sets containing in most cases a few dozen methods only. Our findings for these subjects are dramatic: the

1. We use *recall* to represent the fraction of true impacts in the impact set.
2. Source code, demo, and documentation are at chapering.github.io/diver.

impact sets of DIVER are 35–50% the size of the impact sets of PI/EAS while remaining as *safe*. [3] Indeed, our validation studies confirm that DIVER results are safe for the underlying execution data utilized. This means that DIVER improved the precision of a representative existing technique by 100–186%. We also found that in most cases it takes reasonable storage of a few dozen megabytes while its time costs are acceptable at less than a minute per query on average, and in common cases only a couple seconds. Our evaluation of DIVERONLINE reveals that, based on the same rationale and equal amount of program information, the online approach (in the *All-in-One mode*) can be much faster than the offline analysis for computing at once all queries in the program under analysis, reducing the time cost of the latter by 5–95% (50% on average). For computing the impact set of a specific query only, the online approach can well accommodate also with its *One-by-One* mode albeit with slightly higher runtime slowdown. Moreover, in cases where trace storage causes serious challenges to DIVER, either mode of DIVERONLINE can be greatly favored or even become an enabling option for dynamic impact prediction as it does not generate or store any traces. In general, the online and offline analysis complement to each other to together offer flexible options to users for varied needs and diverse use scenarios. We have thoroughly compared the two modes of DIVERONLINE with DIVER, both qualitatively and empirically.

Through this work, we have demonstrated how dynamic impact analysis can be made much more attractive for developers to use in understanding their software and identifying potential impacts of changes for typical use scenarios. We also show that, even without specialized runtime environments [19], online dynamic impact prediction can be a much better option than offline alternatives in terms of cost-effectiveness especially when tracing and trace storage costs become challenges and/or when users would like to incur higher runtime overhead at once to compute impact sets for all queries for a program with respect to a test suite. In addition, our publicly-available implementation of DIVER, DIVERONLINE, and PI/EAS allows developers and researchers to use and study these techniques for other Java software, and develop relevant new techniques.

In summary, the main contributions of this work include:

- A new dynamic impact analysis technique DIVER and its online counterpart DIVERONLINE that combine *statement-level static* program dependencies and *method-level dynamic* execution trace to compute more precise impact sets than existing techniques, both safely and efficiently, and provide diverse options to accommodate different use scenarios.
- An open source implementation of DIVER and DIVERONLINE along with an improved version of PI/EAS which both offers representative dynamic impact analysis tools to developers and provide reusable facilities to other researchers.
- A comprehensive evaluation that shows the significant advantages of DIVER over existing options in overall cost-effectiveness, and thus demonstrates how dynamic impact analysis can be made more practically useful.
- A detailed comparative study (first of this kind to the best of our knowledge) of online versus offline dynamic impact prediction techniques that reveals the pros and cons of each approach both qualitatively and quantitatively.

3. In this paper, we regard a method in the dynamic impact set as a true positive if the method is dependent on the query via *exercised* data and/or control dependencies. Further, the impact set is *safe* if it includes all true positives and the analysis is *sound* if it always produces safe impact sets.

In the rest of this paper, we first further clarify our problem statement and motivation in Section 2, and then outline background concepts and techniques with an illustrating example in Section 3. The core techniques of DIVER and DIVERONLINE are detailed in Section 4 and key implementation issues are discussed in Section 5. Next, Sections 6 through 10 present our research questions and four empirical studies answering them. Then, we address various threats to the validity of our empirical results in Section 11. Finally, we discuss previous work related to ours in Section 12 before giving concluding remarks in Section 13.

2 PROBLEM AND MOTIVATION

Impact analysis is an integral part of software development, and its importance has been widely recognized both by researchers [36], [37] and the software industry [38], [39], [40], [41]. Yet, existing impact-analysis techniques are facing many challenges that hinder their adoption in practice [38], [40], [42], [43]. Among others, two critical challenges are the *uncertain results* often produced by existing techniques [39], [41], and the limited resources available to developers for impact analysis [41], [43].

One major cause of the challenge due to uncertain results lies in the imprecision of the analysis technique used. In light of our recent study [22], [34], current dynamic impact analyses such as PI/EAS can be too imprecise to be practically useful—on average, merely about one half of the methods it produces are actually impacted. In particular, for methods at the core of a program, PI/EAS can include in their impact sets most or all methods in the program. For example, if querying the entry method of the Apache performance-gauging application JMeter, the developer using PI/-EAS will end up inspecting all 732 methods executed by the test suite, making the analysis almost impossible to realistically adopt.

Since analyzing the effect of candidate changes is essential before applying them, more precise, hence potentially smaller, impact sets are clearly more desirable to developers, as long as the *safety* of the results is preserved (i.e., no true dynamic impacts are missed). For one thing, more precise impact analysis is crucial for the direct use of the analysis in change planning and management as false impacts may misguide developers to introduce buggy changes into existing code. In addition, smaller yet safe impact sets are more likely to be fully inspected, not only reducing potential risks of missing important impacts but also avoiding costly waste of time and other resources. The latter benefit also implicitly mitigate the other challenge that comes from the resource constraints developers are usually subject to.

Moreover, reducing false-positive impacts implied enhancing the effectiveness in various client analyses of impact analysis. For instance, when applied to regression testing [14], imprecise impact sets would end up with unnecessarily large numbers of test cases selected or prioritized, and the false-positives could also cause incorrect test selection or ranking. Similarly, in other applications such as fault localization [44] and changeability assessment [45], giving imprecise (excessively large) results apparently reduce the cost-effectiveness of such tasks. In consequence, the adoptability of imprecise impact analysis tends only to be compromised.

When studying the predictive accuracy of dynamic impact analysis [34], we found that the large imprecision of PI/EAS mainly results from its overly-conservative nature: It utilizes method execution order only to infer impact relations, whereas in general methods executed after a query are not necessarily affected by it. A natural way to reduce this conservativeness hence the

```
1public class A {
2   static int g; public int d;
3   String M1(int f, int z) {
4      int x = f+z, y = 2, h = 1;
5      if (x > y)
6         M2(x, y);
7      int r = new B().M3(h, g);
8      String s = "M3 rets " + r;
9      return s;}
10  void M2(int m, int n) {
11     int w = m - d;
12     if (w > 0)
13        n = g / w;
14     boolean b = C.M5(this);
15     System.out.print(b);}}
```

```
16public class B {
17   static short t;
18   int M3(int a, int b) {
19      int j = 0;
20      t = -4;
21      if ( a < b )
22         j = b - a;
23      return j;}
24   static double M4() {
25      int x = A.g, i = 5;
26      try {
27         A.g = x / (i + t);
28         new A().M1(i, t);
29      } catch(Exception e) { }
30      return x;}}
```

```
31public class C {
32   static boolean M5(A q) {
33      long y = q.d;
34      boolean b = B.t > y;
35      q.d = -2;
36      return b;}
37   static int M0(String[] l) {
38      int a = 0, b = -3;
39      A o = new A();
40      String s = o.M1(a, b);
41      double d = B.M4();
42      String u = s + d;
43      System.out.print(u);
44      return 0;}
45   }
```

Fig. 1: An example program E used for illustration throughout this paper.

PATHIMPACT: M0 M1 M2 M5 r r M3 r r M4 r r x	DIVER: $M0_e$ $M1_e$ $M2_e$ $M5_e$ $M2_i$ $M1_i$ $M3_e$ $M1_i$ $M0_i$ $M4_e$ $M4_i$ $M0_i$ x
PATHIMPACT impact set of M2: {M0,M1,M2,M3,M4,M5}	DIVER impact set of M2: {M2,M5}

Fig. 2: An example execution trace of E used by PATHIMPACT and the corresponding one by DIVER (top row), along with the impact set produced by the two techniques for the same example query M2 (bottom row).

consequent imprecision could be to utilize dependence analysis to guide the execution-order-based impact computation. In addition to the method execution traces, we can leverage static code dependencies to identify impact propagations across executed methods, so as to *prune* false positives from the set of methods executed after the query. For example, a method `m` invokes two methods `int f1()` and `char f2()` in sequential order to do two separate computations, and the two methods do not share any data. Then while `f2` will certainly be executed after `f1`, obviously any change made within `f1` would not affect `f2`. Yet, PI/EAS will report `f2` as potentially impacted by `f1`, which is a false positive in the PI/EAS impact set of `f1`. With static dependence analysis, this false positive would be discovered and eliminated since the analysis will reveal that `f2` is not dependent on `f1`. In general, a method that follows yet would never depend on (according to static dependence analysis) the query in the execution can be removed as a false positive, which hence improves the impact-analysis precision. Although such hybrid (i.e., combining static and dynamic analysis) approaches have been explored previously, most of them have not been able to significantly improve over PI/EAS [16], [26], [27], [28]; others targeted different application scopes [3], [28]. Nonetheless, we believe a better design would lead to significant improvements in precision without much increase in overheads, and we show how to achieve that goal through the development of DIVER.

Like DIVER, most dynamic impact prediction approaches [15], [16], [18], [27], [28], [35], [46] are *offline* in that they first collect program traces and then compute impact sets from the traces *after* program execution. Dynamic impact prediction can be performed *online* as well [19], [20], in which the impact sets are computed *during* program execution and the analysis results are immediately available as soon as the execution terminates. Intuitively, one merit of online analysis against offline analysis is that the former does not incur trace storage and trace processing costs as the latter does. The merit can be significant when such costs become substantial—serializing/deserializing large traces of long-running programs may necessitate heavy I/O (time cost) and/or large disk/database spaces (storage cost). While traces can be reused for different queries in offline analysis (i.e., on-demand querying), users could better instead produce the impact set of any possible query all at once and later pick impact sets on demand—where the online analysis can be more desirable since the otherwise extra runtime overhead due to the need for executing the program multiple times would be avoided, especially if the (time and/or storage) cost savings are appreciable. Yet, existing online dynamic impact prediction techniques [19], [20] suffer from the same imprecision as PI/EAS. Therefore, we will explore an online approach to dynamic impact prediction that achieves the level of precision at least comparable to that of DIVER.

3 BACKGROUND AND EXAMPLE

This section presents the background of this work and an illustrating example. In Figure 1, program E inputs two integers `a` and `b` in its entry method `M0`, manipulates them via `M1` and `M4` and prints the return values concatenated. `M4` updates the static variable `g` to be used by `M2` via a call to `M1`. `M3` and `M5`, invoked by `M1` and `M2`, include field accesses, conditionals, and arithmetics.

3.1 Program Dependencies

Program dependencies are classified as control or data dependencies [47]. A statement s_1 is *control dependent* [48] on a statement s_2 if a branching decision taken at statement s_2 determines whether statement s_1 is necessarily executed. In Figure 1, for example, statement 22 is control dependent on statement 21 because the decision taken at 21 determines whether 22 executes or not. This dependence is *intraprocedural* because both statements are in the same procedure (function or method). Another example is the set of statements 11, 12, 14, and 15 which all are control dependent on statement 5, whose decision determines whether `M2` is called. These dependencies are *interprocedural* [49] because each of them crosses different procedures.

A statement s_1 is *data dependent* [50] on a statement s_2 if a variable v defined (written) at s_2 is used (read) at s_1 and there is a *definition-clear path* in the program for v (i.e., a path that does not re-define v) from s_2 to s_1. In Figure 1, for example, statement 36 is data dependent on statement 34 because 34 defines

b, 36 uses b, and there is a path ⟨34,35,36⟩ that does not re-define b. Data dependencies can also be classified as intraprocedural or interprocedural. In our work, we treat formal parameters as defined at the entry of each procedure (function or method) and also as data dependent on the corresponding actual parameter at each call site for that procedure. For example, the formal parameter q at statement 32 is a definition of q and also data dependent on the actual parameter this at 14.

Incorporating both types of program dependencies together, a dependence graph [11], [48] is a *static* program representation where nodes represent statements of the program and edges represent both control and data dependencies among those statements. In comparison, the *program dependence graph* (PDG) [48] addresses dependencies of single procedures (methods, functions), while the SDG [11] models dependencies of a program consisting of multiple procedures.

3.2 Dynamic Impact Analysis

Dynamic impact analysis uses execution data to find the runtime impacts that program entities such as methods or changes to those entities have on the program. In this paper, we focus on *predictive* impact analysis [1], [15] which has no knowledge of any actual changes to the program. Such a dynamic impact analysis takes a program P, a test suite T, and a set of methods M as inputs, and outputs an *impact set* containing the methods in P to be potentially impacted by M when running T on P.

One example technique is PATHIMPACT [5], [15], which collects runtime traces of executed methods. For each method m in M that is queried for its impact set, PATHIMPACT uses for impact prediction the method execution order found in the runtime traces of P for T. The analysis identifies as impacted m and all methods executed in any trace after m. Thus, the impact set includes all methods called directly or transitively from m, all methods below m in the call stack (those into which the program returns after m), and all methods called directly or transitively from those in the call stack below m.

Figure 2 shows an example trace for PATHIMPACT, where r is a method-return event and x the program-exit event. The remaining marks indicate the entry events of methods. For query M2, for example, PATHIMPACT first finds {M5, M3, M4} as impacted because these methods were entered after M2 was entered and then finds {M0, M1} because these methods only return after M2 was entered. Thus, the resulting impact set of M2 is {M0, M1, M2, M3, M4, M5} for this trace. In cases where more than one trace exists, PATHIMPACT returns the union of the impact sets for all traces.

Method traces are much shorter than statement traces. Also, traces can be compressed. Nevertheless, the *execute-after sequences* (EAS) optimization [18] exists to reduce the space costs of PATHIMPACT without losing essential information needed by the analysis. This approach exploits the observation that only the first and last occurrence of each method in a trace, captured by the first entry event (denoted by e) and last returned-into event (denoted by i), respectively, are required. The resulting technique, PI/EAS, keeps track at runtime of those two events (occurrences) per method without collecting full traces.

Another example technique is INFLUENCEDYNAMIC [16], which utilizes interface-level data dependencies in addition to method execution order to improve impact-analysis precision against PI/EAS. The technique first builds the *influence graph* of the input program, via which parameters and return values passed between methods are represented. Next, it checks dependencies from the graph while traversing method execution sequence to identify potentially impacted methods of a given query. As an example, for the same inputs (query, program, and trace) as above, INFLUENCEDYNAMIC will report the whole program as the resulting impact set as PI/EAS did. In all, this technique was shown only marginally (3–4%) more precise but much (10x) more expensive than PI/EAS [16].

Finally, dynamic slicing [24], in its forward form, could be an option for dynamic impact analysis. And as it performs the finest-grained dependencies at statement level, this technique can produce impact sets of much higher precision than PI/EAS and INFLUENCEDYNAMIC. However, for a method-level impact analysis, dynamic slicing tends to be overly heavyweight in nature and would incur excessive overheads.

In sum, despite its age, PI/EAS remains the most cost-effective technique prior to DIVER [35] for predictive dynamic impact analysis that relies on no extra inputs (but P, T, and M only which should always be available to users adopting a dynamic analysis). Therefore, we choose PI/EAS to represent existing alternatives to our technique for comparative studies.

4 TECHNIQUES

We now present our new approaches to dynamic impact analysis. First, we give an overview of the DIVER approach using an example, focusing on the overall process of our analysis with DIVER. Next, we describe the approximate static dependence model underlying DIVER, with emphasis on the construction of the dependence model. Then, we present in detail the impact computation algorithm at the core of DIVER, which shows how the static dependencies are exploited to prune false-positive impacts from execute-after sets of methods in the method-event trace. We also present DIVERONLINE, our online version of DIVER, including a comparison between offline and online dynamic impact prediction and the online impact computation algorithm of DIVERONLINE. Finally, we clarify the practical applications of our techniques and discuss their limitations.

4.1 Overview

For our new impact analysis DIVER to be *safe* with respect to an execution set and also precise and practical, we need something better than the *execute-after* relation of PI/EAS, which is too conservative. Reaching a method m' after a method m at runtime is necessary for m to impact m', but not all such methods as m' necessarily depend on m. To fix this problem, we build first a whole-program static dependence graph (referred to hereafter as *dependence graph* for brevity), and then use it to find which of those methods dynamically depend on m. This dependence graph is more conservative but lighter-weight than traditional full dependence models [11], [51] it approximates. And it is computed *only once* (before running the program) to be reused for all possible impact-set queries with respect to the single program version under analysis and any executions of that version.

4.1.1 Process

The process of computing the dependence graph and using it for dynamic impact analysis is shown in Figure 3. It works in three phases: static analysis, runtime, and post-processing. The inputs for the entire process are a program P, a test suite (execution set) T, and a query set M. To optimize the static-analysis phase, the

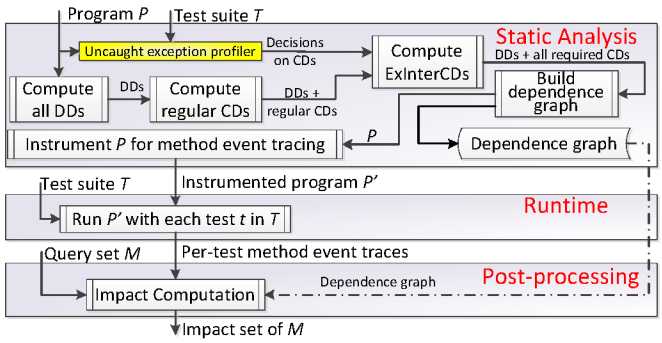


Fig. 3: The overall process flow and three phases of DIVER.

process first runs a profiler which executes T on P to quickly find whether any exceptions are raised by a method but not caught there or not caught at all. This lets the static-analysis phase determine whether it can *safely skip* computing some control dependencies caused by unhandled exceptions for this particular input set T.

After profiling, the static-analysis phase computes data dependencies (DD) and control dependencies (CD). For the CDs, it first computes *regular* CDs caused by branches, polymorphic calls, and intraprocedural exception control-flows. For the remaining exception control-flows [49], [52] (*ExInterCDs* in Figure 3), the process computes the CDs for the exception types that the profiler detected as not handled by the originating methods.

After all dependencies are found, the dependence graph is built. By design, it is passed directly to the post-processing phase and *not used at runtime*. The motivation is to ensure that the runtime phase only collects method traces, just like PATHIMPACT, without other runtime costs. The post-processing will then use the dependence graph to detect from these traces which methods that are executed after a method m actually depended on m, unlike PATHIMPACT which reports all methods executed after m.

For the runtime phase, the static analysis creates the instrumented version P' of P using only probes for monitoring method-entry and returned-into events. The instrumented program P' is similar to that of PI/EAS except that, instead of tracking two integers per method, it traces the *whole* sequence of method events, as the original PATHIMPACT does, because DIVER needs entire traces to determine transitive dependencies at post-processing. Nevertheless, DIVER compresses these traces on the fly at reasonable extra costs to reduce trace storage. In sum, the runtime phase executes P' with test suite T to produce one compressed method-level trace per test case t in T.

The post-processing phase lets the user query the impact set of a method set M. Any number of queries can be made at this point without re-running the previous phases. For a query on method set M, DIVER uses the dependence graph from the static phase and the traces of P' for T from the runtime phase to identify all methods that depended directly or transitively on any method in M on any of those traces. M is included in the result because every method impacts itself.

4.1.2 Precision

The dependence graph underlying DIVER keeps statement-level dependence information that specifies dependencies among methods, including both intraprocedural and interprocedural dependencies. This information is essential for DIVER to precisely identify potential impacts: the interprocedural dependencies represent interface-level impact relations among methods, similar to those in the influence graph used by INFLUENCEDYNAMIC [16]; while the intraprocedural dependencies help DIVER find whether a method is *transitively impacted* by another.

For example, if a method m'' dynamically depends only on m' via interprocedural edge d' and m' dynamically depends only on m via interprocedural edge d, then m'' dynamically depends on m if and only if d' is directly or transitively dependent on d via *intraprocedural* dependencies (edges) inside m' . Without this information, if d' does not depend on d, m'' would be mistakenly reported as impacted by m.

4.1.3 Example

To illustrate, consider the trace for DIVER from input ⟨*a=0, b=-3*⟩ in Figure 2 where subscripts e and i denote the entry and returned-into events, respectively. For a query set M = {M2}, DIVER traverses the trace to find which dependencies (from the dependence graph) are exercised due to methods executed after M2 and, via those dependencies, which methods depended directly or transitively on any occurrence of M2. When DIVER finds M2, the impact set starts as {M2}. Then, the only outgoing dependence from M2 in the graph is exercised because its target M5 occurs next, so M5 is impacted. Thus, DIVER finds the impact set {M2, M5}, in contrast with PATHIMPACT which reports {M0, M1, M2, M3, M4, M5} (i.e., the entire program E).

4.2 Dependence Graph and Impact Propagation

The static dependence graph of the entire program is a key ingredient of our technique. While it is a statement-level program representation where each node represents a statement and each edge a dependence of one statement on another as in a SDG [11], this graph is a conservative approximation of the SDG that attempts to be lighter-weight hence more efficient. For one thing, building a full fine-grained dependence model like SDG would impede the scalability of our technique. On the other hand, we need to capture only key dependence information necessary for *method-level* dynamic impact analysis, for which an *approximate* dependence graph is sufficient.

Specifically, similar to the SDG, the dependence graph built in the static phase of DIVER includes the entire PDG [48] for each individual procedure (method). Thus, creating the DIVER dependence graph incurs the same costs of intraprocedural dependence analysis as building the SDG. On the other hand, unlike the SDG, the DIVER dependence graph approximates, and differentiates itself from, the SDG by dismissing calling-contexts when computing transitive interprocedural dependencies due to procedure (method) calls. The motivation is that DIVER is dynamic after all and, thus, it does not require context-sensitive analysis: DIVER uses this graph only to prune runtime traces. Consequently, the DIVER dependence graph does not include summary edges [11], and DIVER thus avoids the costs due to context-sensitivity during its static analysis, relative to building a full SDG.

Furthermore, *interprocedural* DDs in the dependence graph are classified into three types: *parameter* DDs from actual to formal parameters in method calls, *return* DDs from return statements to caller sites, and *heap* DDs from definitions to uses of *heap variables* (i.e., dynamically-allocated variables not passed or returned explicitly by methods). Accordingly, corresponding edges in the dependence graph are referred to as *parameter*, *return*, and *heap* edges. Importantly, *parameter* and *return* DDs are exercised

at runtime only if the target method executes *immediately* after the source. Thus, the type of a DD lets DIVER at post-processing decide whether the dependence was exercised and the target method of that dependence was impacted by the source.

To facilitate our presentation of DIVER, we refer to the specific target statements of incoming interprocedural dependence edges to, and the source statements of outgoing interprocedural edges from, a method as *incoming ports* (*IP*s) and *outgoing ports* (*OP*s) of that method, respectively. An impact propagating to a method via an incoming edge e will enter the method through the *IP* for e. If an impact propagates beyond this method through outgoing edges, it will exit through all *OP*s that are *reachable via intraprocedural edges* from the *IP* for e. An impact that originates in a method will propagate through *all OP*s of that method.

4.3 Impact Computation

In the post-processing phase of DIVER, queries for impact sets are answered using the dependence graph from the static-analysis phase and the traces produced by the runtime phase. Algorithm 1 formalizes this process, which traverses traces to find methods dynamically dependent on a given query directly or transitively.

The method traces indicate the order in which methods executed and, based on the types of the interprocedural edges in the dependence graph, if the fact that a method m' executed after an impacted method m in a trace implies that m' will be impacted by m. Whether this impact occurs depends on (1) the existence of an edge $\vec{e}$ from m to m' in the dependence graph, (2) that the source (outgoing port) of $\vec{e}$ has already been identified as impacted, and (3) the type of $\vec{e}$, which may constrain m and m' to be consecutive in the trace (e.g., for *parameter* edges).

Algorithm 1 : COMPIS(Dependence graph G, trace L, method c)

```
1: ImpOPs := ∅   // map of edge type to set of impacted OPs
2: ImpactSet := {c} // impact set of c
3: start := false, pm := null // preceding method occurrence
4: for each method event e∈L do
5:     if ¬ start then {start := m(e) = c; if ¬start then continue}
6:     if e is a method-entry event then
7:         if m(e)=c then
8:             for each outgoing edge oe from n(m(e)) in G do
9:                 ImpOPs[type(oe)] ∪= {src(oe)}
10:            pm := m(e) // method occurrence; continue
11:        for each incoming edge ie to n(m(e)) in G do
12:            if type(ie)=return∨src(ie)∉ImpOPs[type(ie)] then
13:                continue
14:            ImpactSet ∪= {m(e)}
15:            for each outgoing edge oe from n(m(e)) in G do
16:                if src(oe) is reachable from tgt(ie) in G then
17:                    ImpOPs[type(oe)] ∪= {src(oe)}
18:    else  // e is a method-returned-into event
19:        for each incoming edge ie to n(m(e)) in G do
20:            if type(ie)=parameter∨src(ie)∉ImpOPs[type(ie)] then
21:                continue
22:            ImpactSet ∪= {m(e)}
23:            for each outgoing edge oe from n(m(e)) in G do
24:                if src(oe) is reachable from tgt(ie) in G then
25:                    ImpOPs[type(oe)] ∪= {src(oe)}
26:    if pm=m(e) then {continue}
27:    for each edge type t ∈{parameter, return} do
28:        ImpOPs[t] \= {z |z ∈ ImpOPs[t] ∧ m(z) = pm}
29:    pm := m(e) // preceding method occurrence
30: return ImpactSet
```

The algorithm inputs a dependence graph G, an execution trace L and a queried method c, and outputs the impact set of c. It uses the following notations: *m(e)* gives the method associated with a method event e; *n(m)* is the set of dependence-graph nodes for all statements in method m; *m(z)* is the method to which port z belongs; *src(d)*, *tgt(d)*, and *type(d)* are the source node, target node, and type of edge d, respectively.

To maximize precision, an interprocedural edge d exercised for the i^{th} time in the trace propagates an impact to its target (*IP* port) only if the source (*OP* port) of d for that i^{th} occurrence has also been impacted. To that end, an impacted *OP* set per edge type, which starts empty at line 1, is maintained at lines 9, 17, and 25. These sets track impact propagations on ports to ensure that only the methods transitively reachable from c through impacted ports are reported as impacted. The impact set starts with the queried method c (line 2) and grows as the trace is traversed (lines 4–29).

Methods executed before the first occurrence of c cannot be impacted, so their events are skipped using a flag *start* (lines 3 and 5). Methods executed after c are checked to determine if they are impacted, for which two key decisions are made. First, the algorithm decides that the impact of c *propagates into* a method *m(e)* if there is an impacted port in *ImpOPs* that is the source of an interprocedural edge of the same type to *m(e)* (lines 11–14 for method-entry events and lines 19–22 for returned-into events).

The second key decision is to determine whether an impact *propagates out* of *m(e)* by finding the *OP* ports of *m(e)* that are reachable, via *intraprocedural* edges inside the method, from the *impacted IP* ports of that method (i.e., the target ports of impact-propagating edges according to the first decision). Those impacted *OP*s are added to *ImpOPs* to continue looking for impacts in the rest of the trace (lines 15–17 for method-entry events and lines 23–25 for returned-into events). As for the queried method c, all of its *OP*s are added to *ImpOPs* when c executes (lines 7–10).

To determine impact propagations through interprocedural edges on the dependence graph, those edges are classified into two categories, as described next, such that all edges in each category share the same propagation rules.

Adjacent edge. DD edges of types *parameter* and *return* are classified as *adjacent* edges. An adjacent edge from method m to method m' models an interprocedural DD between the two methods. Through these edges, an impact can propagate from m to m' only if m' executes immediately after m. To realize this rule, an *OP* z that is the source of an adjacent edge is added to the impacted *OP* set *ImpOPs*, as other impacted *OP*s, when found to propagate the impact beyond *m(z)* in the trace. The port is then removed from that set (lines 26–28) after matching it to an *IP* in the immediate caller or callee when an event for a different method *occurrence* is found, because the corresponding parameter or return value should not be matched to any other *IP*s of methods that occur later in the trace. The method occurrence is tracked by *pm*, initialized at line 3 and updated at lines 10 and 29.

Execute-anytime-after edge. All other interprocedural edges are *execute-anytime-after* edges. Such an edge from m to m' models an interprocedural CD or *heap* DD between these two methods such that an impact in m propagates to m' if and only if m' executes *anytime* after m in the trace. Such edges propagate impacts to their targets as long as their sources (*OP*s) are *impacted* when the targets are reached later. Thus, the sources of these edges are never removed from the impacted *OP* set *ImpOPs* once added to that set (lines 27–28 ignore these edge types).

It is worth noting that the way in which propagation rules are applied depends on the type of method event being processed in the trace. For instance, no *return* edges are considered for impact propagation at method-entry events (lines 12–13) and no *parameter* edges are considered at returned-into events (lines 20–21) because of the semantics of those event types (see Section 3). Also, all *OP*s of the queried method c are marked as impacted at each entry event found for c. Thus, it is not necessary to do the same for the returned-into events of c because the *OP*s of c are already marked as propagated at the entry of c in the trace.

In sum, according to the propagation rules of DIVER, for each event in the trace, the method associated with that event is added to the impact set if it is determined that at least one of its *IP*s is directly or transitively impacted by the queried method. After all events of the trace are processed in order, the algorithm returns as its output the resulting impact set for that trace (line 30). If multiple traces are available, one run of the algorithm per trace is required and the result is the union of the individual impact sets. Also, for computing the impact set of multiple methods, the algorithm can be run once per method, or can be easily adjusted to treat c as a set of those queried methods for efficiency.

4.4 Online Impact Prediction

DIVER computes the impact set for any queries in the third phase by reusing the static dependencies and dynamic method-event traces from the previous two phases (see Section 4.1.1). As such, DIVER is an *offline* impact analysis. Yet, as we mentioned earlier (Section 2), *online* analysis could be more desirable in some usage scenarios when its cost savings over offline analysis are significant.

As follows, we first further motivate the development of our online analysis, called DIVERONLINE, by qualitatively comparing online versus offline dynamic impact prediction in general. Then, we describe the overall workflow of DIVERONLINE in relation to that of DIVER, followed by the detailed presentation of the online impact computation algorithm at the core of DIVERONLINE.

4.4.1 Online versus Offline: Qualitative Comparison

One way to compare these two classes of analysis is to look at the merits of each that are likely the drawbacks of the other in the same regard. Without empirical evidences, the comparison is mostly hypothetical for now. We will validate the qualitative understanding later through quantitative assessment in the evaluation section (Section 10).

Merits of online analysis. An *online analysis avoids the cost of execution tracing that an offline analysis would incur, including the space cost for storing traces and concomitant, extra I/O overheads in the post-processing phase.* Therefore, intuitively, for computing all possible impact-set queries in an one-off execution (hence avoiding to run the program multiple times), an *online analysis can be considerably faster than an offline analysis*. Although the online analysis incurs the time cost of impact computation that the offline analysis would avoid in the runtime phase, that extra cost may be counteracted by its avoidance of the tracing cost incurred by offline analyses. Thus, ultimately an online analysis do not necessarily cause much higher run-time overheads than offline alternatives. In all, the *online analysis would better fit use scenarios where it answers queries faster and/or the trace storage cost of an offline analysis becomes substantial or even a stopper for practical adoption.* To facilitate discussion here and evaluation later on, we assume that the online analysis computes the impact sets for all queries at once by default, which we refer to as its *All-in-One* mode.

Merits of offline analysis. On the other hand, computing all the impact sets regardless of which one of them is needed would be apparently wasteful if the user is just interested in the impact set of one specific query. To better accommodate this use scenario, the online analysis may just answer one query in an one-off program execution, which we refer to as its *One-by-One* mode. However, with this mode users would have to run the analysis (and the program) repeatedly for multiple queries, incurring the run-time overhead of the analysis multiple times also. In contrast, the *offline analysis incurs the run-time overhead only once* and answers any queries *on demand* based on the collected traces without rerunning the program. Also, *with the offline approach, users do not have to know the queries before the run time*. To overcome this comparative limitation, the online analysis may work in the *All-in-One* mode as described above. In that mode, the online analysis computes the impact sets for all possible (single-method) queries of the program under analysis and then also answers queries *on demand*—results for multiple-method queries can be simply derived from the single-method query results for each member method through iterative set-union operations. Since the offline analysis only needs one-off executions and readily reuses traces for on-demand impact computation, we only develop and focus on its *One-by-One* mode.

Online analysis: All-in-One versus One-by-One . We expect the online analysis to be generally more expensive when working in the *One-by-One* mode than in the *All-in-One* mode due to the extra run-time overheads (proportional to the total number of queries) in the former. The *One-by-One* online analysis has its potential merits, though, in comparison to the analysis in its *All-in-One* mode: (1) the *One-by-One* analysis would incur likely much shorter total time in use scenarios where users are just interested in ad-hoc impact analysis for only a few times or just interested in a particular query in mind; (2) in use scenarios where the original program executions (i.e., without instrumentation) take relatively long time and/or consume large memory, the *One-by-One* analysis could be an enabling option: computing the impact sets of all queries at once especially when the queries are in large numbers may blow up memory or make the analysis unbearably slow (e.g., by awaking the paging process of the operating system).

In sum, the online and offline dynamic impact analysis each has its own (unique) pros and cons, and they appear to be *complimentary* to each other, in terms of efficiency and ability to accommodate various, particular use scenarios. Thus, providing both modalities would give more *flexibility* to users hence increases the adoptability of our techniques.

4.4.2 Analysis Algorithm

The overall workflow of DIVERONLINE is similar to the process of DIVER as shown in Figure 3. Specifically, the static analysis phase is shared by the two techniques. The main differences are that (1) the dependence graph constructed in the first phase flows to the runtime phase of DIVERONLINE for the online impact computation, (2) the runtime phase of DIVERONLINE takes the query set M as an input, computes the impact set for M while executing the instrumented program P', and produces the impact set as the output, and accordingly (3) DIVERONLINE does not produce any traces in the runtime phase and no longer has the post-processing phase. By design, DIVERONLINE essentially uses the

same static and dynamic program information as DIVER does and, for a given query set, both techniques produce the same impact set.

Algorithm 2 gives the pseudo code of the *All-in-One* online analysis algorithm reusing the notations from Algorithm 1. For better efficiency, we use bit vectors (*BitSet*) to maintain impacted ports and the impact set for each method. We create a global index *midxs* for the full set M_P of methods in P and use the integer identifier *midxs[m]* for a method m to mark the impact status associated with m in relevant bit vectors. In addition, given a map structure d, *keys(d)* returns the set of keys of d.

The algorithm takes the dependence graph G passed to it from the static analysis phase as the input, and returns the map *ImpactSets* from each method in M_P to the bit vector representing its impact set as the output. The two key components of the algorithm, ONENTER and ONRETURNINTO, are the two run-time monitors for method-entry and method-returned-into events, respectively—they are invoked upon each of such events for each executed method. For each program execution, ONINITIALIZE is invoked only once when the program starts executing—right before the first method execution; ONTERMINATE is also invoked only once, when the program terminates—right after the last method execution. In contrast, DIVER contains these four run-time monitors too but uses them for recording method event traces.

Prior to online impact computation, the algorithm initializes two hashmaps (lines 2–3) to maintain the impacted *OP*s per edge type and the impact set for each method in M_P (i.e., each possible single-method query with respect to the input program P). As in DIVER, it also keeps track of the previously executed method (relative to the method being monitored) with *pm* (line 4). The rationale of marking the impact status of ports for propagating impacts within methods via static intraprocedural dependencies and across method via interprocedural dependencies (lines 9–10, 16–18, and 31–33) is similar to Algorithm 1, so is the trivial inclusion of a method itself in its impact set when the method first executes (lines 6–8). The key difference lies in the need of doing that for all methods that have executed at least once (lines 11 and 26) instead of for the given single query only as in Algorithm 1.

When a method m is entered (line 5), the algorithm checks for each executed method x (loop of 11–18) whether the impact originating from x has propagated to m through impacted ports and incoming edges targeting any nodes in m (lines 12–14) and, if so, adds m to the impact set of x (line 15) and continues to propagate the impact out of m (loop of 16–18). Those impacted ports of each executed method, except for m (line 21), that expect to propagate the impact to a method other than m via adjacent edges (line 22) will be removed (line 23) to avoid false-positive impacts since such edges can propagate impacts only one method away. When the execution is returned into m (line 25), the process is similar but *parameter* edges are dismissed for impact propagation (line 28). As the program execution terminates (line 40), *ImpactSets* contains the impact sets for all executed methods (line 41). To obtain the impact set of a specific query c on demand, DIVERONLINE simply traverses the bit vector *ImpactSets[midxs[c]]* associated with that query to collect the impacted methods by looking up the global method index *midxs*.

The algorithm for the *One-by-One* mode of DIVERONLINE is the same in structure as Algorithm 2, having the same four run-time monitors. However, the specific, known query is an additional input and the algorithm maintains the set of impacted ports and impact set for only that query. Also, all method events before the first event of the query will be ignored. Thus, the algorithm itself is very similar to Algorithm 1: the ONINITIALIZE component consists of lines 1–3, the ONENTER and ONRETURNINTO monitors correspond to the process upon method entry and returned-into events, respectively, with both containing lines 26–29, and the ONTERMINATE simply returns the result *ImpactSet* (line 30). We omit the pseudo code due to the similarity.

Algorithm 2 : ONLINEIMPACTCOMPUTE(Dependence graph G)

```
1: function ONINITIALIZE(void) // upon program start
2:     ImOPs:= ∅   // map of edge type to impacted OPs per method
3:     ImpactSets:= ∅  // map of each method to its impact set
4:     pm := null  // preceding method occurrence
5: function ONENTER(Method m) // upon entering method m
6:     if midxs[m]∉keys(ImpactSets) then
7:         ImpactSets[midxs[m]] := new BitSet(|M_P|)
8:         ImpactSets[midxs[m]].set(midxs[m])
9:     for each outgoing edge oe from n(m) in G do
10:        ImOPs[midxs[m]][type(oe)] ∪= {src(oe)}
11:    for each method x in keys(ImOPs) do
12:        for each incoming edge ie to n(m) in G do
13:            if type(ie)=return∨src(ie)∉ImOPs[x][type(ie)] then
14:                continue
15:            ImpactSets[x].set(midxs[m])
16:            for each outgoing edge oe from n(m) in G do
17:                if src(oe) is reachable from tgt(ie) in G then
18:                    ImOPs[x][type(oe)] ∪= {src(oe)}
19:    if pm=m then {continue}
20:    for each method x in keys(ImOPs) do
21:        if x=midxs[m] then {continue}
22:        for each edge type t ∈{parameter, return} do
23:            ImOPs[x][t] \= {z |z ∈ ImOPs[x][t] ∧ m(z) = pm}
24:    pm := m // preceding method occurrence
25: function ONRETURNINTO(Method m) // upon returning into m
26:    for each method x in keys(ImOPs) do
27:        for each incoming edge ie to n(m) in G do
28:            if type(ie)=parameter∨src(ie)∉ImOPs[x][type(ie)] then
29:                continue
30:            ImpactSets[x].set(midxs[m])
31:            for each outgoing edge oe from n(m) in G do
32:                if src(oe) is reachable from tgt(ie) in G then
33:                    ImOPs[x][type(oe)] ∪= {src(oe)}
34:    if pm=m then {continue}
35:    for each method x in keys(ImOPs) do
36:        if x=midxs[m] then {continue}
37:        for each edge type t ∈{parameter, return} do
38:            ImOPs[x][t] \= {z |z ∈ ImOPs[x][t] ∧ m(z) = pm}
39:    pm := m // preceding method occurrence
40: function ONTERMINATE(void) // upon program termination
41:    return ImpactSets
```

4.5 Applications and Limitations

In contrast to descriptive impact analysis (e.g., [7], [29]), impact prediction enables a proactive approach to reliable software evolution—instead of leaving developers to deal with the aftermath resulting from changes that broke the software, it helps developers identify and understand possible change effects *beforehand* so as to avoid applying failure-inducing changes. Further, compared to static impact prediction (e.g., [12], [43], [53]), dynamic impact prediction is more useful when the developers focus on a specific set of program executions thus are concerned about the impact of potential change locations for those specific, rather than all possible, executions. The resulting dynamic impact

sets can be used not only for planning changes but also for understanding the program behaviour with respect to the concrete executions. In fact, such understandings have even broader applications than program comprehension. For example, decisions in regression-test selection and/or prioritization can benefit much more from the *dynamic* impact sets specific to the executions generated from relevant regression tests than static impact sets (e.g., [14], [54]). As another example, when debugging, developers need to concentrate on the faulty executions instead of all potential program executions. Narrowing down the search space to the *dynamic* impact set (with respect to those faulty executions) of responsible change locations can greatly facilitate the fault-localization process (e.g., [55], [56]). The approaches we proposed in this paper aim at more precise dynamic impact sets than existing peers to enhance the effectiveness of all these applications of dynamic impact prediction. Moreover, our techniques are particularly attractive when only the program code (bytecode, rather than source code) and executions/inputs of interest are available without any additional artifacts (e.g., design specification and repository information).

The merits of dynamic impact prediction above come with related constraints, though. First, our techniques are applicable only to the scenarios where the specific program inputs or concrete executions are indeed available for the analysis. Second, how well the inputs/executions cover and represent the behaviours of the program under analysis may substantially affect the quality and usefulness of the resulting impact sets. In general, the more representative of the program behaviours the inputs/executions are, the more effective the dynamic impact sets are for the applications. In addition, since the concrete executions utilized are unlikely to capture all possible program behaviours, the dynamic prediction results hold for those particular executions but may not for other executions. Note that these limitation are generally shared by any dynamic analysis [57] rather than unique to our techniques.

5 IMPLEMENTATION

In this section, we describe key aspects of our implementation of PI/EAS, the baseline approach to compare with ours, in addition to that of our techniques DIVER and DIVERONLINE.

5.1 Exception Handling in PI/EAS

The original description of PI/EAS [18] deals with exceptions handled in the raising method or its caller. However, if neither method handles the exception at runtime, the *returned-into* events for all methods in the call stack that do not handle the exception will not be logged and those methods can be mistakenly missing in the resulting impact set. To illustrate, consider the program of Figure 1 with `M2` as the query. If an exception is raised at `M5`, which is called by `M2` but never caught thereafter, the integers used by PI/EAS to record the *last* returned-into events for `M1` and `M0` will not be updated to reflect that these methods *executed after* `M2`. Thus, `M1` and `M0` will be missing from the impact set of `M2`.

To address this problem, we implemented a *corrected* version of PI/EAS, which we call PI/EAS$_C$. PI/EAS$_C$ captures all returned-into events by wrapping the entire body of each method in a try-catch block to identify uncaught exceptions. The added catch block, when reached by such an exception, creates the corresponding returned-into event (which would be missed otherwise) and then *re-throws* the exception to continue the execution as originally intended to preserve the semantics of the program.

5.2 DIVER

To build the dependence graph, we used our dependence-analysis library DUA-FORENSICS [58]. For exceptional control dependencies, our implementation takes the *exceptional control flow graph* (ExCFG) provided by Soot [59] and applies both the classical algorithm for control-dependence computation [48] and the extended algorithm for interprocedural control dependencies [49]. In addition, we adapted multi-headed and multi-tailed ExCFGs by adding virtual *start* and *end* nodes joining all head and tail nodes, respectively. Also, many types of programs, such as service daemons, contain infinite loops [60] which result in *tailless* ExCFGs. Thus, we treated all jumping statements for the outermost infinite loops as exit nodes. This treatment allowed us to directly apply existing control-dependence analysis algorithms.

When computing interprocedural exception CDs, DIVER includes in the *throwable* set of each ExCFG node all exceptions, both *checked* (declared) and *unchecked* (undeclared) for that method, thrown by that node via a throw instruction or a method it calls that throws unhandled exceptions. DIVER also reuses the method-event monitoring we implemented for PI/EAS$_C$, although it keeps the *whole* traces it needs (just like PATHIMPACT) instead of the two integers per method only needed by PI/EAS$_C$.

5.3 DIVERONLINE

DIVERONLINE reused the static analysis component of DIVER, including the dependence graph construction and the instrumentation for method event monitoring. The four event monitors of DIVERONLINE for the *All-in-One* mode are implemented as per Algorithm 2. For the *One-by-One* mode, the four monitors are implemented primarily by refactoring the impact computation component of DIVER as per Algorithm 1 as described in Section 4.4. The dependence graph is loaded (deserialized) at the beginning of the runtime phase, as is the global method index created as an extra step of the static analysis phase and serialized as a separate file. Processing each method event upon its occurrence immediately, according to our experience, can drag the analysis significantly. Thus, we created a buffer of events that are loaded in the ONENTER and ONRETURNINTO monitors, and then in the monitor ONTERMINATE the events are processed in a batch manner. As a result, the online analysis was greatly accelerated thanks to the reduction in context switching (between the execution of the original program and that of our run-time monitors) and cache misses. Our current implementation used a heap buffer of 4MB (for buffering 1M events).

Note that DIVERONLINE does not record the method events or produce any traces, nor does it have a post-processing phase as DIVER and PI/EAS have. For the *All-in-One* mode, though, DIVERONLINE contains a helper script to facilitate looking up the results (i.e., the impact sets of all possible queries) for the impact set of a specific query on demand. The *One-by-One* mode does not need such a script since it just outputs the impact set for the specific query, which is passed as a (command-line) parameter to the runtime phase.

6 RESEARCH QUESTIONS

This section introduces our empirical evaluation of DIVER with PI/EAS$_C$ as the baseline, using the implementation of the two techniques as described in Section 5. The goal was to assess the effectiveness of this new technique as well as its efficiency. In

addition, we explored the performance of DIVERONLINE versus DIVER to empirically examine the pros and cons of online dynamic impact prediction as we qualitatively discussed in Section 4.4. Accordingly, we formulated five research questions:

RQ1 How effective is DIVER with respect to existing method-level dynamic impact analyses for arbitrary queries?

RQ2 Is DIVER practical to use in terms of time and space costs?

RQ3 What are the effects of query sizes (the number of methods in one query) on the effectiveness of DIVER?

RQ4 How effective would DIVER be for repository-based queries in practical application scenarios compared to its effectiveness for arbitrary queries?

RQ5 Is online dynamic impact prediction with DIVERONLINE efficient in comparison to its offline counterpart DIVER?

To answer these questions, we conducted four empirical studies. Our first two studies aim at the effectiveness of DIVER with respect to arbitrary changes limited to single methods and across multiple methods (RQ1 and RQ3), respectively, in addition to gauging the practicality of DIVER in terms of its efficiency (RQ2). The third study focuses on using real changes actually made by developers to examine the practical effectiveness of DIVER (RQ4), complementary to the first two studies. For studies on arbitrary changes, we compare DIVER against the baseline for any possible changes to be made in a single method for each query (referred to as *single-method query*) or in multiple methods as one query (referred to as *multiple-method query*). For the third study, we regard all methods actually changed in one commit as a query (referred to as *repository query*), which can be either a single- or multiple-method query. The fourth study focuses on the efficiency of DIVERONLINE in contrast to DIVER in terms of the time cost of impact computation and runtime slowdown (RQ5). All our studies were performed consistently on a Red Hat Linux workstation that is configured with a Quad-core Intel Core i5-2400 3.10GHz processor and 4GB RAM.

Recall. Since a method would not be impacted by a given query if that method never executed after the query, the impact sets produced by PI/EAS$_C$ are always safe because they include conservatively any methods that executed after the query. DIVER improves the precision of PI/EAS$_C$ by pruning from executed-after-query methods (i.e., PI/EAS$_C$ impact sets) that are not data/control dependent on the query. Importantly, since DIVER prunes a method only if there was no any *exercised* data or control dependencies of that method on the query, it only removes false positives but no true positives from the PI/EAS$_C$ impact set with respect to the execution examined. Thus DIVER impact sets still include all methods that could actually be impacted through data/control dependencies in the utilized executions.

In sum, results (impact sets) from DIVER and PI/EAS$_C$ are always guaranteed to hold for the analyzed executions, hence both techniques are *soundy* [61]—we consider our analyses (the static analysis part in particular) soundy instead of *sound* as they do not have a fully sound handling of all dynamic features in Java that are well recognized/accepted not to be fully handled (e.g., reflection API and native method calls, dynamic loading, customized class loaders, and exceptions and related flows, etc.) in order to obtain reasonably acceptable analysis precision and scalability [61]; however, our analyses are sound as dynamic analyses [57] otherwise—we even handled exceptional control flows as described before for which our static analyses may dismiss the full handling to remain soundy [62]. Therefore, in the following studies, we treat as *safe* every impact set computed by either technique, and thus use impact set size ratios to measure relative precision between these two techniques. To empirically verify this assumption, we conducted two validation studies, including an automated validation using dynamic slicing and a manual validation based on understandings of the source code and runtime program behaviours (Section 7.3.3).

TABLE 1: Subjects for the Arbitrary-Query Studies

Subject	#LOC	#Classes	#Methods	#Tests
Schedule1	290	1	24	2,650
NanoXML-v1	3,521	92	282	214
Ant-v0	18,830	214	1,863	112
XML-security-v1	22,361	222	1,928	92
BCEL 5.3	34,839	455	3,834	75
JMeter-v2	35,547	352	3,054	79
JABA-v0	37,919	419	3,332	70
OpenNLP 1.5	43,405	734	4,141	344
PDFBox 1.1	59,576	596	5,401	29
ArgoUML-r3121	102,400	1,202	8,856	211

7 STUDY I: SINGLE-METHOD QUERIES

In this study, we evaluate the effectiveness and efficiency of DIVER against the baseline technique for arbitrary locations or changes but limited to single methods. To that end, for each subject, we compute and compare the impact sets of the two techniques exhaustively for every method in the subject as a separate query.

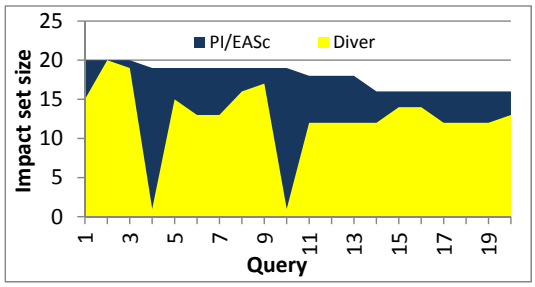
7.1 Subjects

We chose ten Java programs of various types and sizes, as summarized in Table 1, for our study. The size of each subject is measured as the number of non-comment non-blank lines of code (LOC) in Java. The first column gives the name of each subject with the version (or revision number) we used. The other columns list the number of total classes (*#Classes*), methods (*#Methods*), and test cases (*#Tests*) that come with the respective subject.

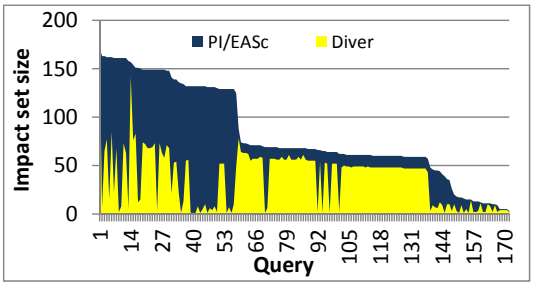
Schedule1 is part of the Siemens suite [63] which represents small modules. NanoXML is a lean and efficient XML parser. XML-security is an Apache library for signatures and encryption. JMeter is an Apache application for performance testing. Ant is a cross-platform build tool. We took these subjects and their test suites from the SIR repository [64] and picked the first available version of each. JABA [65] is a Java-bytecode analyzer obtained from its authors, coming with a full set of regression tests. For each of the other four subjects, we checked out a stable release, including the test suite, from its SVN repository. BCEL [66] is the Apache library that manipulates Java class files. PDFBox [67] is a PDF document processing tool, also from the Apache project as is OpenNLP [68], a machine-learning based toolkit for natural language processing. Finally, ArgoUML [69] is an open-source UML modeling tool.

7.2 Methodology

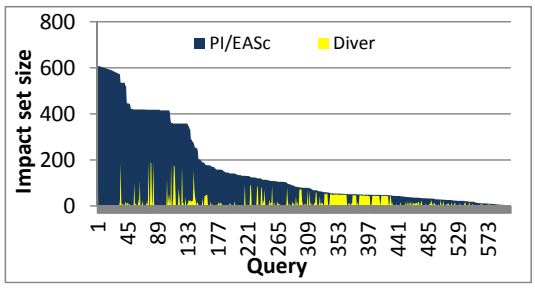
For our experiments, we applied PI/EAS$_C$ and DIVER separately to each subject. To obtain the method traces, we used the entire test suites provided with the subjects. Using each of the two techniques, we separately computed the impact sets for all methods in each subject. As expected, some methods are never executed and thus have empty dynamic impact sets. We excluded data points



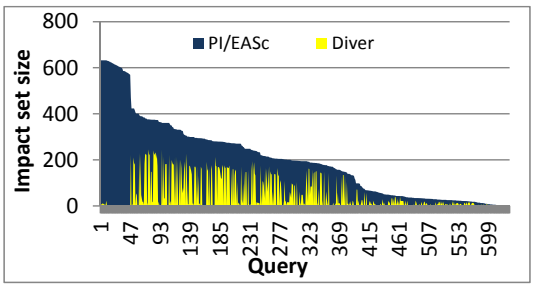
(a) Schedule1



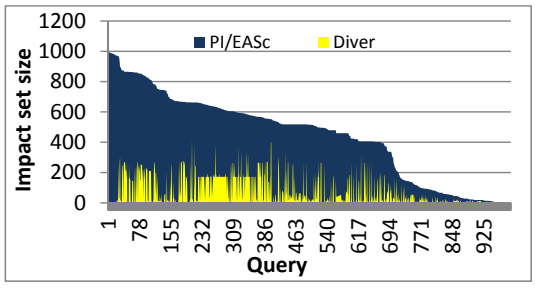
(b) NanoXML



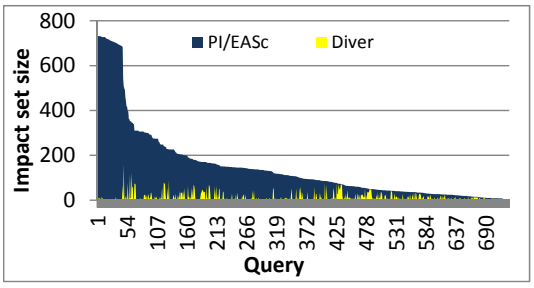
(c) Ant



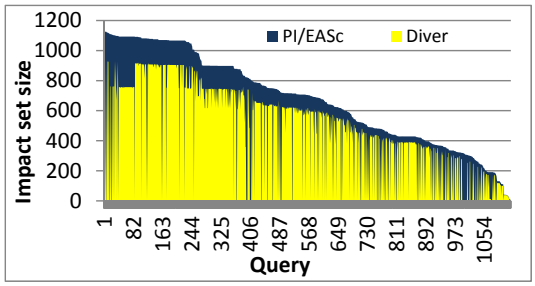
(d) XML-security



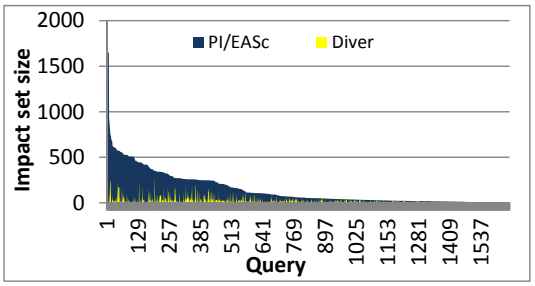
(e) BCEL



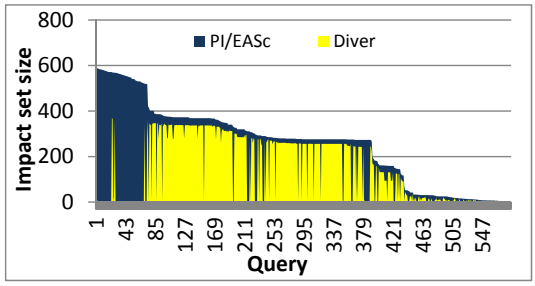
(f) JMeter



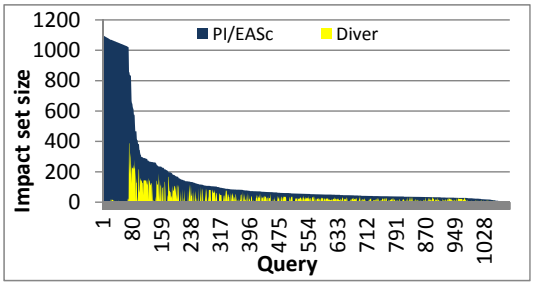
(g) JABA



(h) OpenNLP



(i) PDFBox



(j) ArgoUML

Fig. 4: Impact set sizes (y axis) of DIVER versus PI/EAS$_C$ for each query (x axis) and subject (caption under each single plot) from the study on single-method queries. The queries are sorted in non-ascending order of impact set sizes of PI/EAS$_C$ to facilitate comparisons between the two techniques. For every single query, the DIVER impact set was always a subset of the corresponding PI/EAS$_C$ impact set. Given the soundiness [61] of both techniques, smaller impact-set size implies higher precision.

TABLE 2: Relative Precision of DIVER Measured by the Size Ratios of Its Impact Sets over PI/EAS$_C$ per Subject and Overall

Subject	#Queries (methods)	PI/EAS$_C$ Impact Set Size			DIVER Impact Set Size			Impact Set Size Ratio			Wilcoxon
		mean	stdev	median	mean	stdev	median	mean	stdev	median	p-value
Schedule1	20	18.0	1.6	18.5	12.8	4.7	13.0	71.3%	24.5%	75.0%	6.65E-05
NanoXML	172	82.6	48.1	68.0	37.1	28.9	48.0	51.7%	33.1%	60.8%	2.40E-30
Ant	607	159.5	173.4	79.0	17.9	34.3	4.0	25.7%	33.6%	6.9%	2.94E-100
XML-security	632	199.8	168.4	194.0	45.1	68.1	7.0	28.8%	30.3%	12.4%	4.79E-102
BCEL	993	446.0	280.1	515.0	69.2	92.5	15.0	17.4%	18.2%	10.5%	2.29E-164
JMeter	732	149.6	172.6	96.0	12.3	18.5	4.0	18.8%	25.1%	6.6%	8.58E-122
JABA	1,129	677.0	301.2	705.0	471.9	308.2	550.0	66.9%	33.7%	84.0%	1.51E-186
OpenNLP	1,657	146.9	202.4	53.0	17.6	31.2	7.0	26.8%	28.8%	14.7%	1.23E-272
PDFBox	588	258.2	172.6	276.0	146.8	141.6	131.0	57.8%	38.3%	80.8%	2.38E-98
ArgoUML	1,098	151.0	261.2	54.0	27.6	44.9	13.0	31.5%	26.0%	33.3%	1.66E-181
Overall average		278.1	299.8	151.0	105.4	205.9	11.0	**34.54**%	**33.87**%	**23.08**%	**0**

for those queries from our results as they are not applicable for data analysis of the study. For each query, we report the union of per-test impact sets and cumulative querying cost for all individual test cases in the entire test suite when calculating the effectiveness and efficiency of a technique on that query, respectively.

To compare the analysis precision between the two techniques, we calculated for each query the impact-set size for DIVER and PI/EAS$_C$ and the *size ratio* of the first to the second. We computed the means, medians, and standard deviations of those metrics for all queries per subject and all subjects. As we discussed earlier, both techniques over-approximate the concept of *semantic* (true) dependence [47], their results are *dynamically safe* (i.e., safe with respect to the test suites utilized). Thus, the ratios reflect *relative precision* improvements that DIVER attains over PI/EAS$_C$ for executions from those test suites, although they may not indicate the absolute precision relative to ground-truth impacts which are unknown until actual changes are available. And in all of our studies, for either technique, smaller impact sets indicate higher effectiveness; for DIVER, the higher the relative precision it has, the more effective it is.

To measure and compare the efficiency of the two techniques, we computed for each subject the time and space costs of their respective static-analysis and runtime phases. This was needed only once per subject and technique because all queries at post-processing reuse the results of the first two phases. For the post-processing phase, we collected the time costs *per query* and report the means, medians, and standard deviations of these costs for all queries per subject and overall for all subjects.

7.3 Results and Analysis

This section presents the results of the study on single-method queries. We report and discuss the relative precisions of DIVER versus PI/EAS$_C$ and the costs that both techniques incur.

7.3.1 RQ1: Effectiveness

Figure 4 presents the details of effectiveness results for DIVER and PI/EAS$_C$ from the study on single-method queries. For each subject, the area chart depicts the impact set sizes given by the two techniques for all queries for that subject. In each chart, the x axis represents the queries and the y axis the resulting impact-set sizes. To facilitate visual comparisons, results are sorted by non-increasing PI/EAS$_C$ impact-set size. For example, for the first query in Schedule1, PI/EAS$_C$ reported 20 possibly-impacted methods for Schedule1's test suite whereas DIVER reported (safely) that only 15 of them could have been impacted.

The charts confirm that, for all queries, DIVER never produced a larger impact set than PI/EAS$_C$. We also verified that, for every query, the DIVER impact set was constantly a subset of the corresponding PI/EAS$_C$ impact set. Moreover, for a majority of queries, DIVER reported much smaller impact sets and, therefore, achieved a much greater precision overall. The contrast between DIVER and PI/EAS$_C$ is considerable and it is particularly sharp for six of the eight largest subjects for which, on average, DIVER computed impact sets as less than 30%, and as little as 17%, large as the ones reported by PI/EAS$_C$.

Table 2 provides three statistics per subject and overall (the last row) for the corresponding data points of Figure 4: the mean, standard deviation (*stdev*), and median of the impact-set sizes. The *#Queries* column lists the number of single-method queries per subject, which is equal to the respective method-level *test coverage*. To the right, the table presents the same statistics but for the impact-set size ratio of DIVER to PI/EAS$_C$. As in other tables of this paper, the *overall average* is the statistic over all data points of all subjects rather than over the per-subject statistics.

Results in the table reaffirm the major findings from Figure 4. Large numbers of false positives from PI/EAS$_C$ were identified as such and discarded by DIVER. For example, PI/EAS$_C$ identified 151 methods on average in its impact sets for ArgoUML, whereas DIVER reported only 28 with a mean ratio of 31.5%. (These values are means of per-query ratios, not ratios of per-query impact-set size means, as in other similar tables.) Also, the large standard deviations indicate that the impact-set sizes fluctuate greatly across queries for every subject except Schedule1. The comparatively-small median sizes in many cases imply that large deviations are caused by a few queries with large impact sets.

For Ant, XML-security, BCEL, JMeter, OpenNLP and ArgoUML, the medians are especially small compared to the means, which indicates that DIVER has an especially-greater precision than PI/EAS$_C$ for many queries. The results also suggest that DIVER is mostly even stronger with respect to PI/EAS$_C$ for larger subjects, which are more representative of modern software. For JABA and PDFBox, however, DIVER has larger impact sets and ratios. We examined these two subjects and found that they have generally much tighter inter-module couplings than other subjects. Thus, PI/EAS$_C$, which reports impacts simply based on execution order, tends to guess correctly more impacts than for other subjects. Nevertheless, DIVER can be quite beneficial even in these worst cases as its reductions peak at queries that have the largest impact sets, where developers would spend the most excessive efforts with PI/EAS$_C$. Finally, for the smaller subjects Schedule1 and NanoXML, DIVER is less effective than average possibly due to the proximity and interdependence of the very few methods in these programs.

We applied two statistical analyses to these results. The first one is the Wilcoxon signed-rank one-tailed test [70] for all queries

TABLE 3: Time Costs in Seconds of DIVER and PI/EAS$_C$, Including the Overheads of Profiling Uncaught Exceptions for DIVER

Subject	Prof.	Static analysis phase		Runtime phase			Post-processing phase					
							PI/EAS$_C$			DIVER		
		PI/EAS$_C$	DIVER	normal	PI/EAS$_C$	DIVER	mean	stdev	median	mean	stdev	median
Schedule1	12.7	4.8	5.6	4.0	10.1	15.7	0.7	0.1	0.8	14.6	6.0	14.2
NanoXML	12.1	11.3	14.4	0.4	1.0	5.4	0.1	0.1	0.1	6.2	8.8	0.5
Ant	29.2	27.3	142.4	1.2	1.5	2.0	0.1	0.1	0.1	3.2	7.6	0.1
XML-security	37.1	33.4	157.7	4.3	4.8	14.8	0.1	0.1	0.1	7.4	9.6	0.4
BCEL	47.7	32.9	1,717.2	4.8	7.4	20.9	0.1	0.1	0.1	116.1	105.6	192.8
JMeter	50.5	38.3	371.9	12.1	12.6	14.8	0.1	0.1	0.1	2.3	7.8	0.2
JABA	62.1	55.3	289.2	11.0	11.9	14.0	0.3	0.2	0.3	78.3	82.5	59.2
OpenNLP	53.7	50.6	734.8	51.6	56.8	58.7	0.1	0.1	0.1	75.2	228.5	0.4
PDFBox	69.6	59.2	689.7	6.9	9.8	11.8	0.1	0.1	0.1	113.2	149.7	25.6
ArgoUML	190.1	172.3	7,465.2	8.2	9.7	11.2	0.1	0.1	0.1	15.9	58.2	0.1
Overall average	**70.3**	**61.7**	**1,614.1**	**16.8**	**19.0**	**23.0**	**0.1**	**0.2**	**0.1**	**55.4**	**133.8**	**1.6**

in each subject. This is a *non-parametric* test that makes no assumptions about the data distribution. The last column in Table 2 shows the resulting p-values. For $\alpha = .05$, the null hypothesis is that DIVER is not more precise than PI/EAS$_C$. The p-values show strongly that the null hypothesis is rejected and, thus, the superiority of DIVER is statistically significant for these subjects and test suites. In the bottom row, the combined p-value of 0 calculated using the Fisher method [71] shows that the significance is particularly strong over all queries of the ten subjects—the Fisher method provides a means to aggregate multiple individual p-values to provide an overall view of the statistic significance.

The second analysis is Cliff's delta [72] for the *effect size* of the precision differences, which is also non-parametric. We used a 95% confidence level (i.e., α=0.05) per subject with the impact-set sizes of DIVER and PI/EAS$_C$ as the experimental and control groups, respectively. For all subjects but Schedule1, the effect sizes are 0.98 to 1.0 with confidence intervals within [0.95, 1.0]. For Schedule1, the effect size is 0.95 within [0.78, 0.98]. These values mean that, if the results *were normally distributed* (just for the sake of the effect-size interpretation), about 84% of the DIVER impact sets would be smaller than for PI/EAS$_C$ [73]. For the other 16%, the impact sets would be the same (because PI/EAS$_C$ cannot outperform DIVER).

Limitations of test inputs. As Tables 1 and 2 together suggested, the coverage of test inputs used for some of the subject programs was relatively low. Consequently, the execution data utilized by our technique on those subjects (e.g., PDFBox) might be less representative of their behaviors than the data generated from the test inputs that achieved higher coverage for corresponding subjects (e.g., NanoXML). In general, the evaluation results on the effectiveness of our analyses are limited by the quality of test inputs utilized in our studies. To minimize the effect of such limitations, we purposely chose subjects for which we can access quality test inputs. It is noteworthy that these are limitations of the evaluation results presented with respect to the particular subjects rather than those of our technical approach itself. Also, according to the results obtained (Figure 4 and Table 2), the effectiveness of DIVER did not seem to be clearly correlated with the test coverage—DIVER achieved no greater/lower impact-set reduction on subjects with higher/lower-coverage test inputs. Nevertheless, we would not claim that the limited test coverage does not affect the validity and generality of our empirical results and associated conclusions, either in this study or the following ones. Instead, the effectiveness results reported should be understood with respect to the test inputs and corresponding execution data (i.e., method-execution events) we actually used for the impact analysis.

In all, for ten Java subjects of different types and sizes, DIVER can *safely* prune 65% of the impact sets computed by PI/EAS$_C$. This amounts to an increase in precision by a factor of 2.86 (i.e., by 186%) over the (almost) best (most precise and cost-effective) existing method-level technique prior to DIVER [35].

7.3.2 RQ2: Efficiency

Table 3 shows the time costs of each phase of each technique, including the time of uncaught exception profiling (column *Prof.*), static analysis, test-suite execution for the non-instrumented program (*normal*) and for both techniques, and post-processing. All costs except for post-processing were incurred only once per subject. For the last phase, we show per subject and overall the mean, standard deviation (*stdev*), and median of per-query costs.

The profiling numbers suggest that automatically finding the static-analysis settings is cheap—a minute or less for most subjects and three minutes for the largest program ArgoUML. As expected, for static analysis, DIVER incurred higher costs than PI/EAS$_C$. For both techniques, these costs increase with the size of the program, with DIVER growing faster, in all subjects but BCEL. Our manual examination reveals that this exception is mainly due to a few abnormally long static initializers each consisting of a large number of constant initializations in BCEL, slowing down dependence computations hence the dependence-graph construction. And the long sequence of constant initializations in a single method is mainly used for speeding up symbol lookups for bytecode parsing and manipulation, which could be regarded as a special case of application domain. However, other than BCEL and ArgoUML, the DIVER static analysis finished within 12 minutes, which is reasonable because this is done only once per program version to support impact computation for all possible queries during the post-processing phase. For an application of special domain such as BCEL and large (industry-scale) subject such as ArgoUML, a half- to two-hour static analysis seems acceptable too, which can be done during nightly builds.

For the runtime phase, both techniques had small overheads: in one minute at worst. For the post-processing phase, due to the traversal of longer traces, DIVER needed more time than PI/-EAS$_C$. However, the average cost of 55 seconds per query seems practical, albeit a little less for BCEL and PDFBox. For BCEL, the higher cost is mainly due to the same reason that causes the long static-analysis time relative to its size; while for PDFBox the reason is more likely to be what causes the relatively lower impact-set size reductions seen in Figure 4, as discussed earlier. Yet, less than two minutes per queried method still seems acceptable, and multiple queries can be processed in parallel. In fact, the small median implies that in most cases, a single query will cost only a

TABLE 4: Space (Storage) Costs of DIVER and PI/EAS$_C$

Subject	Runtime (trace) data (MB) PI/EAS$_C$	DIVER	Dependence graph (MB)
Schedule1	1.5	5.4	0.1
NanoXML	0.4	1.6	0.9
Ant	0.3	1.7	5.5
XML-security	0.4	1.8	5.4
BCEL	0.3	23.1	11.6
JMeter	0.2	0.7	9.0
JABA	0.9	9.8	20.4
OpenNLP	0.3	83.9	20.9
PDFBox	0.1	14.8	17.1
ArgoUML	0.7	3.9	40.8
Overall average	**0.4**	**24.7**	**18.1**

couple seconds. Moreover, note that the query cost for each query was the sum of per-test querying costs for all tests in the whole test suite. In some application scenarios, such as debugging and comprehension, developers may just need to use a subset of all tests hence expect even lower post-processing overhead. It is also noteworthy that the cost DIVER incurred did not monotonously grow with program sizes. Thus developers would not necessarily expect even longer analysis time when applying the technique to an even larger (than 100KLOC) program.

Table 4 shows the space costs of the two techniques for relevant phases. As expected, the DIVER traces use more space than the PI/EAS$_C$ integers. In addition, DIVER incurs the cost of storing the dependence graph during static analysis for use later in query processing. One expected correlation can be seen between space use—especially trace sizes—and post-processing times (Table 3), which is that longer traces mostly lead to greater post-processing costs. However, these space costs of DIVER at a few dozen MB or less are quite low for today's storage availability.

In all, DIVER achieved significantly greater effectiveness for these subjects at acceptable time and space costs. Only the static-analysis cost for BCEL and ArgoUML suggests that more optimizations in our implementation or the dependence-graph construction and impact-computation algorithms, or both, might be needed for larger subjects with longer executions and those of special application domains. In addition, our tool is currently not tuned or optimized but rather a research prototype; thus, it is reasonable to expect considerable reduction in the analysis time from a well-optimized implementation. Finally, we used only a commodity machine for our study; a better-configured hardware platform, with larger memory for example, would see even higher efficiency of DIVER.

Given the drastic reduction of false-positive impacts it achieves relative to the baseline, DIVER offers a much more cost-effective option: The time developers would spend on inspecting 65% more false impacts, particularly in the cases of very-large impact sets, should easily pay off, by far, the extra one-time cost of at worst a couple of hours and additional querying time of no more than a couple of minutes.

7.3.3 Validation of Recall

We assumed that DIVER is *soundy* and a sound dynamic analysis [57] (Section 6) as per our definition of true positive, safety, and soundness (Section 1). Further, we empirically validated the *safety* of impact sets produced by DIVER using two case studies. It is worth pointing out that only if we can show that each DIVER impact set is *safe*, can we then use the relative impact-set size ratios as a measure of precision and effectiveness for RQ1.

Methodology. In the first case study, for each of the ten subjects, we computed the method-level forward dynamic slice of each method in that subject program using our trace-based forward dynamic slicer [74]. The dynamic slicer works at statement level as usual. To obtain the method-level slice of a method m, we used each statement (along with the variable defined there, if any) in m as a slicing criterion to calculate the statement-level forward dynamic slice for the same test inputs used by DIVER. Then, we took the union of all such slices [75] for m, and lifted the slice to method level (by picking the enclosing method of each statement in a slice). As such, we computed the forward dynamic slice for each (single-method) query (i.e., each method covered by any of the given test cases) for which DIVER computed the impact set in Study I. We used the slice of each query as the ground-truth impact set to validate the recall of DIVER with respect to the query.

In the second study, we randomly picked twenty out of all the queries from each subject and manually identified the ground-truth impact sets for the same test inputs DIVER used in Study I. (For Schedule1, the entire subject was covered by the validation study since it happened to have 20 queries.) We limited the scale and scope of this study such that during the random query selection, queries for which the DIVER impact set had more than 50 methods were skipped. The reason was because the manual inspection was both extremely time-consuming and tedious. In identifying the manual ground-truth impact set for each query, we exploited our code comprehension of the subject program and followed the program execution paths with respect to the associated test suite in a debugging setting (within the Eclipse IDE).

Results. In both case studies, our results confirmed that DIVER impact sets were constantly safe (i.e., it had always 100% recall) relative to the above ground-truth impact sets for corresponding queries and test inputs, although for most of the selected queries we did find false positives in the DIVER impact sets. This consistency shows that DIVER has indeed remained soundy while pruning false positives from the PI/EAS$_C$ impact sets. As the results of Study II were directly derived (through unionization) from the impact sets of single-method queries and Study III used the same impact-computation algorithm as Study II, we skipped the recall validation for those studies.

Note that while we used dynamic slicing for recall-validation purposes, dynamic slicing itself is not a competing solution to dynamic impact prediction against our approach—as confirmed in our validation study, dynamic slicing would be too heavyweight and expensive to be practically adoptable. In our experiments, computing a single method-level dynamic slice costed over 30x time on average compared to computing an impact set for the same query with the slicer implementation sharing the same underlying analysis facilities (e.g., pointer analysis and dependence computation algorithms) as used by DIVER.

DIVER achieves a constant perfect recall of 100% according to our exhaustive, automatic validation and selective, manual validation of its resulting impact sets against the ground-truth impact sets for each possible query. Thus, the relative impact-set size ratios can be used as a precision and effectiveness measure as per our experimental methodology.

8 STUDY II: MULTIPLE-METHOD QUERIES

The goal of this second study is to continue to investigate the performance of our technique for arbitrary program locations or

TABLE 5: Average Impact-Set Size Ratios of DIVER over PI/EAS$_C$ for Multiple-Method Queries with Query Sizes 1 to 10

Subject	Query size (number of methods in each query)									
	1	2	3	4	5	6	7	8	9	10
Schedule1	0.71 (0.25)	0.78 (0.17)	0.82 (0.13)	0.85 (0.11)	0.87 (0.10)	0.90 (0.09)	0.89 (0.10)	0.92 (0.07)	0.92 (0.07)	0.91 (0.09)
NanoXML	0.52 (0.33)	0.52 (0.30)	0.53 (0.27)	0.53 (0.24)	0.54 (0.22)	0.54 (0.20)	0.54 (0.19)	0.55 (0.18)	0.56 (0.18)	0.56 (0.18)
Ant	0.26 (0.34)	0.25 (0.32)	0.24 (0.31)	0.24 (0.30)	0.23 (0.28)	0.23 (0.27)	0.22 (0.26)	0.22 (0.25)	0.21 (0.23)	0.21 (0.22)
XML-security	0.29 (0.30)	0.29 (0.29)	0.29 (0.29)	0.29 (0.28)	0.30 (0.27)	0.30 (0.26)	0.31 (0.25)	0.32 (0.25)	0.32 (0.24)	0.33 (0.23)
BCEL	0.17 (0.18)	0.18 (0.18)	0.18 (0.18)	0.19 (0.17)	0.19 (0.17)	0.20 (0.17)	0.20 (0.17)	0.21 (0.17)	0.22 (0.16)	0.22 (0.16)
JMeter	0.19 (0.25)	0.18 (0.24)	0.18 (0.23)	0.17 (0.23)	0.17 (0.22)	0.17 (0.21)	0.16 (0.20)	0.16 (0.19)	0.16 (0.18)	0.16 (0.18)
JABA	0.67 (0.34)	0.69 (0.32)	0.70 (0.29)	0.72 (0.27)	0.73 (0.25)	0.74 (0.23)	0.75 (0.21)	0.76 (0.19)	0.77 (0.18)	0.77 (0.16)
OpenNLP	0.27 (0.29)	0.25 (0.27)	0.24 (0.26)	0.23 (0.24)	0.22 (0.23)	0.21 (0.21)	0.21 (0.20)	0.20 (0.19)	0.20 (0.18)	0.19 (0.17)
PDFBox	0.58 (0.28)	0.58 (0.38)	0.59 (0.38)	0.59 (0.38)	0.60 (0.37)	0.60 (0.37)	0.61 (0.37)	0.61 (0.36)	0.62 (0.36)	0.62 (0.36)
ArgoUML	0.32 (0.26)	0.32 (0.26)	0.32 (0.26)	0.32 (0.25)	0.32 (0.25)	0.32 (0.25)	0.32 (0.25)	0.32 (0.25)	0.32 (0.24)	0.32 (0.24)
Overall average	**0.35** (0.28)	**0.35** (0.28)	**0.35** (0.27)	**0.35** (0.25)	**0.35** (0.24)	**0.35** (0.23)	**0.35** (0.23)	**0.35** (0.22)	**0.35** (0.21)	**0.35** (0.20)

changes. In the previous (single-method) study, for each single query, we limited those location or changes to single methods to exhaustively gauge the ability of DIVER to assess the effect of each individual method on the rest of the program. By contrast, this study focuses on queries each containing multiple methods. For one thing, multiple-method queries are common in use scenarios where developers plan a large code change consisting of a few smaller changes spread over multiple methods. Also, performing such queries is necessary for assessing the collective effects of multiple methods together on the entire program.

8.1 Experimental Setup

For a multiple-method query, we refer to as *query size* the number of methods constituting that query. In this study, we experimented with nine such sizes ranging from two to ten. We used the same ten subjects with their test inputs as those in the single-method study. To compute the impact set of a multiple-method query, DIVER simply first computes the single-method impact set for each constituent method in the query and then takes the union of all those impact sets as the resulting impact set. Since the querying process for each constituent method is independent of that for others, the multiple-method querying is parallelized simply via multi-threading (one thread per single-method querying). In contrast, PI/EAS$_C$ deals with a multiple-method query by first finding the earliest entrance event of all constituent methods and then taking as the impact set the set of methods whose returned-into events occurred after that entrance event [18].

For each subject and query size, we computed impact sets of multiple-method queries such that each of the entire set M of methods in the program was included exactly in one query. To that end, for every query size qs from two to ten, we randomly picked, and removed afterwards, qs unique methods from M to form a query, iteratively until no more methods left in M. As a result, the last query might not be of size qs, but we simply treat it as a qs-size query. To reduce biases in these random selections, we repeated the entire process $1,000$ times with varying randomization seeds. We then computed the means and standard deviations of impact set sizes from DIVER and PI/EAS$_C$, and the same statistics of impact-set size ratios of DIVER over PI/EAS$_C$ for all queries in the $1,000$ repetitions.

8.2 Results and Analysis

This section presents the results of Study II on multiple-method queries. We use the results to primarily answer RQ3 but also to further answer RQ1 and RQ2.

8.2.1 RQ1: Effectiveness

Table 5 summarizes the mean impact-set size ratios of DIVER over the baseline (i.e., the relative precision of DIVER), with standard deviations of means (*stdev*) shown in the parentheses, for all the impact-set data points collected following the experimental procedure described above. The bottom row shows the overall average over all the ten subjects per query size. We put the results of single-method queries, the same as shown in Table 2, in the second column (for qs=1) to facilitate comparisons. Each of the other columns to the right shows the results for one of the other nine query sizes we studied.

For almost all of these subjects, the effectiveness (precision) of DIVER relative to the baseline for multiple-method queries is very close to that for single-method ones. More specifically, with the increase in query size, most subjects saw a slight decrease in the precision (i.e., increase in impact-set size ratios), implying less overlapping among DIVER impact sets of constituent methods than that among PI/EAS$_C$ impact sets. Schedule1 and JABA, however, had relatively large magnitude in such decreases, with the impact-set size ratios growing by 20% and 10% as the query sizes increasing from one to ten, respectively. One possible reason is that the tighter coupling among methods in these two subjects (relative to other subjects) leads to shorter distances among constituent methods in the execute-after sequences, which results in even larger overlapping among constituent PI/EAS$_C$ impact sets hence even slower growth in the size of their union.

Interestingly, on the other hand, for a few other subjects (Ant, JMeter, and OpenNLP), DIVER achieved even greater impact-set reductions relative to PI/EAS$_C$ for larger queries. The most noticeable such inverse correlation was seen with OpenNLP, for which the impact-set sizes of ten-method queries are reduced by 8% on average in contrast to those of the single-method queries. This observation might be explained by possibly larger size and number of dependence clusters [76], at least at the method level [54], in the source code of these subjects than in others. The existence of these dependence clusters lead to slower growth in the size of the union of constituent DIVER impact sets in comparison to PI/EAS$_C$ impact sets for multiple-method queries.

Nevertheless, generally the effectiveness of DIVER appears to be quite stable, as shown by the overall average impact-set size ratios: When the query size grows from one to ten, the ratio over all data points of the ten subjects keeps on a still value of 35%, with steady variations as well. This finding implies that DIVER is in general as effective for multiple-method queries as for single-method ones. Also, given this overall closeness in effectiveness measures (i.e., impact-set sizes and size ratios) between the two classes of queries, we omit the statistical-testing and effect-size analysis results, which were in fact quite similar too between them.

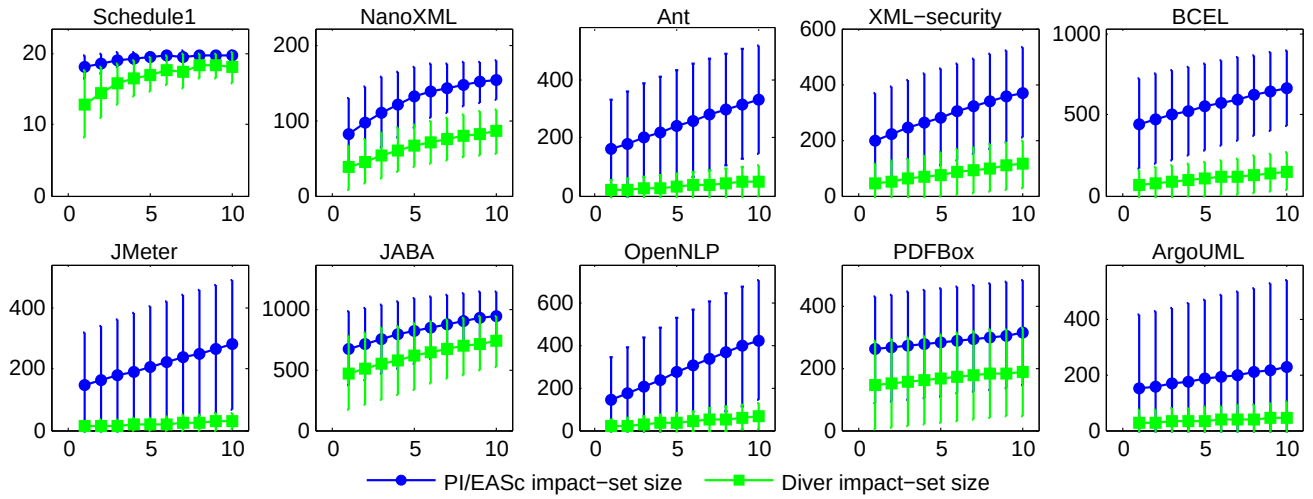


Fig. 5: Average impact-set sizes (y axis) of DIVER versus PI/EAS$_C$ for multiple-method queries with query sizes 1 to 10 (x axis)

8.2.2 RQ2: Efficiency

Our study results also show that performing a query of larger sizes did not incur appreciably higher costs than querying the impact sets of single methods. In fact, with our implementation where the impact sets for individual methods are computed in parallel, the querying costs of multiple-method queries are very close to, no more than 10% in the worst case than, those for single-method queries, both on average and in individual queries. Otherwise, since the variation in query sizes is irrelevant to the first two phases of DIVER, the overheads of those two phases did not change with varying query sizes. Therefore, we omit detailed efficiency metrics for this study here.

8.2.3 RQ3: Effects of Query Size

As can be seen from the summary results on impact-set size ratios in Table 5, overall the relative precision of DIVER did not seem to fluctuate much with the changes in query sizes. Nevertheless, to help further investigate the effects of query size on the effectiveness of DIVER, Figure 5 delineates the underlying data points for those size ratios: the impact set sizes, in ten individual plots. Each plot summarizes the contrast in impact-set sizes between DIVER and PI/EAS$_C$ for the query sizes from one to ten (as listed on the x axis) per subject (shown as the plot title), represented by curves with square and circle markers, respectively. For each query size, the markers on the curves indicate the means of impact-set sizes of all queries of that size, and the error bars extending toward above and below the markers indicate the associated standard deviations. Some of these error bars for PI/EAS$_C$ impact-set size means are not invisible because they are covered by overlapping error bars for corresponding DIVER means.

For any query and subject, the impact set of DIVER is noticeably smaller than that of the baseline, as we expected since the same contrast holds for each constituent method in the query as seen in the previous study. And in terms of composition, the DIVER impact set of any multiple-method query is constantly a subset of the PI/EAS$_C$ impact set of the same query, since the same inclusion relation holds for single-method queries as well. There are two additional major observations. First, with growing query sizes, the impact sets of both techniques grow monotonically, but the differences in slope between each pair of curves exhibit a slower growth of DIVER impact sets than that of the baseline ones, for all subjects except Schedule1. On the other hand, however, due to the increasing absolute sizes of the impact sets, the size ratios of DIVER to the baseline keep relatively steady in most cases with the same exceptions as discussed earlier on the size-ratio numbers of Table 5. A second observation is that DIVER saw a much smaller variations in impact-set sizes than PI/EAS$_C$, according to the standard deviations shown by the error bars, except for PDFBox in which a few queries had very-large impact sets. Our inspection reveals that those queries are mostly dispatching or coordinating methods at the core of this program.

In all, the query size did not appear to significantly affect the ultimate effectiveness of DIVER relative to PI/EAS$_C$, despite the relative precision of DIVER increased or decreased slightly with varying query sizes in some situations. This finding also dovetails what we found from the data presented in Table 5.

> In general, DIVER achieves the same level of precision for multiple-method queries with a variety of (ten) different query sizes as for single-method queries at a close level of cost also, with worst-case overhead increase of 10%. The query size does not appear to much affect the cost-effectiveness of DIVER.

9 STUDY III: REPOSITORY QUERIES

In the previous two studies, we investigated the effectiveness and efficiency of DIVER against the baseline technique with respect to arbitrary (all possible) queries, which gives a general estimation of DIVER performance. In practical use scenarios, however, developers may just need to inspect the impacts of specific sets of methods only. For instance, in the context of change-impact analysis, such practical queries are commonly those to which developers may plan to apply code changes. Thus, to complement the studies on arbitrary queries, in this study we further evaluate DIVER concerning how well it would perform on queries of particular interests—methods actually changed by developers between software revisions.

9.1 Experimental Setup

For this study, we intended to choose three open-source Java programs that are currently in active evolution and retrieve streams of consecutive revisions from their source repositories that reflect

TABLE 6: Statistics of Subjects Used for the Repository-Change Study

Repository (subject)	Revisions downloaded	#Valid revisions	#LOC	#Methods	#Methods changed	#Tests
XML-security 1.0.4	350313–350616	47	22,231–30,113	1,261–2,142	25.2 (95.6)	92
Ant 1.4.0	267546–269587	71	40,315–45,924	4,011–4,782	10.1 (13.9)	183
PDFBox 1.1.0	620304–928462	65	59,446–67,893	5,142–6,416	6.1 (16.1)	32
Total		183	121,992–143,930	10,414–13,340	11.7 (49.7)	307

series of code changes with different *semantic focuses* (each focus primarily addressed one topic, such as fixing a bug or adding a new functionality feature). For each subject, we started with the first release (of the chosen version, such as 1.0.4 for XML-security) as the original/starting revision and then continuously checked out following revisions until we found all the revisions dedicated for one semantic focus. To perform meaningful analyses, we only kept revisions that included at least one non-blank non-comment source-code change covered by the test inputs provided with the subject. During this procedure, we also dismissed revisions that only added new methods since the new methods are not known to the previous version hence it is impossible to predict their impacts. The remaining revisions were regarded as *valid* for our experiments. We repeated this procedure for one or two more semantic focuses (so that we consider changes of at least two different focuses) for each subject, and then analyzed all the valid revisions we gathered from the subject repository.

As a result, we got different numbers of valid revisions to use for different subjects, but each pair of consecutive revisions has at least one changed method that we use as the impact-set query against the earlier version that DIVER analyzes. Table 6 lists, in the first three columns, the repositories (subject programs) we selected, the ranges of all revisions we checked out from each repository, and the number of code-changing revisions that we actually analyzed per subject. The next three columns summarize the range of source sizes in terms of non-common non-blank lines of Java code (*#LOC*), range of total numbers of methods (*#Methods*), and the average number of methods changed between each pair of consecutive revisions (with the standard deviations shown in the parentheses). The last column indicates the number of test inputs available, and we actually used, for each subject.

Specifically, the 47 valid revisions of XML-security contained two semantic focuses, the first on upgrading the entire codebase to adapt to new versions of dependency libraries Xalan and Xerces, while the second fixing xmlns:xml namespace-handling issues. The 71 revisions of Ant focused on three series of semantic changes: adding several build tasks (Tar, Ear, and Filter), fixing usage examples of customized tasks, and enhancing the core build engine. Finally, the 65 PDFBox revisions concentrated on two topics, fixing bugs in the pdfbox.filter module and developing patches for the Fonts package.

Note that these subjects are a subset of those used in our previous two studies, although there are considerable gaps in the versions and sizes for each corresponding subject: each in this study has noticeably larger size than the same subject in the previous two, especially Ant. This selection is driven by two major considerations. First, we attempted to reuse experimentation utilities (e.g., setup scripts) from previous studies. Second and more importantly, we intended to get a sense of the practical lower-bound effectiveness of DIVER, within the ten subjects at least, so as to inform about the *estimated* worst-case benefits in saving impact-inspection efforts that developers may expect from our technique in real-world situations—albeit we do not claim that this lower bound is valid with DIVER for developers in any application scenarios in general. This second consideration led to the chosen three because, except for subjects that have no accessible or active version-control history of considerable length (e.g., JABA), these three received the worst-case effectiveness for the largest query size in Study II (recall that in most cases the relative precision of DIVER seemed to continuously worsen with growing query sizes).

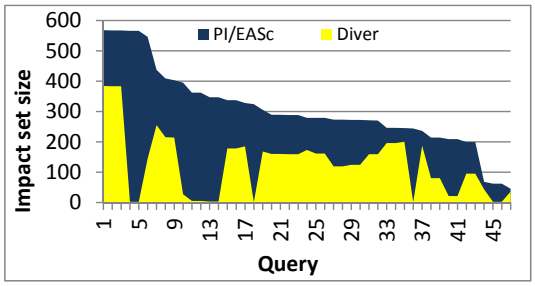
For the dynamic analysis, we utilized the set of test inputs coming as part of the starting revision of each subject. In addition, we consistently use that input set for all the following valid revisions of that subject to avoid possible relevant biases in computing aggregate statistics over all revisions: in fact, we did not find code changes in test inputs among all the valid revisions we utilized for any of these subjects.

To facilitate performing experiments with large numbers of program versions in this study, we developed an experimentation pipeline that automatically checks out a specified number of valid revisions from a given SVN repository, starting from a particular revision. In addition, the pipeline also configures, with some manual intervention, and compiles all valid revisions, as well as running DIVER and PI/EAS$_C$ on each of those revisions. One essential part of this tool is to find the set of methods changed between two SVN revisions. For that purpose, we developed a tool based on the srcML library [77], which represents the abstract syntax tree (AST) of given source code in XML. By comparing the ASTs of two revisions, the tool reports methods added, deleted, or modified from the earlier to the later version of the code.

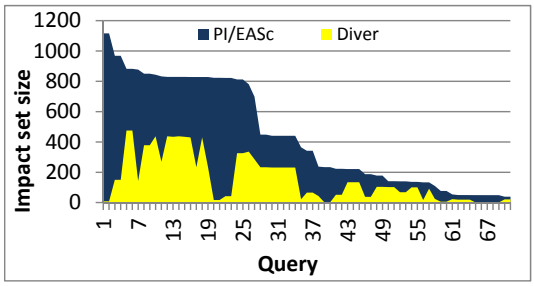
With these repository changes, we continue to evaluate DIVER against PI/EAS$_C$ in terms of effectiveness and efficiency using the same metrics as in the other two studies. For each subject, among all its valid revisions, we treat the methods changed between each pair of consecutive versions as one query, whether it be a single- or multiple-method query, and compute the impact set of the query with DIVER and PI/EAS$_C$ separately. Note that each of the valid revisions we studied corresponds to a developer commit to the code repository of the associated subject. Thus, each repository-based query is essentially the group of methods that the developer changed in that commit (on top of the immediately previous valid revision/commit). Again, for each pair, both techniques are applied only to the earlier version since the later version is not known to a *predictive* impact analysis.

9.2 Results and Analysis

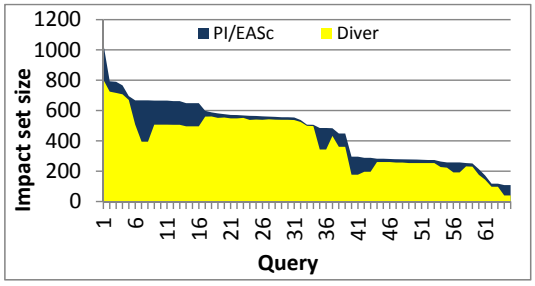
In this section, we present and discuss the results of the third study, focusing on assessing the performance of DIVER relative to the baseline technique in practical application contexts: for real queries that developers would actually run with an impact-analysis tool when evolving their programs. This study aims primarily at RQ4 in addition to further answering the other two common research questions (RQ1 and RQ2).



(a) XML-security



(b) Ant



(c) PDFBox

Fig. 6: Impact set sizes (y axis) of DIVER versus PI/EAS$_C$ for each query (x axis) and subject (plot caption) from the study of repository-based queries, plotted in the same format as Figure 4.

9.2.1 RQ1: Effectiveness

The full effectiveness results on repository-based queries are plotted by area charts in Figure 6, each for an individual subject in the same format as the ones in Figure 4. For each of the two techniques compared, the numbers of data points are those of the repository-based queries listed on the horizontal axes, which are also shown in the second column of Table 7.

The contrasts in impact-set sizes between the two techniques demonstrate again the effectiveness advantage of DIVER over the baseline approach: for two of these three subjects, DIVER impact sets are less than half smaller compared to corresponding PI/-EAS$_C$ ones for almost all cases (queries). For PDFBox, however, the improvement DIVER gained for most of the studied queries is not as significant as in other subjects, although the impact-set reductions are constantly noticeable. One plausible reason might still be the relatively much tighter method-level couplings which has made it among the worst-case subjects seen in our previous studies. Nevertheless, the observation that DIVER reduces the highest percentages (about 30%) of false impacts of PI/EAS$_C$ for the top (about 25%) queries that have largest impact sets implies that our technique remains much more effective where it is most needed: for queries that PI/EAS$_C$ will report most false impacts. This finding is also akin to what can be seen from Figure 4.

The area charts are plotted as such that the queries were listed in an non-ascending order of corresponding baseline impact-set sizes from the left to the right on the x axes. Meanwhile, the number of methods that executed after a query was not necessarily correlated with the number of methods that are dependent on the query at runtime. Thus, the DIVER impact-set sizes appear to fluctuate (and visually the area boundaries are jagged) along the x axes. The variations in the magnitude of impact-set reduction across different queries within each of the three subjects are similar to those observed in previous studies: the visual differences between Figure 6 and 4 with respect to these subjects are largely due to the fact that the total numbers of the repository-based queries are much smaller than those of arbitrary queries, hence the much lower density of actual data points in Figure 6.

In the same format as Table 2, the summary results of comparative effectiveness of DIVER are shown in Table 7. The absolute numbers on impact set sizes and ratios generally consolidate our observations from Figure 6, but even highlight the merits of DIVER in an average case: It is able to prune 58–68% of false impacts that would be produced by PI/EAS$_C$, corresponding to a precision improvement by 138–213% over the baseline; even for the worst-case subject, PDFBox, the impact-set reduction is 24% on average, meaning an improvement by 32% in precision.

To gauge the magnitude of differences in impact-set sizes between DIVER and PI/EAS$_C$, we conducted two statistical analyses, the Wilcoxon hypothesis testing and Cliff's Delta-based effect size, in same procedures and settings as we did for Study I. The p-values from the Wilcoxon tests for each subject and the combined Fisher p value for all subjects are shown in the last column of Table 7. The numbers confirm the strong statistical significance of such differences both individually and collectively. Results from the Cliff's Delta analysis show that the effect size for the three subjects is constantly 1.0, within a 0.95 confidence interval of [0.7, 1.0], implying that at least over 76% of all queries will receive an impact set from DIVER that is smaller than the average impact-set size of the baseline [73].

9.2.2 RQ2: Efficiency

The static-analysis costs of these three subjects are all close to the respective ones of similar sizes in Study I. Over all the studied revisions, this cost ranges from 13 to 16 minutes in the slowest case of Ant, slightly higher than 10 to 13 minutes for PDFBox. XML-security only needs two to three minutes in this phase. Runtime overheads are generally very low, with the highest of 27 seconds for Ant again, which has the largest number of test inputs among these three. DIVER incurred 15s and 6s for PDFBox and XML-security, respectively. In terms of storage costs, PDFBox saw the largest but no more than 20MB: the maximal space costs for the other two were both below 7MB.

The execution time of impact-computation in the post-processing phase varied with the query sizes in any of the three subjects (but not always proportionally). However, since multiple queries were processed in parallel, the querying costs were generally close to the single-method cases of subjects of similar sizes in Study I: the highest was seen by PDFBox, two minutes per query on average; for Ant and XML-security, the average costs were about one minute and 11 seconds,respectively.

In all, for programs at the scale of these three, the extra time and storage costs DIVER incurred seem reasonably acceptable, especially with respect to the effectiveness gains it brought. Even

TABLE 7: Effectiveness Results for Repository-Based Queries

Subject	#Queries	PI/EAS$_C$ Impact Set Size			DIVER Impact Set Size			Impact Set Size Ratio			Wilcoxon
		mean	stdev	median	mean	stdev	median	mean	stdev	median	p-value
XML-security	47	303.3	113.5	283.5	127.3	95.6	159.5	42.25%	27.07%	52.70%	9.69E-06
Ant	71	459.6	368.5	237.0	111.2	124.1	66.0	31.53%	24.22%	22.87%	1.61E-04
PDFBox	65	426.8	262.8	555.0	372.8	240.7	502.5	76.49%	26.99%	88.04%	8.25E-11
Overall average		407.8	253.5	288.0	208.3	146.9	124.0	**50.25%**	**31.31%**	**55.36%**	**1.27E-16**

with PDFBox, for which DIVER appears to have the lowest cost-effectiveness than for the other subjects, the inspection effort saved by DIVER would still well outweigh those extra overheads. Overall, the efficiency measures from this study are quite similar to those from the previous two.

9.2.3 RQ4: Comparison to Arbitrary-Queries Studies

Compared to the effectiveness it achieved for arbitrary queries, DIVER appeared to be appreciably less effective when working on repository-change-based queries. For example, the overall average impact-set reduction against the baseline was around 65% for arbitrary queries, for varying query sizes within the range of one to ten, while the reduction was 50% for repository-based ones. There are at least three possible reasons for this gap.

The first one to note is the different sets of subjects used between this study and the previous two. The three repositories chosen for this study are connected to the three of same names used in those two. Comparatively, however, the subjects in this study are all considerably larger than the corresponding ones there. For Ant, in particular, the one from its SVN repository is more than twice as large as the one from SIR, thus they are effectively two quite different subjects (the program changed dramatically).

The second one concerns the different construction of the experimental results between the two groups of studies. In the arbitrary-query group, for each of the subjects of large variety of sizes, we consider all methods defined in the program as possible queries and compute the overall metrics for all data points from the relatively large set of ten subjects; while in the repository-query group, which has much smaller number of subjects of much lesser variety in size, we only consider a very small subset of methods, namely those changed between consecutive code-changing SVN commits, for computing the overall metrics. Note that the overall metrics are essentially the weighted average of per-subject averages where the weights are the number of queries of each subject. In the first group, the subjects that account most for the overall statistics, namely those having the largest numbers of queries hence carrying the most weights, have the highest reductions, around 70% or even over 80%, except for only one (JABA) (Table 2); while in the second group, the subject having the lowest reduction rate (PDFBox) carries nearly the largest portion (of the three) of the total weight in the overall measure.

The last yet the most important reason is that, by our experimental design, we purposely chose the subset of programs used in the first group for which DIVER had the worst-case performance for the purpose of gauging the effectiveness lower bound of our technique in the second group. Although the actually used set of three subjects differs, mainly in size, from that subset, the 50% average reduction DIVER achieved over the baseline can still reasonably approximate the lower-bound relative precision of DIVER with respect to PI/EAS$_C$.

In all, the results from Study III show that DIVER is able to prune 50% false impacts produced by PI/EAS$_C$, or correspondingly 100% improvement in precision, on average for more than 180 queries retrieved from three active open-source repositories. While at the first glance the improvement for these repository-based queries is apparently less significant than the one for arbitrary queries, the former potentially represents the worst-case performance of DIVER, within the chosen set of ten subjects at least. Although we cannot claim that the relative precision improvement (of 100%) is the lower-bound effectiveness of our technique in general, this study suggests that developers using our technique may expect at least such an amount of benefit in most cases: saving their inspection effort that would be devoted to half of the methods reported by PI/EAS$_C$ per query.

> DIVER reduced the baseline impact sets for repository-based queries by 50% on average, with the same level of analysis cost as seen with single-method queries. As an estimated effectiveness lower bound, a precision gain by at least 100% relative to the baseline can be expected by developers using DIVER.

10 STUDY IV: ONLINE VERSUS OFFLINE

As we discussed earlier (Section 4.4), online and offline approaches to dynamic impact prediction have respective pros and cons and can be complementary to each other in terms of application scenarios and efficiency. This study aims to quantify such pros and cons by measuring the costs of impact analysis with these two approaches. With subjects and their test suites used in there that generally produce short traces, the first three studies seemingly showed that the space costs of DIVER were negligible—thus while DIVERONLINE further saved the space cost due to the traces generated at runtime, the savings seemed to be practically insignificant. To demonstrate the needs for and merits of online analysis we hypothesized about before, we use two additional, large-trace subject programs and recognize the challenges related to space cost due to trace storage. The foregoing studies confirmed that the query size did not much affect the time costs of our dynamic impact prediction. Also, since multiple-method queries are computed using the same sequence of method events (in memory or traces) as for computing single-method ones, query sizes do not affect the space cost either. Finally, the online and offline approaches share the same static analysis phase. Therefore, in this study we focus on the costs of other phases considering only single-method queries.

10.1 Experimental Setup

In this study, we used the same subjects and same test inputs for each subject as we did in Study I. For comparative evaluation of the online versus offline approach, we reused the efficiency results from that study too, including the querying time of each single-method query and the execution time of instrumented programs. To verify the functional correctness of DIVERONLINE, we also computed the impact sets for individual queries and reused corresponding results from Study I for comparison. We considered both

TABLE 8: Time and Space Costs of Computing Impact Sets for All Queries per Subject with DIVERONLINE versus DIVER

Subject (#queries)	Post-static-analysis time cost: seconds					Trace storage cost: MB (#trace files)	
	DIVERONLINE		DIVER	DIVERONLINE:DIVER Ratio			
	One-by-One	*All-in-One*		*One-by-One*	*All-in-One*	DIVERONLINE	DIVER
Schedule1 (20)	744.8	293.3	307.7	242.1%	95.3%	0 (0)	5.4 (2,650)
NanoXML-v1 (172)	1,132.7	1,012.2	1,071.8	105.7%	94.4%	0 (0)	1.6 (214)
Ant-v0 (607)	1,608.1	620.6	1,944.4	82.7%	31.9%	0 (0)	1.7 (112)
XML-security-v1 (632)	5,633.4	3,220.3	4,691.6	120.1%	68.6%	0 (0)	1.8 (92)
BCEL 5.3 (993)	13,079.6	4,796.8	115,308.2	11.3%	4.2%	0 (0)	23.1 (75)
JMeter-v2 (732)	12,177.7	1,207.6	1,698.4	717.0%	71.1%	0 (0)	0.7 (79)
JABA-v0 (1129)	109,581.6	81,032.3	88,414.7	123.9%	91.7%	0 (0)	9.8 (70)
OpenNLP 1.5 (1657)	123,395.5	65,639.3	124,665.1	99.0%	52.7%	0 (0)	83.9 (344)
PDFBox 1.1 (588)	166,344.0	54,790.8	66,573.4	249.9%	82.3%	0 (0)	14.8 (29)
ArgoUML-r3121 (1098)	99,862.4	1,500.4	17,469.4	571.6%	8.6%	0 (0)	3.9 (211)
Average (small trace)	**73,714.1**	**31,771.5**	**63,554.9**	**116.0%**	**50.1%**	**0 (0)**	**24.7 (89)**
JABA-v1 (1812)	322,823.9	160,189.4	196,917.8	163.9%	81.3%	0 (0)	42,227.1 (13,962)
OpenNLP 1.6 (2029)	3,235,134.9	1,183,737.6	2,705,960.2	119.6%	43.7%	0 (0)	15,381.1 (5,572)
Average (all)	**672,363.5**	**255,856.7**	**552,097.4**	**121.8%**	**46.3%**	**0 (0)**	**9,409.1 (3,301)**

variants of DIVERONLINE, *All-in-One* and *One-by-One*, when evaluating the online approach.

In addition, we used two more subject programs, which are upgraded versions of corresponding subjects in Study I: JABA-v1 and OpenNLP 1.6; and beyond functionality changes, these two subjects come with augmented test suites that include 114 and 418 test cases, respectively. The total number of methods in JABA-v1 remained the same as JABA-v0 (3,332), and that in OpenNLP 1.6 increased to 4,594 from the 1.5 version. Of these methods, 1,812 in JABA-v1 and 2,029 in OpenNLP were covered by the new test suites and were all used as impact-prediction queries.

To compare the efficiency of DIVERONLINE against DIVER, for each subject, we first measure the total time cost incurred by DIVERONLINE *All-in-One* for computing the impact sets of all queries for the subject; with DIVERONLINE *One-by-One*, we run the impact analysis per query and accumulate the total time cost for all queries. Similarly, we obtained the total time cost of DIVER as well by summing up the single-query time of the same, full set of queries, yet with the run time added to the total cost too for a fair comparison. In essence, this first metric includes the time cost for obtaining all impact sets (of methods covered by the corresponding test inputs) beyond the static analysis time which is the same among all these techniques compared, giving a holistic view of the comparative efficiency of each technique. The second metric is the mean time cost per query incurred by these techniques, which normalizes the total cost by the number of queries in each subject, gauging how fast each of these techniques would respond to a query from an average-case point of view. For the two additional subjects, we continue to measure their storage costs. Finally, we compare the runtime slowdown among these techniques to investigate how practical it would be to apply the impact analysis when running the programs for ordinary use, mainly from a user-experience perspective. This last metric is computed as, for a one-time execution of the subject on the given test inputs, the ratio of the time incurred by each of these techniques to the normal (original program) run time.

10.2 Results and Analysis

Our per-query impact set comparison between DIVER and (the two variants of) DIVERONLINE confirmed that the online algorithm did not change the effectiveness of our approach with respect to the offline algorithm: the impact set produced by either mode of DIVERONLINE was the same as DIVER for any of the full set of queries. This consistency was expected because both algorithms share the same rationale for dynamic impact prediction using the same amount and types of program information (i.e., static dependencies and method execution events). Therefore, we skip the effectiveness and recall evaluation in this study. Next, we compare the efficiency (in terms of time and space costs) of online versus offline analysis concerning the four metrics described above.

10.2.1 RQ5: Efficiency of Online versus Offline Analysis

Table 8 lists the total time cost (in seconds) incurred by the two variants of DIVERONLINE versus the offline approach DIVER for computing the impact sets of all possible queries per subject. For each subject, the number of queries involved in the computation is shown in the parenthesis. Note that the DIVER numbers have included the runtime costs as shown in Table 3. We also highlighted two overall averages (weighted by the numbers of queries): the average over the ten subjects used in foregoing studies which all produced relatively small traces (*Average (small trace)*), and the average over the full set of 12 subjects including the two additional ones that produced large traces (*Average (all)*). To facilitate later discussion, we hereafter refer to those ten as *small-trace* subjects.

The results reveal that the online analysis (in the default *All-in-One* mode) was always more efficient than the offline approach. By computing all the impact sets at once during runtime, DIVERONLINE saved substantially the analysis time cost beyond the first phase (where again the online and offline analysis share the same cost), by 30% or above for the majority of the 12 subjects. Many individual subjects saw even much greater savings: for example, with BCEL and ArgoUML the cost of the online analysis was only less than 9% of the offline analysis cost; the cost reduction by moving the impact analysis to runtime was dramatic with Ant and OpenNLP as well. The reason that these four subjects received the greatest time-cost savings was connected to the observation that they were also among the subjects that received the greatest impact-set reduction as seen in Table 2: their underlying dependence graphs have relatively sparse interprocedural dependencies. As a result, in these programs the impact originating from each query mostly only propagated shortly, leading to comparatively small impact sets; meanwhile, the computation time for the impact propagation incurred by DIVERONLINE for all the queries simultaneously tended to be often overweighed by the accumulated time spent by DIVER on repeatedly traversing the entire execution trace and performing associated I/O operations (once per query). Despite the negligible (around 5%) differences seen with the two smallest subjects Schedule1 and NanoXML,

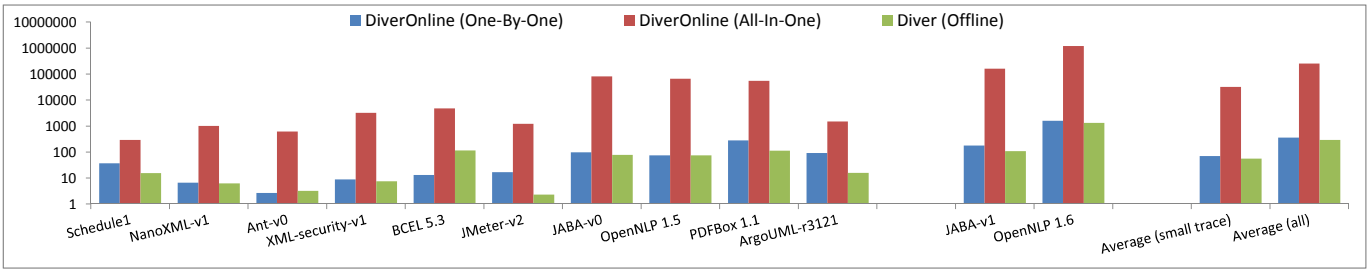


Fig. 7: Expected time cost (y axis in seconds) for obtaining the impact set of only one query with DIVERONLINE versus DIVER.

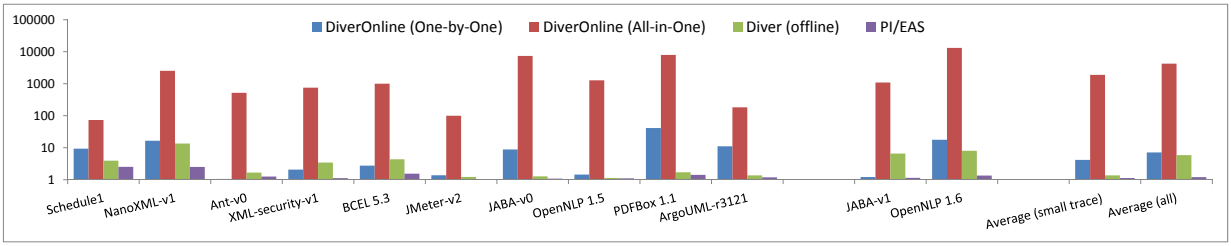


Fig. 8: Runtime slowdown in a single program execution with DIVERONLINE versus DIVER.

the *All-in-One* online analysis saved the time cost of the offline alternative by 50% on overall average, weighted by the per-subject numbers of queries (as shown in the last but the fourth row of the table), over the *small-trace* subjects. As we observed before, for these subjects the savings in the space cost due to trace storage were at most underwhelming. The overall savings for the 12 subjects were even greater (54%).

The *One-by-One* mode, however, rendered DIVER a favorable option over DIVERONLINE in terms of the time cost in question. For the two subjects, Ant and BCEL, with which the *All-in-One* analysis exhibited the greatest efficiency advantage over DIVER as mentioned above; now even in the *One-by-One* mode DIVERONLINE continued to reduce the offline-analysis time significantly (by almost 20% and 90%, respectively). Yet, most of the *small-trace* programs saw considerable penalty of repeatedly executing the program in the *One-by-One* analysis. For NanoXML and OpenNLP, the cost was at best comparable between the online and offline analysis. The worst case was found with JMeter and ArgoUML, for which the online approach was over seven and five times (respectively) slower. On overall average across the ten *small-trace* subjects, the *One-by-One* analysis users would need to wait about 16% longer than DIVER to obtain the impact sets of all possible queries for a given subject along with its test inputs.

Comparing between the two variants of DIVERONLINE reveals that *One-by-One* online analysis was constantly slower in contrast to the *All-in-One* impact prediction, and in most cases the difference was quite substantial. In the cases of PDFBox and ArgoUML, for instance, the disadvantage of answering one query at a time was highlighted: for ArgoUML, the largest subject out of the ten in our study, the online analysis was about 100 times slower in the *One-by-One* mode than it was when computing all queries at once. The main cause was that the dependence graph, also the largest among the ten subjects, needed to be loaded and the test suite (211 test cases) needed to be fully exercised *for each query*, while the inter-method loose coupling in this program (as mentioned earlier) led to the relatively small difference in the cost of computing all impact sets at the same time versus the cost of computing the impact set for only one query as discussed before.

For an average-case look at the time cost of answering a single query, Figure 7 depicts the contrast in the per-query time (shown on the y axis in logarithmic scale) among the three algorithms for each subject and overall (listed on the x axis). As shown, the normalization by query numbers did not much change the observation from Table 8 concerning the contrast between the *One-by-One* online analysis and the offline approach: DIVER was almost always noticeably (yet not hugely) more efficient than DIVERONLINE *One-by-One* for any of these subjects. The only exception was still Ant and BCEL with which the *One-by-One* analysis achieved the best efficiency as seen before, which implied that, on these two programs, computing the single query (in DIVERONLINE) was even cheaper than serializing the traces (in DIVER). Note that the per-query time cost of the *One-by-One* analysis is at least greater than the normal runtime cost (as seen in Table 3). Compared to its *One-by-One* mode, the *All-in-One* online approach has its disadvantage relative to offline analysis revealed conspicuously: DIVERONLINE *All-in-One* was overwhelmingly more expensive than DIVER for all the subjects studied. The major reason was that, even only a single query is requested, the *All-in-One* still computes the impact sets of all possible queries—the time cost remains the same as that for computing all impact sets as seen in Table 8. Thus, for users interested in a single query only, DIVER would be a better option in general.

The run-time slowdown imposed by a dynamic impact prediction technique can affect its adoptability: if the technique drags the ordinary program execution too much, it might not be acceptable to use the analysis on a regular basis in practice. In the cases of using user inputs or collecting field data [14], large slowdown could be a serious applicability obstacle too. Our results plotted in Figure 8 (with y axis in logarithmic scale) suggest that using DIVERONLINE in the *All-in-One* mode can be quite challenging in use scenarios where the impact analysis was performed during normal program operation, as it slows down the normal program execution enormously (typically causing an 100–1,000-times overhead). This was expected, though, as the impact-computation cost for all possible queries was included in the run time of the analysis. Thus, the higher the impact-computation

TABLE 9: Comparison Between Offline and Online Dynamic Impact Prediction

Modality	Advantages	Disadvantages
Offline analysis (DIVER)	• one-off program execution • reusable traces for on-demand, arbitrary queries • shorter waiting time for a single query only • low runtime overhead	• additional time cost for tracing and trace processing • additional space cost for trace storage • longer total time for computing all queries • additional file-system resource consumption
Online analysis (DIVERONLINE)	• no tracing and trace-processing time cost • no trace storage cost • shorter time for computing impact sets of all queries • much less consumption of file-system resources	• possible need for executing program multiple times • on-demand querying: none or at high run-time overhead • longer waiting time for a single query only • additional program run-time slowdown

cost, the higher the overhead. The *One-by-One* online analysis, albeit mostly also causing higher slowdown than DIVER, did not drag the normal program operation as much as the *All-in-One* approach. In fact, the overhead was mostly below 10-times and the increase over the offline analysis was not dramatic either. With two exceptional cases, Ant and BCEL, the online analysis caused even lower overhead than the offline alternative, consistent with the observation in this regard from Figure 7.

Space cost. As we mentioned earlier, for the *small-trace* programs, the space-cost savings with DIVERONLINE relative to DIVER were mostly trivial because the total trace storage only needed 25MB on average (with the maximal of 84MB for OpenNLP 1.5) as seen in Table 4. However, with certain programs, long-running and/or complex ones for instance, the offline impact analysis would need to produce huge amount of traces, even to the extent the space cost may become a serious blockade for the analysis adoption. Table 8 (the rightmost two columns) shows the trace storage cost of DIVERONLINE versus DIVER. As anticipated, DIVERONLINE as an online analysis did not incur any space cost after the static-analysis phase, thus it did not produce any trace files either. In contrast, DIVER always incurred space cost in this regard, including not only the external storage space but also a number of files (as listed in the parentheses of the last column). Note that carrying a large number of files is not just clumsy, but can also consume limited file-system resources (e.g., file descriptors in Unix systems).

As shown, DIVER consumed the amount of storage and file-system resources (maximal of 2,650 trace files for Schedule1) that could be considered reasonable with the *small-trace* subjects. However, for the two added subjects, JABA-v1 and OpenNLP 1.6, the traces demanded over 15GB (for OpenNLP 1.6) and even up to 42GB storage with as many as 13K trace files (for JABA-v1). To avoid run-time memory blowup, we implemented DIVER to dump traces for each test case in possibly multiple segments such that each segment never gets over 3MB in size and is saved in a separate file. Alternatively, one may choose to buffer longer even full traces in memory and dump at the end of execution to fewer or even only one single trace file. However, for programs like JABA-v1 and OpenNLP 1.6, the alternative solution would very likely run out of memory. In contrast, an online approach like DIVERONLINE, regardless of the working mode, saves all these resources and associated costs in entirety. Thus, users who do not want to incur these large costs would better choose an online impact analysis over the offline approach. Moreover, when such costs become unaffordable, the online mode may become an *enabling* strategy for impact analysis.

Summary of empirical comparison. The online analysis and offline analysis each has its own merits and drawbacks, as we first discussed hypothetically in Section 4.4. Table 9 further sums up the contrast between the two approaches based on the empirical findings just discussed above. For a succinct comparison, quantitative measures presented earlier are not repeated here in the table. Generally, unless runtime slowdown is a major concern, DIVERONLINE (in the *All-in-One* mode) would be more desirable than DIVER in most situations because the online approach either saves total analysis time or saves storage costs. The rationale for the time saving with DIVERONLINE lies in that the extra time spent on impact computation during runtime (compared to DIVER) is readily paid off and is outweighed by the time that would be spent on tracing-induced file-system I/Os (i.e., serializing and deserializing the method-execution events at run time and post-processing time, respectively). The rationale for the storage-cost saving comes naturally from the nature of any online dynamic analysis: since DIVERONLINE does the analysis in parallel with the program execution, no tracing is performed thus no traces need to be stored in the file system; the method-execution events are discarded immediately after they are utilized for impact computation (albeit possibly buffered in memory shortly depending on the implementation/optimization strategies adopted). Important for the second rationale, the online impact-computation algorithm must be designed as such that all execution events are scanned in the order they occurred and every event is processed only once. According to Algorithm 2, DIVERONLINE satisfies this requirement thus it eliminates the need for tracing.

Note that, in practical scenarios, both the additional space cost incurred by DIVER (e.g., storage cost and file-descriptor usage) and the additional time cost incurred by DIVERONLINE (e.g., runtime slowdown with the *All-in-One* analysis) could become a serious obstacle for the adoption of these techniques. Yet, the tremendous runtime overhead of the *All-in-One* online approach might be counterbalanced by its great savings (by 50% on average) in the total time for querying all impact sets at once. The great savings, given its same level of precision as DIVER, imply large improvement in cost-effectiveness as well. While its *One-by-One* mode would not provide on-demand querying (the query needs to be known before program execution), the online analysis can do so in the *All-in-One* mode (by computing all impact sets and supplying results for specific queries in trivial time) albeit at the cost of causing large runtime slowdown.

All in all, the online approach would be favored in situations where the trace size explodes during execution (due to very-long loops, for instance), the consequent storage and/or file-system resource consumption may not be easily afforded, or it is preferred to compute all impact sets at one time faster. In contrast, the offline approach may be favored in scenarios where users mostly do ad-hoc impact-set querying for a specific method, the program under analysis does not produce large traces, or users could not afford high runtime overheads. (For online analysis, exploiting specialized runtime environments such as dynamic compiler technology [19] may reduce the run-time overheads.) Therefore, providing both (online and offline) options in a dynamic impact prediction solution would potentially better meet diverse needs.

Compared to the offline approach to dynamic impact prediction (with DIVER), the online approach (with DIVERONLINE) can generally at least either save the total analysis time (considerably in most cases) or save substantial storage cost (even to the extent that it becomes an enabling option for the impact analysis). In all, the online analysis and offline analysis complement to each other, together meeting diverse needs hence making our holistic solution more practically adoptable and useful.

11 THREATS TO VALIDITY

The main *internal* threat to the validity of our results is the possibility of having implementation errors in DIVER and PI/EAS$_C$. However, the implementation of both techniques is based on Soot and DUA-FORENSICS that have matured over many years. In addition, we manually checked individual results of every phase of DIVER and PI/EAS$_C$ for our study subjects. Especially for the post-processing phase, we manually verified the impact-computation algorithm of DIVER against each subject using a few sample queries. One more *internal* threat lies in the possible errors in our experimental and data-analysis scripts, including the experimentation tools used for Study III. To reduce this risk, we tested and debugged those scripts and tools, especially the source-code differencer that finds methods changed between two revisions, and checked their operation for each relevant experimentation step.

Yet another *internal* threat is, when computing the dependence graph for DIVER, the risk of missing static dependencies due to Java language features such as reflection, multi-threading, and native methods. However, we confirmed that, for our study subjects running on the test suites we utilized, there was no use of such features except for native methods in Java libraries, which are modeled *conservatively* by DUA-FORENSICS [58], and reflection APIs in JMeter, BCEL, and Ant, the only three subjects, out of the ten we utilized, that used reflection. In these three subjects, the reflection calls targeted only methods whose names are (or can be readily resolved to) constant strings, which our analyses fully handled through simple string analysis (mainly by intraprocedural string-constant propagation [50]). Thus, for the chosen subjects and test inputs, our analyses were not much affected by these features that usually lead to unsoundness. Yet, in general, Java programs may include more sophisticated uses of features and/or constructs that our analyses do not fully handle. Therefore, it is under the constraints against typical dynamic language features (including reflection API and native method calls, dynamic loading, and customized class loaders) that our analyses produce safe results (however, our analyses are still considered *soundy* even with those features used, as discussed earlier). Automatically identifying these constraints in a given program would be a promising direction of future work. As another immediate next step, we plan to empirically assess the impact of unsoundness resulted from the constraints on the effectiveness of our techniques. Our analyses currently do not deal with multi-threading in general: concurrent program impact analysis would be addressed as a separate line of effort [17]. Finally, for Schedule1, we used a version translated from C to Java, so we verified that the outcomes of all 2,650 test cases remained the same as those from the original program. There could still be potential biases (in terms of code structure and program behaviors with respect to the original C version) introduced in the translation, though. We chose this subject mainly because it well represents a real-world, small program/module in a distinct domain while coming with a large number of test cases—we intended to include subjects of diverse domains and scales to maximize the representativeness of our evaluation results.

The main *external* threat to the validity of out study is our selection of Java subjects and their test suites. This set of ten subjects does not necessarily represent all types of programs from a dynamic impact-analysis perspective. To address this threat, we picked our subjects such that they were as diverse as possible in size, type of functionality, coding style, and complexity. Additionally for the repository-query study (Study III), the three repositories may not be generally representative of all such repositories, and the chosen ranges of revisions may or may not represent the entire respective repositories for the three subjects. To reduce this threat, we purposely chosen repositories of a variety of source scales and from varying application domains, and for each we studied a reasonably large number of revisions.

Another *external* threat is inherent to dynamic analysis in general: the test suites we used cannot exercise all behaviors of the respective subjects. Many methods were not covered by the test suites, so we could not apply the techniques to those methods. In addition, some of subjects (e.g., PDFBox and ArgoUML) had test inputs of relatively low coverage. Thus, our results must be interpreted in light of the extent of the ability of those test suites to exercise their subjects. The test suites we used were also mainly focused on functionality testing, with which the program behaviours exercised might deviate from those exercised by field data [14] (i.e., executions collected from actual usage at real users' sites). To mitigate these issues, we chose subjects for which reasonably large and complete test suites were provided—at least from a functional point of view. Also, most of these subjects and test suites have been used by other researchers in their studies, and therefore they can be seen as good benchmarks. For the subjects used, we presently do not have access to their field data. Thus, we currently limited our program execution traces to those from the functional test inputs. However, it would be rewarding to explore the evaluation of our approach with respect to field data, which is part of our future work. Another future line of research is to empirically and systematically investigate the effects of the sizes and coverage of test inputs on the effectiveness of dynamic impact prediction techniques including our techniques. As a preliminary evidence, our result showed no clear correlation between test suite size and the effectiveness of our approach (see Table 2). Importantly, the goal of the our techniques is to provide more precise hence more usable impact sets at reasonable costs for *the provided executions*, thus improving the quality of test inputs (e.g., coverage) is primarily a direction orthogonal to our approach.

For the study with multiple-method queries, one more possible *external* threat comes from the range of query sizes we considered. It is possible that the present range of sizes from one to ten is not typical of actual numbers of methods changed in one repository commit by developers in practice. Unfortunately, it is not feasible to exhaust all possible query sizes. Nevertheless, comparative results from ten different query sizes can still offer considerable evidence for understanding the overall comparative effectiveness of the two techniques compared and its variation with changing query sizes. Also, note that for two of the three repositories we chose, the average number of code-changing methods per revision is just about ten or fewer, among the studied revisions at least. And additional threat lies in our simplified treatment of multiple-method changes dismissing the interaction between changes across multiple methods. Instead, we simply unionized the single-query

results for all member methods in the such queries for a safe approximation, which seems appropriate for a *predictive* analysis.

The main *construct* threat concerns the design and methodology of our studies. Since we have no relevant knowledge about the comparative importance of methods in each subject, we implicitly treat each method equally in terms of its weight or importance with respect to others in the subject. As a result, in cases where developers see higher weights with certain methods than others, the precision improvements from DIVER experienced by developers might not be exactly the same in practice as we reported. In addition, we have translated the impact-set size ratios into relative precision improvement between DIVER and the baseline, which is based on the assumption that both techniques are soundy. While the soundness of a dynamic analysis is indeed defined with respect to the test inputs (accordingly the executions generated from those inputs) utilized by the analysis [57], and in that sense both DIVER and PI/EAS$_C$ impact sets are safe, neither of the two techniques can guarantee perfect recall for predicting the impacts that *actual* changes will have if the changes modify control flows: the technique does not know about those changes.

An additional *construct* threat is about the way we attempt to identify the approximate lower bound of the effectiveness (relative precision) of DIVER. We picked, from the pool of subjects for the arbitrary-query studies that have a considerable length of revision history, three subjects for which DIVER had the worst-case effectiveness, and then used more recent revisions from their online repositories for Study III, from which we assess the lower bound. This decision is based on the assumption that, for *all* the subjects that have available and usable repositories, changes in their later versions do not affect much the performance of DIVER relative to PI/EAS$_C$, so that their being the worst cases among all in that pool is still valid. While we cannot guarantee about this assumption, the nature of these subjects should be reasonably stable during a relatively short length of evolution: in fact, for two of the three subjects, the versions utilized in Study III do not depart quite away from the one used in the previous two according to the size differences. Nevertheless, we do not intend to claim that the lower bound estimated from the repository-queries study can generalize for DIVER and PI/EAS$_C$.

Finally, a *conclusion* threat to validity is the appropriateness of our statistical analyses. To reduce this threat, we used a non-parametric hypothesis test and a non-parametric effect-size measure which make no assumptions about the distribution of the data (e.g., normality). However, due to the nature of the Cliff's Delta method, the interpretation of the effect sizes based on that method is still subject to the assumption about the normality of underlying data points. Therefore, the meaning of those effect sizes can only be taken with respective constraints. Another *conclusion* threat is that we could only compute meaningful impact sets for methods that have at least been covered by one test input. As a result, our statistical analyses are applied to those queries only. To reduce this threat, we considered all possible such queries in each of the three studies with respect to its query space and experiment goals.

12 RELATED WORK

In [35], we reported preliminary results on DIVER, including the impact-computation algorithm that combines static dependencies and dynamic method events. This paper substantially expanded that work by presenting more details of the technique. Furthermore, the scale of the empirical evaluation has been largely extended, including the increase in the number of subjects from four to ten, and two additional studies (using multiple-method and repository-based queries). Recently, on top of DIVER, we developed a framework for dependence-based dynamic impact analysis [46], [78]. In contrast, while this work continues to focus on exploiting static program dependencies to attain much higher precision than existing alternative options for dynamic impact analysis, that framework explores a disparate direction—using more dynamic information beyond method-execution traces—to provide multiple cost-effectiveness tradeoffs of the analysis. Different from the central goal of reducing imprecision hence impact-inspection effort in this work, the framework aims at providing users with flexibility in tool selection so as to meet diverse needs with various use scenarios and/or varied resource budgets [78].

Other previous work related to ours includes slicing and dependence approximation, in addition to impact analysis utilizing various artifacts, such as execution order, structural information, and static dependencies.

Inferring execution order. As an initiative in method-level dynamic impact analysis, the PATHIMPACT [15] technique introduced by Law and Rothermel computes dynamic impacts purely on the basis of execution order of methods by recording the occurrences of method-entrance and method-exit events. Later, they extended this approach to accommodate impact analysis of a series of evolving versions of a program such that impact sets for later revisions can be incrementally computed to obtain an overall optimized efficiency [21].

On the other hand, to optimize PATHIMPACT for a single program version, Apiwattanapong and colleagues present execute-after sequences (EAS) [18] which avoids tracing full sequences of method events so as to improve both time and space efficiency of the analysis without loss of precision relative to PATHIMPACT. Two more follow-up techniques are proposed by Breech and colleagues also aiming at better efficiency over PI/EAS, such as the online version relying on dynamic compilation techniques [19] and another optimization which computes impact sets for all functions in one pass of trace scan [20].

To develop DIVER, we employe the full sequence of method-execution events as the single form of dynamic information for impact analysis as PATHIMPACT did. However, DIVER also exploits static program information that was not utilized by PATHIMPACT or EAS. In comparison to other similar techniques following up PI/EAS, the main goal of DIVER is to improve the precision of method-level dynamic impact analysis while maintaining a practical level of efficiency, rather than only enhancing the efficiency (i.e., reducing computation time and storage costs) of PI/EAS as targeted by those techniques.

Exploiting structural information. Other than method execution order, various types of structural information of programs have also been employed by many researchers for impact analysis. Orso and colleagues present COVERAGEIMPACT [14] which uses method-level test coverage and static forward slicing. It was found to achieve greater efficiency but lower precision than PI/EAS [5]. Similar to EAS, static execute-after approaches (SEA) [12] also use execution orders to find impacted methods but based on the static call graph [79] instead of runtime traces.

Other static impact analyses exist [1], [10], [17], including those based on static slicing [80], coupling measures [6], change-type classification [81], and concept lattices [9], [82]. Static impact analyses are more conservative than dynamic ones, which makes them less precise but more general. DIVER also performs a static

analysis—for the dependence graph construction—but, instead of producing a conservative impact set directly from that graph, it uses the dependence model to guide the analysis of runtime traces for precise dynamic-impact analysis.

Impact-analysis techniques based on dependence analyses other than static slicing have been explored as well. Sun *et al.* propose OOCMDG [81] and LoCMD [9] to model dependencies among classes, methods, and class fields, which are directly derived from class-method memberships, class inheritance, method-call relations, and use relations between classes and methods. Their techniques are also extended for impact analysis with hierarchical slicing [83] and for multiple levels of granularity [84].

Unlike these approaches where only structural dependencies (e.g., call edges) based on object-oriented features are considered, we model all interprocedural data and control dependencies in the DIVER dependence graph. CALLIMPACT [85] computes change impacts with information on variable definitions and uses but, like SIEVE [29] and CHIANTI [7], it is a *descriptive* impact analysis: They rely on the knowledge of actual changes made between two program versions to compute impact sets. DIVER is a *predictive* technique, using potential change locations but assuming no knowledge about actual changes to be made there.

Incorporating static dependencies. Similar to DIVER, INFLUENCEDYNAMIC [16] combines dependence analysis and dynamic information for impact analysis. However, it considers dependencies among methods with respect to parameter passing, returned values, and global variables only, a subset of the dependencies that DIVER models. Using such partial dependencies, it is only able to prune very few false-positive impacts relative to EAS. As a result, the precision improvement it gained over PI/EAS is merely marginal yet at much greater cost than EAS. In addition, its design and implementation suggest that this technique primarily addresses (and is directly applicable only to) procedural programs (while our approach targets object-oriented software). Its experimental environment (e.g., GCC with customizations) is not compatible with ours either. Thus, it is difficult to reproduce or compare to the results of the technique, as would be required for including it in our empirical studies.

Huang and Song [26] extend INFLUENCEDYNAMIC for objected-oriented programs by adding dependencies between fields, as the RWSETS tool does [86], and improve it further by including dependencies due to runtime class inheritances [27]. Another hybrid technique, SD-IMPALA [28], extends its pure static-analysis-based version IMPALA [4] to improve the recall of impact analyses by utilizing the runtime traces of method events and field accesses. Unfortunately, like INFLUENCEDYNAMIC, these approaches also model partial data dependencies only and none of them achieve a significantly higher precision than PI/EAS.

In contrast, DIVER uses the complete set of static dependencies of the program under analysis, although for efficiency purposes it *approximates* instead of directly adopting the fine-grained dependence model (i.e., the system dependence graph [11]). Also, different from INFLUENCEDYNAMIC and its follow-up techniques, our approach targets and empirically achieved a significantly more precise dynamic impact analysis. Since it is known that those previous techniques were unable to improve against PI/EAS significantly, and they either target different environment (e.g., INFLUENCEDYNAMIC for procedural programs) or would require considerable extra efforts to implement (none of them have implementation available to us), we could not include them in our empirical comparisons but instead take PI/EAS as the baseline technique for our evaluation purposes.

Slicing. Forward dynamic slicing [87] is, at first glance, an option for dynamic impact analysis, but it can be too expensive for that purpose [18], [21]. For method-level impact analysis, dynamic slicing would have to be applied to most, if not all, statements inside the queried method. Apparently, this fine-grained dynamic dependence analysis would be overly heavyweight for method-level impact analysis. In contrast, DIVER performs this task much more efficiently and directly at the method level.

For static impact analysis, program slicing (static) has been directly applied to impact analysis [80]. In contrast to this and other static approaches, DIVER follows a dynamic approach focusing on concrete program executions to provide impact sets more relevant to the specific program behaviour with respect to those executions. Also, compared to the statement-level impact analysis based on static slicing, DIVER produces method-level impact sets, although both utilize fine-grained (statement-level) static dependencies.

With sensitivity analysis and execution differencing [88], a quantitative approach to slicing [56] was proposed to predict and prioritize dynamic impacts based on semantic dependence quantification. To approximate the semantic dependencies from which the quantitative impact sets are immediately derived, the approach emulates a number of N changes to the query location in the program under analysis at runtime, and identifies impacts from the differences between the original program execution trace and the traces of N modified executions. It further quantifies the impacts according to the times each impacted entity appeared in the execution differences (referred to as impact frequency) and then ranks the impacts by their frequencies. The motivation was to prioritize (through the ranking process) the entities in the impact set that can be too large to fully inspect, hence allowing users to prioritize their limited budget to focus on the most likely impacted entities first. In contrast, this work focuses on reducing the impact set produced by imprecise impact-prediction approaches through utilizing static program dependence information, without differentiating the entities in the impact set. Also, unlike our method-level analysis, the quantitative impact-prediction technique works at statement level. On the other hand, cutting off false-positive entities of an impact set and prioritizing all entities within the impact set share the common goal of reducing the effort of impact inspection. Thus, the approach proposed in this paper is complementary to the impact prioritization approach in [56].

Dependence approximation. In [89], an abstract system dependence graph (ASDG) is proposed to directly model method-level dependencies so as to assist programmers with code comprehension. Compared to the ASDG which is built by first constructing the full SDG [11] and then simplifying statement-level edges to capture only dependencies between corresponding methods, we dismiss context-sensitivity of the original SDG when creating the DIVER dependence graph. Thus, the static dependencies in DIVER are cheaper to compute than those in the ASDG. Other approaches directly modeling method-level dependencies exist (e.g., [90]), which may benefit a method-level dynamic impact analysis. At present, DIVER utilizes statement-level static dependencies with method-execution traces to essentially compute method-level *dynamic* dependencies for impact analysis. It would be of interest to investigate using method-level static dependencies directly for the same purpose but even higher efficiency, as part of future work.

13 CONCLUSION

Dynamic impact analysis provides a key approach to understanding the impact of one set of program entities on the rest of the program hence a fundamental technique for reliably evolving software. However, this technique has not been extensively adopted in practice. Different studies have shown that, among others, a major obstacle is the lack of effective tool supports for this analysis. We thus investigated the accuracy of existing such techniques [34] and revealed that the PI/EAS approach, which represents almost the most effective technique in the literature, is highly imprecise. This imprecision is not surprising, though, given its overly conservative nature. To make dynamic impact analysis practically useful and adoptable, much better effectiveness is needed. More precise impact analysis also potentially leads to increased effectiveness of its applications, including testing, debugging, comprehension, and various other dependence-based tasks.

We presented a new hybrid dynamic impact analysis DIVER which combines approximate static dependencies and method-level execution traces. DIVER tracks impact propagation and prunes static dependencies that are not exercised by the method-execution traces, so as to achieve, at acceptable overheads in computation time and data storage, largely improved effectiveness over the baseline PI/EAS. We also explored in detail the online approach to dynamic impact prediction by developing and evaluating a technique DIVERONLINE and investigated the pros and cons of online versus offline dynamic impact analysis on a common basis. Our comprehensive evaluation showed that DIVER can drastically remove false positives from the impact sets produced by the baseline approach, translating to an average improvement in precision by 100% or above. Meanwhile, DIVER keeps the overheads of the analysis practically acceptable. Thus, it attains a much higher level of cost-effectiveness than the existing options of the same kind. Our study comparing offline versus online analysis using DIVER and DIVERONLINE reveals that, when built on the same basis (algorithmic rationale and program information utilized) without specialized runtime environments, the online approach can be significantly (50% on average) more efficient while giving the same precision in contrast to offline approach for computing the impact sets of all queries in a one-off execution. Also, in cases where trace storage becomes challenging, the online approach also could be more preferable. Yet, these benefits of online analysis come at the cost of higher runtime overheads. By offering both options, we provide a more adoptable solution to developers that accommodates diverse use scenarios.

We identified several rewarding directions to explore next in discussing the limitations of our approach (Section11). As an additional future work, for complex or long-running applications where DIVER might be too expensive, we plan to make greater use of abstraction at method level and on the traces [91] to balance effectiveness and scalability beyond the option of online analysis (e.g., modeling static dependencies at method level directly [53] and then use the model together with method-execution events [92] to capture run-time method-level dependencies).

ACKNOWLEDGMENTS

The preliminary formulation of this work was substantially benefited from valuable discussion with Dr. Raul Santelices and was sponsored by ONR Grant N000141410037 to the University of Notre Dame. The author would also like to thank the anonymous reviewers for their thoughtful comments that have helped greatly with the improvement of this article through revisions, as well as the faculty startup fund given by Washington State University that supported the continuation of the research.

REFERENCES

[1] 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.

[2] L. C. Briand, J. Wuest, and H. Lounis, "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems," in *Proceedings of IEEE International Conference on Software Maintenance*, Aug. 1999, pp. 475–482.

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

[4] L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Damasio, "On the precision and accuracy of impact analysis techniques," in *Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science*, 2008, pp. 513–518.

[5] 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.*, Edinburgh, Scotland, may 2004, pp. 491–500.

[6] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, 2009.

[7] 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.

[8] M. Sherriff and L. Williams, "Empirical Software Change Impact Analysis using Singular Value Decomposition," in *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, Apr. 2008, pp. 268–277.

[9] X. Sun, B. Li, S. Zhang, C. Tao, X. Chen, and W. Wen, "Using lattice of class and method dependence for change impact analysis of object oriented programs," in *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 2011, pp. 1439–1444.

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

[11] 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.

[12] J. Jasz, A. Beszedes, T. Gyimothy, and V. Rajlich, "Static execute after/before as a replacement of traditional software dependencies," in *IEEE International Conference on Software Maintenance*, 2008, pp. 137–146.

[13] L. Li and A. J. Offutt, "Algorithmic analysis of the impact of changes to object-oriented software," in *Software Maintenance 1996, Proceedings., International Conference on*. IEEE, 1996, pp. 171–184.

[14] 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.

[15] 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.

[16] 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.

[17] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.

[18] 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. Engg.*, May 2005, pp. 432–441.

[19] B. Breech, A. Danalis, S. Shindo, and L. Pollock, "Online impact analysis via dynamic compilation technology," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004, pp. 453–457.

[20] B. Breech, M. Tegtmeyer, and L. Pollock, "A comparison of online and dynamic impact analysis algorithms," in *Proceedings of Ninth European Conference on Software Maintenance and Reengineering*, 2005, pp. 143–152.

[21] 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.

[22] 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.

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

[24] 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. [Online]. Available: http://doi.acm.org/10.1145/93542.93576

[25] X. Zhang, R. Gupta, and Y. Zhang, "Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04, Washington, DC, USA, 2004, pp. 502–511.

[26] L. Huang and Y.-T. Song, "Precise dynamic impact analysis with dependency analysis for object-oriented programs," in *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications*, 2007, pp. 374–384.

[27] ——, "A dynamic impact analysis approach for object-oriented programs," *Advanced Software Engineering and Its Applications*, vol. 0, pp. 217–220, 2008.

[28] M. C. O. Maia, R. A. Bittencourt, J. C. A. de Figueiredo, and D. D. S. Guerrero, "The hybrid technique for object-oriented software change impact analysis," in *14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 252–255.

[29] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Sieve: A tool for automatically detecting variations across program versions," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, 2006, pp. 241–252. [Online]. Available: http://dx.doi.org/10.1109/ASE.2006.61

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

[31] 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.

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

[33] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *Proc. of Int'l Conf. on Autom. Softw. Eng.*, Sep. 2008, pp. 218–227.

[34] H. Cai and R. Santelices, "A Comprehensive Study of the Predictive Accuracy of Dynamic Change-Impact Analysis," *Journal of Systems and Software (JSS)*, vol. 103, pp. 248–265, 2015.

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

[36] V. Rajlich, "Changing the paradigm of software engineering," *Communications of the ACM*, vol. 49, no. 8, pp. 67–70, 2006.

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

[38] 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*. ACM, 2006, pp. 492–501.

[39] 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*. ACM, 2008, pp. 241–250.

[40] 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. 51:1–51:11.

[41] 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.

[42] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 185–194.

[43] M. Acharya and B. Robinson, "Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems," in *Proceedings of IEEE/ACM International Conference on Software Engineering, Software Engineering in Practice Track*, May 2011, pp. 746–765.

[44] 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.

[45] M. Ajrnal Chaumun, H. Kabaili, R. K. Keller, and F. Lustman, "A change impact model for changeability assessment in object-oriented software systems," in *Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on*. IEEE, 1999, pp. 130–138.

[46] H. Cai and R. Santelices, "A Framework for Cost-effective Dependence-based Dynamic Impact Analysis," in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 231–240.

[47] 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.

[48] 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.

[49] S. Sinha, M. J. Harrold, and G. Rothermel, "Interprocedural control dependence," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 209–254, Apr. 2001. [Online]. Available: http://doi.acm.org/10.1145/367008.367022

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

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

[52] S. Sinha and M. J. Harrold, "Analysis and testing of programs with exception handling constructs," *Software Engineering, IEEE Transactions on*, vol. 26, no. 9, pp. 849–871, 2000.

[53] H. Cai and R. Santelices, "Abstracting program dependencies using the method dependence graph," in *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 49–58.

[54] 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.

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

[56] H. Cai, R. Santelices, and S. Jiang, "Prioritizing change-impact analysis via semantic program-dependence quantification," *IEEE Transactions on Reliability*, vol. 65, no. 3, pp. 1114–1132, 2016.

[57] D. Jackson and M. Rinard, "Software analysis: A roadmap," in *Proceedings of the Conference on the Future of Software Engineering*. ACM, 2000, pp. 133–145.

[58] 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.

[59] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "Soot - a Java Bytecode Optimization Framework," in *Cetus Users and Compiler Infrastructure Workshop*, Oct. 2011.

[60] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer, "A new foundation for control dependence and slicing for modern program structures," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 5, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1275497.1275502

[61] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhotak, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Moller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Commun. ACM*, vol. 58, pp. 44–46, 2015.

[62] "Soundiness statement generator," http://soundiness.org/#Generator, 2017, [Online; accessed 01-23-2017].

[63] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proc. of Int'l Conf. on Softw. Eng. (ICSE 94)*, 1994, pp. 191–200.

[64] 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.

[65] "Java Architecture for Bytecode Analysis," http://www.cc.gatech.edu/aristotle/Tools/jaba.html, 2005.

[66] "Apache Commons BCEL," http://projects.apache.org/projects/commons_bcel.html, 2012.

[67] "Apache PDFBox," http://pdfbox.apache.org/, 2010.

[68] "Apache OpenNLP," http://opennlp.apache.org//, 2013.

[69] "The ArgoUML Project," http://argouml.tigris.org/, 2003.

[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] F. Mosteller and R. A. Fisher, "Questions and answers," *The American Statistician*, vol. 2, no. 5, pp. 30–31, 1948.

[72] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 1996.

[73] R. Coe, "It's the effect size, stupid: What effect size is and why it is important," in *the Annual Conference of the British Educational Research Association*. Education-line, 2002.

[74] H. Cai and R. Santelices, "TracerJD: Generic Trace-based Dynamic Dependence Analysis with Fine-grained Logging," in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, 489–493.

[75] Á. Beszédes, C. Faragó, Z. M. Szabo, J. Csirik, and T. Gyimóthy, "Union slices for program maintenance," in *Software Maintenance, 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 12–21.

[76] M. Harman, D. Binkley, K. Gallagher, N. Gold, and J. Krinke, "Dependence clusters in source code," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 1, pp. 1–33, 2009.

[77] J. I. Maletic and M. L. Collard, "Supporting source code difference analysis," in *20th IEEE International Conference on Software Maintenance*. IEEE, 2004, pp. 210–219.

[78] H. Cai, R. Santelices, and D. Thain, "DiaPro: Unifying dynamic impact analyses for improved and variable cost-effectiveness," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 2, pp. 1–50, 2016.

[79] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: a control call graph based technique," in *Proceedings of Asia-Pacific Software Engineering Conference*, 2005, pp. 1–9.

[80] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Proceedings of the 33rd ACM SIGSOFT International Conference on Software Engineering (ICSE), Software Engineering in Practice Track*, May 2011, pp. 746–765.

[81] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, "Change impact analysis based on a taxonomy of change types," in *2010 IEEE 34th Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 2010, pp. 373–382.

[82] P. Tonella, "Using a concept lattice of decomposition slices for program understanding and impact analysis," *Software Engineering, IEEE Transactions on*, vol. 29, no. 6, pp. 495–509, 2003.

[83] X. Sun, B. Li, C. Tao, and S. Zhang, "HSM-based change impact analysis of object-oriented Java programs," *Chinese Journal of Electronics*, vol. 20, no. 2, pp. 247–251, April 2011.

[84] M. Petrenko and V. Rajlich, "Variable granularity for improving precision of impact analysis," in *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. IEEE, 2009, pp. 10–19.

[85] C. Gupta, Y. Singh, and D. S. Chauhan, "An efficient dynamic impact analysis using definition and usage information," *International Journal of Digital Content Technology and its Applications*, vol. 3, no. 4, pp. 112–115, 2009.

[86] M. Emami, "A practical interprocedural alias analysis for an optimizing/-parallelizing c compiler," Ph.D. dissertation, McGill University, 1993.

[87] M. Kamkar, "An overview and comparative classification of program slicing techniques," *J. Syst. Softw.*, vol. 31, no. 3, pp. 197–214, Dec. 1995. [Online]. Available: http://dx.doi.org/10.1016/0164-1212(94)00099-9

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

[89] Z. Yu and V. Rajlich, "Hidden dependencies in program comprehension and change propagation," in *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, 2001, pp. 293–299.

[90] J. P. Loyall and S. A. Mathisen, "Using dependence analysis to support the software maintenance process," in *Software Maintenance, 1993. CSM-93, Proceedings., Conference on*. IEEE, 1993, pp. 282–291.

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

[92] H. Cai and R. A. Santelices, "Method-level program dependence abstraction and its application to impact analysis," *Journal of Systems and Software*, vol. 122, pp. 311–326, 2016.