Dynamic Dependence Summaries

VIJAY KRISHNA PALEPU, GUOQING XU, and JAMES A. JONES, University of California, Irvine

Software engineers construct modern-day software applications by building on existing software libraries and components that they necessarily do not author themselves. Thus, contemporary software applications rely heavily on existing standard and third-party libraries for their execution and behavior. As such, effective runtime analysis of such a software application's behavior is met with new challenges. To perform dynamic analysis of a software application, all transitively dependent external libraries must also be monitored and analyzed at each layer of the software application's call stack. However, monitoring and analyzing large and often numerous external libraries may prove to be prohibitively expensive. Moreover, an overabundance of library-level analyses may obfuscate the details of the actual software application's dynamic behavior. In other words, the extensive use of existing libraries by a software application renders the results of its dynamic analysis both expensive to compute and difficult to understand. We model software component behavior as dynamically observed data- and control dependencies between inputs and outputs of a software component. Such data- and control dependencies are monitored at a fine-grain instruction-level and are collected as dynamic execution traces for software runs. As an approach to address the complexities and expenses associated with analyzing dynamically observable behavior of software components, we summarize and reuse the data- and control dependencies between the inputs and outputs of software components. Dynamically monitored data- and control dependencies, between the inputs and outputs of software components, upon summarization are called *dynamic dependence summaries*. Software components, equipped with dynamic dependence summaries, afford the omission of their exhaustive runtime analysis. Nonetheless, the reuse of dependence summaries would necessitate the abstraction of any concrete runtime information enclosed within the summary, thus potentially causing a loss in the information modeled by the dependence summary. Therefore, benefits to the efficiency of dynamic analyses that use such summarization may be afforded with losses of accuracy. As such, we evaluate the potential accuracy loss and the potential performance gain with the use of dynamic dependence summaries. Our results show, on average, a 13× speedup with the use of dynamic dependence summaries, with an accuracy of 90% in a real-world software engineering task.

CCS Concepts: • **Software and its engineering → Software testing and debugging;**

Additional Key Words and Phrases: Dynamic analysis, dependence analysis, summaries, dynamic slicing

ACM Reference Format:
Vijay Krishna Palepu, Guoqing Xu, and James A. Jones. 2017. Dynamic dependence summaries. ACM Trans. Softw. Eng. Methodol. 25, 4, Article 30 (January 2017), 41 pages.
DOI: http://dx.doi.org/10.1145/2968444

This work is supported by the National Science Foundation under grants CCF-1116943, CAREER CCF-1350837, CCR-0325197, CNS-1321179, CCF-140982, and CNS-1613023, and by the ONR under grants N00014-14-1-0549 and N00014-16-1-2913.
Authors' address: V. K. Palepu, 5243 Bren Hall, UC Irvine, Irvine CA 92697-3440; email: vpalepu@uci.edu; G. Xu, 3212 Bren Hall, UC Irvine, Irvine CA 92697-3440; email: guoqingx@ics.uci.edu; J. A. Jones, 5214 Bren Hall, UC Irvine, Irvine CA 92697-3440; email: jajones@uci.edu.

DOI: http://dx.doi.org/10.1145/2968444

1. INTRODUCTION

As the needs of society are increasingly accomplished with software systems, and those software systems become more complex and interrelated, software developers are, to an increasing extent, building components of software that interact with and build upon existing software components. Rather than writing all needed functionality from the low-level operating system to the high-level client interfaces, developers regularly use features that were developed by others, provided by components such as APIs, libraries, middleware, and infrastructures. Today's reality is a scenario that was predicted in the late 1960s by McIlroy [1968].

Prior research has identified some of the challenges that can be faced when depending on and assembling existing components, often called *components, off-the-shelf* (COTS). One common challenge to reusing third-party components is that analysis tasks become increasingly expensive as the extent and depth of component reuse increases (e.g., layer upon layer of transitive component reuse). To properly analyze the program, the effects of the underlying infrastructure and all of its layered components must be understood. As such, an analysis that is complete and exhaustive must analyze all transitively underlying components to determine how they affect the program under test.

Orso et al. [2001] discussed some of the challenges of performing analysis in the presence of external components and proposed abstract representations (i.e.,metadata) to provide information about component functionality. Later, Orso et al. [2007] extended these ideas for component metadata by specifying a concrete metadata scheme to enable regression test selection in the presence of components. Although Orso's solution for regression test selection provides a powerful solution for the specific task of regression test selection, the challenges of performing analysis in the presence of external components extend to many other (more heavyweight) dynamic software analysis tasks. For example, dynamic dependence analysis necessitates the tracing of data and control flows through all encountered libraries and components during the whole execution. Frequent profiling of methods in these large, and often numerous, libraries and components contributes extensively to the already heavy runtime costs, making the analysis often prohibitively expensive even for modestly sized software applications.

An important approach to reduce the costs of program analyses and provide such metadata is to summarize the behaviors of these components. Once generated, component summaries can be reused, or applied, during future executions of those components to improve the efficiency for a given program analysis. Indeed, the static program analysis community has extensively studied summary-based program analysis, with the development of various techniques that summarize procedural effects to achieve modular and efficient program analyses techniques (e.g., Salcianu and Rinard [2005], Rountev et al. [2008], Xu et al. [2009], Yorsh et al. [2008], and Dillig et al. [2011]). However, as shown by prior research (e.g., Korel and Laski [1990] and Zhang and Gupta [2004a]), static analysis can lead to overly conservative modeling of heap data effects, thus resulting in potential losses in accuracy in the underlying dynamic analysis.

In this work, with dynamic dependence analysis as an example, we dynamically compute dependence summary metadata that will characterize and capture external effects of reused components for a modern object-oriented language, where we treat each method as a component unto itself. We compute such summaries for methods that are ancillary to the development task at hand. Examples of such ancillary methods may include methods that are a part of standard libraries, external third-party code modules, or sections of a large software system that are not under test. Such dynamic dependence summary metadata can then be reused to model the external effects of such methods during their subsequent executions. (Note: The demarcation of methods as ancillary for a development task can be calibrated as needed by the developer.)

Dependence summaries are computed for a method by *dynamically* observing the control- and data dependencies within some representative executions of the method,

where each execution of the method results in a distinct dynamic dependence summary. Such a set of dynamic dependence summaries, for a given method, are then *abstracted* to remove any runtime data that is specific to their respective executions. Abstraction of dynamic dependence summaries partly enables the reuse of the summaries for modeling the external effects of subsequent executions of the method. That said, different executions of the same method may exhibit different external heap data effects (i.e., due to different control flow paths), thus potentially resulting in differences in the different abstracted dynamic dependence summaries for the same method. To generically model the varying external effects of such a method, the abstracted dynamic summaries are *aggregated* into a single dynamic dependence summary. The resulting aggregated dynamic dependence summary for a given method is used as a generic model of the behavior of any subsequent execution of the method in question.

Such a summary-based approach to dynamic dependence analysis leads to improved efficiency through reductions in execution trace sizes, trace-recording times, and dependence analysis times. Dynamic dependence analysis forms the basis for a variety of dynamic techniques, such as dynamic program slicing [Agrawal and Horgan 1990], bloat analysis [Xu et al. 2010], tainting-based information flow analysis [Newsome and Song 2005], and potential parallelism detection [Holewinski et al. 2012]. Thus, such a summary-based approach stands to improve the efficiency (in space and time) for all such techniques that rely on dynamic dependence analysis. Moreover, dynamic dependence summaries, due to their reliance on dynamic information, stand to model the external effects of dynamic dispatch, accesses of individual array elements, and dynamically observed control flow with potentially substantial levels of precision.

Such potential gains in efficiency, due to creating and using dynamic dependence summaries, are met with two potential sources of inaccuracies. First, a reliance on dynamic data inherently renders our approach and the resulting dynamic dependence summaries unsound, as a given set of executions of a method may not exhibit all possible external heap data effects. Second, the aggregation of multiple dependence summaries for a given method leads to a generic model of the external effects of a method's invocation. Such a generic model of a method's external effects, when used to describe the external effects of a specific invocation of the method, may introduce spurious dependence relationships. As such, an aggregate dynamic dependence summary introduces imprecisions within the dependence summary of a single invocation of a given method.

Given the trade-offs between efficiency and accuracy while using dynamic dependence summaries, an assessment of the applicability of using such summaries is necessary. The applicability of using such summaries is thus evaluated by studying the trade-off between efficiency and accuracy. We evaluated the accuracy of dynamic dependence summaries when modeling the future invocations of a given method, as a consequence of using dynamically observed information to construct dynamic dependence summaries and the aggregation step in our approach to build dependence summaries. To carry out such an evaluation, we implemented our summary-based dynamic dependence analysis and carried out empirical experimentation. We additionally studied the performance gains afforded by a summary-based approach as a result of relying on summarized components during the execution of the software. The first experiment in our evaluation investigates the extent to which eight real-world software systems rely on external components during the execution of the software systems. A second experiment examines the impact of using summarized metadata on actual performance costs, as against performing exhaustive analysis through all external components. The third experiment in our evaluation investigates the extent of accuracy losses owing to the process of creating summarized dependence metadata, with a purpose of reusing such metadata. We finally present a case study that assesses the impact of using such summarized metadata on dynamic slicing—a runtime technique that relies on dynamic

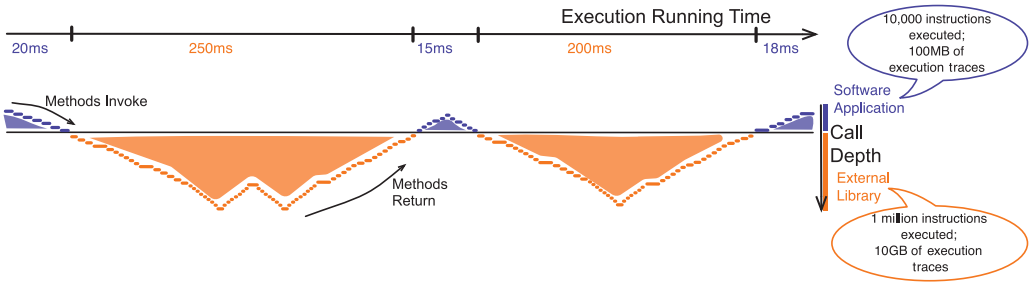


Fig. 1. Conceptual illustration of the profiling costs of library and application code.

dependence analysis. The results of our evaluation show an average speed up of 13× in the favor of a summary-based approach, with an accuracy of 90% in a real-world software engineering task.

This article expands on our previous conference paper [Palepu et al. 2013] in the following ways: (a) a more complete summarization analysis is now performed that models dynamic control dependencies in addition to dynamic data dependencies; (b) an algorithm for the aggregation of dynamic dependence summaries is presented for the first time; (c) a new experiment that assesses the accuracies of dynamic dependent summaries, independent of any runtime client analyses is presented for the first time; and (d) experimental data for two additional subjects is included, lending greater evidence of generalizability.

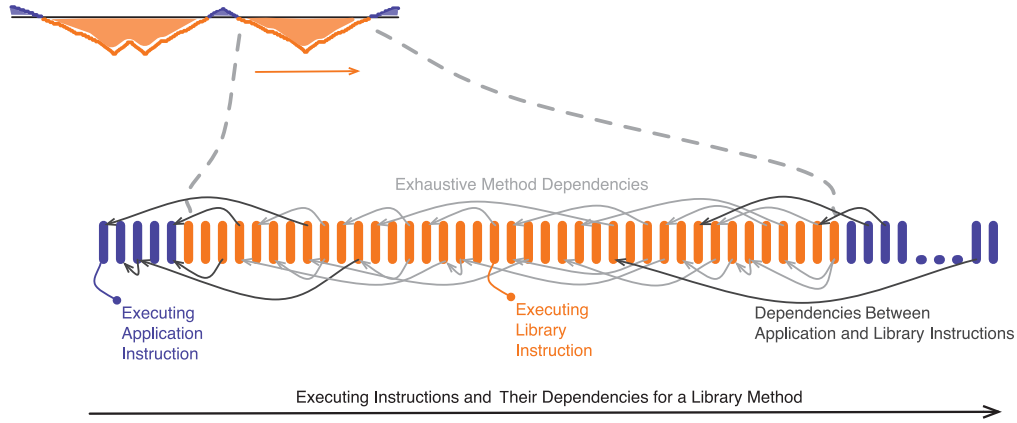
The main contributions of this work are the following:

(1) Definition of the technique for producing summarized dependence analysis results for software components, which can enable improved efficiency of subsequent analysis tasks.
(2) Exposition of the process by which component analysis results are captured, encoded as dynamic dependence summaries, and reused for future analysis tasks.
(3) Evaluations of the impact of utilizing such summary metadata for dynamic analyses in terms of both accuracy and efficiency.

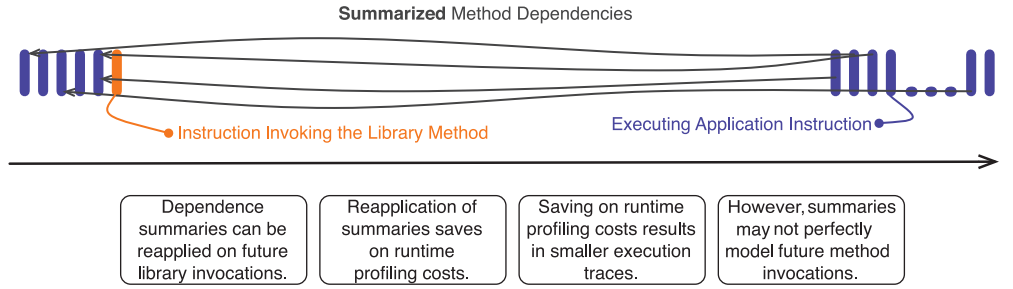
2. MOTIVATION

Profiling software executions is necessary to analyze dynamic dependencies. As such, our motivation for summarizing dependencies stems from the overwhelming and potentially prohibitive costs of profiling executions of standard and third-party libraries that support a software application's execution. In this section, we clarify this idea with the help of conceptual illustrations to motivate our work. To further support this idea, we present the results of a pilot study with real-world software subjects that illustrate the scale at which software libraries are typically used.

Software applications frequently execute methods from standard and third-party libraries, resulting in significant runtime spent in the execution of library code. To highlight this idea, Figure 1 pictorially illustrates four facets of an application's execution: (a) the chronology of executing instructions, progressing from left to right; (b) successive method invocations with a deepening call stack, progressing from top to bottom; (c) successive returns of method invocations, progressing from bottom to top; and (d) a colored distinction between the execution of application code (in blue) and library code (in orange). The application and library code are also separated by a threshold in the vertical call depth of the method call stack, with the blue application code above the threshold and the orange library code below.



(a) A conceptual illustration of the influence and dependence of executing instructions in a library method's invocation (in orange) on the execution of application code (in blue)



(b) Summarized method dependencies for a single execution of a library method

Fig. 2. Conceptual illustration of dynamic dependence summaries.

The wide and deep valleys of orange in Figure 1 that depict library code execution pictorially illustrate the significant execution time spent in executing library code with extensive and deep call stacks compared to the brief sections of blue that depict execution of application code. Although such standard and third-party libraries are ancillary to the actual implementation and development of a software application, these external libraries exert substantial influence on the eventual executions of the software application. For instance, consider the `put` and `get` methods of the `HashMap` collection from the Java standard library. Successive calls to the `HashMap` API can be used to store (`put`) and retrieve (`get`) data at various points in an execution using a `key` value, thus creating dynamic dependencies between the `key` and the data being stored. Entirely ignoring the effects of invoking the `HashMap` methods will clearly result in missed dependencies in a dynamic dependence graph. That said, applications often rely on libraries heavily during executions. As a result, profiling the execution of library code becomes an important and expensive facet of runtime analysis of software systems. This notion of a runtime dependence on external libraries is illustrated in Figure 2(a).

Figure 2(a) shows a "zoomed-in" illustration of the executing instructions within a library method call invoked from within the application code. The illustration shows how executing instructions are data and control dependent on instructions executed previously. In particular, the illustration depicts how executing library instructions depends on application instructions, and in turn executing application instructions are dependent on the execution of library instructions. Dependencies between the

execution of library and application instructions necessitate the runtime analysis of library instructions to comprehensively analyze the software application's executions. Moreover, such runtime analysis of library instructions cannot be ignored even as they substantially contribute to an application's runtime as depicted in Figure 1.

The goal of exhaustively profiling library invocations is to monitor their effects on the execution of the software application. As such, a natural idea to reduce the cost for analyzing library method invocations would be to summarize their effects. The summaries of the effects of library method invocations can then be reused for future library method invocations to model their influence on the software application's execution—without exhaustively profiling the library method invocations. This notion is illustrated in Figure 2(b), where instead of depicting all executed instructions of the library method invocation and their dependencies, like in Figure 2(a), only the summarized method dependencies and the execution of the application's instructions are shown.

Unsoundness and imprecision. It is important to note that the summary-based dynamic dependence analysis can introduce both unsoundness and imprecision while modeling dependencies between the inputs and outputs of future method invocations. On one hand, the quality of a method's summary relies on the coverage of the tests used to train the summary. Thus, a summary may miss certain dependence relationships due to the lack of test cases. On the other hand, the method summary aggregates information from multiple executions of the method. Thus, the application of the summary for a specific invocation may generate additional spurious dependence relationships that would not have been added in a regular dependence analysis. However, the original motivation of this work is to bring down the overwhelming and often prohibitive costs of execution profiling for dynamic dependence analysis. As such, the goal of this work with dynamic dependence summaries, despite their obvious potential for introducing inaccuracies, is to investigate this trade-off to determine if and when the inaccuracies can be tolerated to benefit runtime efficiency and thus make dynamic dependence analyses feasible.

Creating summaries statically. Further, it is worth noting that although this work looks at dynamically creating dependence summaries, such summaries can indeed be created statically. However, static analysis techniques are potentially conservative in modeling heap data effects, as suggested by prior research (e.g., Korel and Laski [1990] and Zhang and Gupta [2004a]). As such, method summaries that rely only on static analysis may render overly conservative approximations of runtime heap data effects, as in the cases of dynamic dispatch of polymorphic methods, access of individual array elements, or dynamically observed control flow. For example, method summaries built from static analysis are often imprecise in distinguishing among objects of different subtypes that share a common supertype, making it particularly difficult for use in a dynamic analysis. Additionally, statically built summaries are only possible with access to the method binaries, prior to their execution, which may not always be possible, such as in cases of metaprogramming where the components are loaded dynamically, from remote URLs through custom, user-defined loaders, or in situations where the binaries are rewritten on-the-fly (i.e., during the execution of the program).

Pilot Study. The preceding discussion was guided by conceptual illustrations to describe the potentially prohibitive expense associated with runtime profiling of external components for dynamic dependence analysis. To study this expense of analyzing software executions, we carried out a pilot study. As a part of this pilot study, we investigated the number of runtime instructions[1] that get executed in a typical execution of

[1]A "runtime instruction" or an "instruction instance" is a dynamic instantiation of a static program instruction. A single static program instruction can be executed multiple times and can lead to multiple runtime instructions.

real-world, large-scale software systems and would need to be recorded and analyzed. Additionally, to investigate how summarizing effects of external components can assist with the expenses of dynamic analysis, we also studied the number of instructions executed from within the Java standard library (i.e., `rt.jar`), which is essentially an external library. The following four real world, large-scale subjects were used for this pilot study: BLOAT (>41KLOCS), JYTHON (>245KLOCs), FOP (>102KLOCs), and PMD (>60 KLOCs). These large-scale subjects were obtained from the DACAPO performance benchmarks [Blackburn et al. 2006].

The results of our pilot study suggest that the executions of the large-scale software systems that we selected are composed of 563×10^6 runtime instructions on average, with the breakdown of runtime instructions for each software system as follows: BLOAT (391×10^6 runtime instructions), JYTHON ($1{,}734 \times 10^6$ runtime instructions), FOP (110×10^6 runtime instructions), and PMD (18×10^6 runtime instructions). A reasonably efficient scheme to record the execution of each runtime instruction in an execution trace would approximately require 160 bits or 20 bytes of memory—32 bits each to record the runtime instruction's thread ID, owner class, owner method, bytecode offset, and operand value.

Given this scheme and these subjects, an average execution of these benchmarks requires 10GB of disk storage to record an execution trace of nearly 563×10^6 runtime instructions. Apart from storing executions at such scale, analyzing them would also be a significant challenge unto itself.

When counting only those runtime instructions that were executed from the Java standard library (i.e., *rt.jar*), more than 490.75×10^6 runtime instructions were executed, with the breakdown for each software system as follows: BLOAT (239×10^6 runtime instructions), JYTHON ($1{,}658 \times 10^6$ runtime instructions), FOP (57×10^6 runtime instructions), and PMD (9×10^6 runtime instructions).

By summarizing only the lowest-level library (*rt.jar*), on average, nearly 65% of the costs associated with recording and analyzing runtime instructions may be eliminated. Such reductions are significant and can be made possible for dynamic dependence analyses by appropriately summarizing the effects of library-specific runtime instructions. Further, such projected reductions in space requirements were achieved when the Java standard library was treated as the sole external component for all executions. Such software applications often use several external libraries, thus presenting further avenues for summarization and potential cost savings.

3. OVERVIEW

We now present a high-level overview of our approach by the means of four principal challenges involved in computing and using dynamic dependence summaries. We also provide an overview of our resolutions for each challenge presented in this section. We discuss these challenges in the context of a simple software program and its execution that we use as a running example henceforth in this article. With such challenges as background, along with informal descriptions of our approach to resolve those challenges, we present the formal concepts that define dynamic dependence summaries in Section 4. In addition, we devote Section 5 to describe how such dynamic dependence summaries for method invocations can be used for a summary-based dynamic dependence analysis. We start by describing the running example, followed by discussions for the following principle challenges: (1) defining dependence summaries with objects, (2) abstracting concrete summaries, (3) accounting for varying method behavior, and (4) reusing dynamic dependence summaries.

Running Example. Figure 3(a) shows the implementation of the main application (lines 01 through 13) and the library `IntList` that supports the storage and retrieval

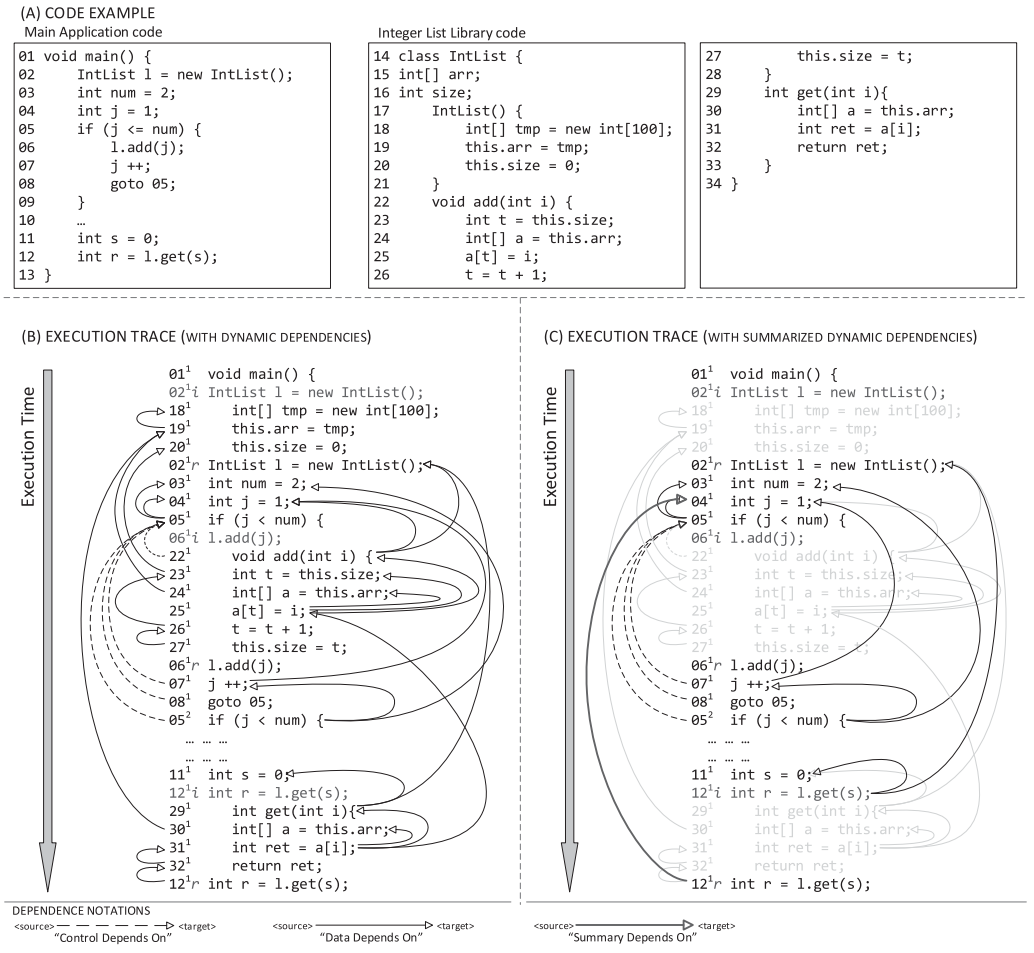


Fig. 3. Example program: code and execution trace (with dynamic dependencies).

of integer values (lines 14 through 34) in the form of a list. The main application iteratively adds integer values (lines 03 through 9) into the `IntList` object created in line 02. After the addition of integer values, lines 11 and 12 retrieve an integer value from the `IntList` object. The `IntList` library supports the creation of a new list of integers (lines 17 through 21), the addition of an integer value at the end of the integer list (lines 22 through 28), and the retrieval of an integer value at a designated position in the integer list (lines 29 through 33).

Example execution trace. Figure 3(b) shows the execution trace of a single run of the application, along with the use of the `IntList` library. Each line in the trace is an event depicting the execution of a single source code statement and is represented by the source code statement itself along with the line number of the source code statement, annotated with an integer i representing the i^{th} execution of the statement. (Note: A statement can be executed multiple times in a single program execution.) The execution starts with the invocation of the `main` method, as shown in the first line of the execution trace, and progresses "downward" with every succeeding line in the trace, as shown in

Figure 3. Execution events in the trace that depict invocation[2] of library methods are annotated with the alphabet "*i*" (e.g., $\boxed{06^1\ i\ \texttt{l.add(j);}}$). Similarly, the completion of library methods' execution, after their return, is depicted with the alphabetical annotation "*r*" (e.g., $\boxed{06^1\ r\ \texttt{r.add(j);}}$). Additionally, execution of methods and instructions within the `IntList` library are portrayed with textual indentations in the execution trace to better represent the execution of library instructions.

Furthermore, dotted and solid edges with triangular arrowheads are used to portray control and data dependencies, respectively, between different execution events in Figure 3(b). The dependence edges are drawn starting from the *dependent* execution event, with the edges ending (with the arrowhead) at the *dependee* execution event. For instance, the solid edge $\boxed{19^1} \longrightarrow \boxed{18^1}$ implies that execution event $\boxed{19^1}$ is data dependent on execution event $\boxed{18^1}$.

Challenge 1: Defining Dependence Summaries with Objects. Horwitz et al. [1990] pioneered summary-based dependence analysis, in which a summary edge of a procedure relates an input parameter i with an output parameter o, depicting a possible (direct or transitive) dependence of the computation of the value in output o on the value in input i. Such summary edges between a procedure's inputs and outputs abstracts away intermediate dependence relationships within the procedure.

However, modern object-oriented languages, such as Java, impose new challenges in modeling such procedure inputs and outputs and the relationships between them that existing techniques do not address. The notion of inputs and outputs for a method in an object-oriented language, like Java, is much broader than those discussed in Horwitz et al. [1990], because a method can access not only its arguments but potentially all objects reachable from the arguments.

In this work specifically, for a single instance of a method invocation, inputs and outputs are considered as a set of (heap or stack) locations where

(1) *input* locations exist before a given method invocation and can be read during the method invocation, and
(2) *output* locations can be written to during the method's invocation and are accessible after the completion of the method invocation.

Example. Figure 4 illustrates the dynamic dependence summary for the method invocation $\boxed{06^1\ i\ \texttt{l.add(j);}}$—the method invocation's inputs, outputs, and the dynamic dependencies between them. Figure 4 shows the input and output sets for the method invocation on the left- and right-hand sides, respectively. The input set contains the heap or stack locations that served as the method invocation's inputs. Similarly, the output set contains heap or stack locations that were modified during the method invocation and are available as its outputs. Arrows going from the output set to input set depict the dependencies (direct or transitive) between the inputs and outputs. Further, the *owner*-to-*member* relations between the different locations within the input or output sets are also depicted graphically with an arrow going from the owner object to the member. For instance, the relation $o^{02} \rightarrow size$ shows that *size* is a member of the owner object o^{02}. Such owner-to-member relations are also represented textually with a dot separator (.) between the owner and member (e.g., $o^{02}.size$).

Now, the invocation $\boxed{06^1\ i\ \texttt{l.add(j);}}$ for the method `void add(int i)` accepts as inputs the integer value j and receiver object l and does not explicitly return any value. However, a closer inspection suggests that the invocation $\boxed{06^1\ i\ \texttt{l.add(j);}}$ results in the

[2]In this work, unless mentioned otherwise, a "method invocation" or "invocation" refers to a specific instance in a series of runtime invocations of a single method within a program execution.

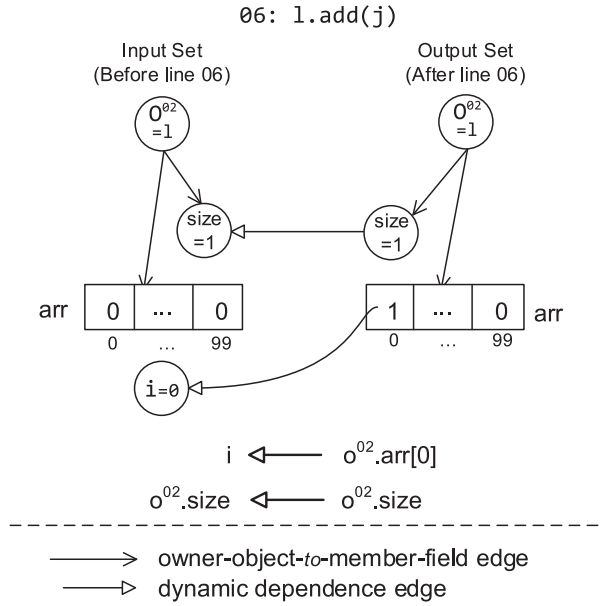


Fig. 4. Concrete dynamic dependence summary for the runtime invocation of the add method, as shown in the example program execution trace in Figure 3.

modification of the first element in the arr data structure (denoted as $o^{02}.arr[0]$ in Figure 4), and it updates the value in the field, $o^{02}.size$. As such, the array element ($o^{02}.arr[0]$) and the updated field ($o^{02}.size$) are available as outputs after the return of the method. Moreover, the initial value of the field $o^{02}.size$ served as an input toward the field's update during the method invocation, even though it was not passed as an actual argument to the method. Essentially, the inputs and outputs for a specific method invocation are not restricted to the method's actual arguments or a possible return value. A method's dependence summary should model such method inputs and outputs, along with the dependencies between them, using the method invocation's constituent data and control flow.

Critically, each location in the input and output sets (e.g., *size*), where necessary, is modeled using a combination of a root object (e.g., o^{02}) and an *access path* that specifies how the location is accessed from the parameter (e.g., $o^{02}.size$) through a series of member dereferences. Such modeling of inputs and outputs, and the dynamic dependencies therein, would yield effective applications of dependence summaries for downstream client analyses. The concepts of inputs, outputs, and access paths are formally defined in Section 4.

Challenge 2: Abstracting Concrete Summaries. Figure 4 depicts the inputs, outputs, and the dependencies between them for a specific method invocation within a program execution. We call such a model of dependencies that is tied to a specific method invocation a *concrete dynamic dependence summary*, or simply *concrete summary*, as formally defined in Section 4. For example, the object o^{02} is assigned to the concrete value l in the concrete dependence summary, as shown in Figure 4. Since the value l is tied to the specific method invocation $\boxed{06^1\ i\ \texttt{l.add(j);}}$, it might not be valid for a different invocation of the method `void add(int i)`. Hence, it follows that such concrete information in a dependence summary that is specific to a single invocation of a

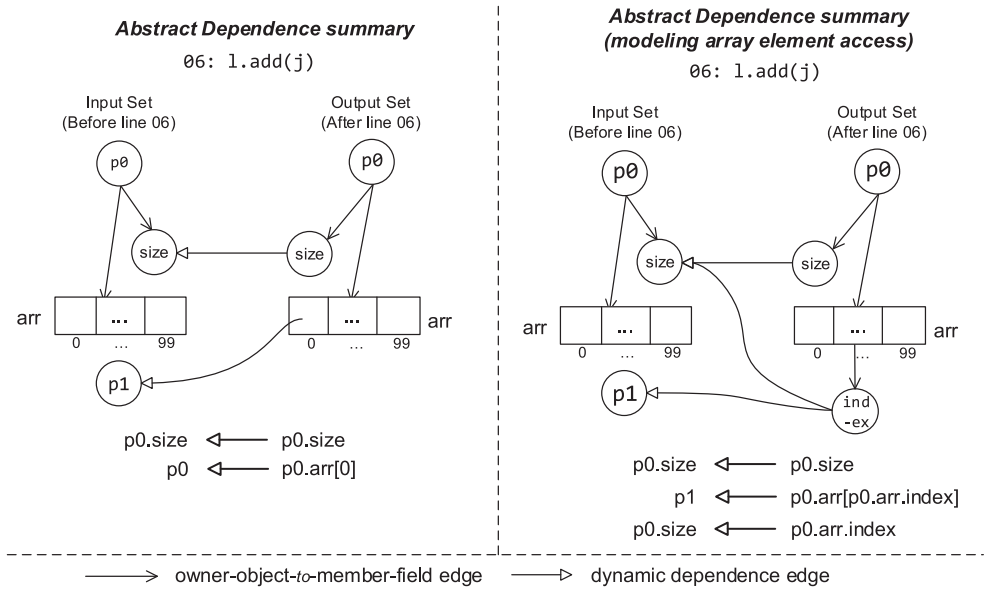


Fig. 5. Abstract dynamic dependence summaries, with and without modeling array element access, for the runtime invocation of add method at line 06, as shown in the example program execution in Figure 3.

given method prevents us from reusing or *applying* the summary for other invocations of the method in question.

To enable the reuse of a concrete summary, the information tied to a specific method invocation should be replaced with abstract information that may be applicable to all possible invocations of the method. In turn, the substitute abstract information should allow translation back to the concrete information that is specific to other invocations of the given method. In other words, the abstraction of concrete summaries should, in part, allow application of a method summary for all invocations of the method instead of specific invocations.

Abstraction of method summaries is performed in this work by using symbolic names for method arguments. As illustrated in Figure 4, the actual arguments for the method-invocation $\boxed{06^1\ i\ \texttt{l.add(j);}}$, at line 02, are the concrete object l and the concrete variable j. Upon abstraction of the concrete dynamic dependence summary for the given method invocation, the concrete object l and the concrete variable j are replaced by symbolic names p_0 and p_1, respectively. As depicted in Figure 5(a), the final set of *abstract summary edges* use the symbolic names p_0 and p_1 instead of their counterparts in the concrete summary.

Challenge 2.1: Abstracting Concrete Array Accesses. Precise handling of array accesses can be critically important in the dependence analysis of method invocations within software systems. When an abstract summary involving an array access is applied at a method invocation, we wish to understand precisely which array element is used or defined inside the method execution. Without such information, spurious dependence relationships may be generated—any data retrieved from an array would depend on any data added into the array. Precise handling of array accesses is challenging because the index used to access the array is not projected as an output of the method, and thus no summary will contain its dependence information.

To address this problem, we create a special symbolic name for each array index, as shown in Figure 5(b). If the accessed array is an input or output of the method, the index used to access the array is considered a (special) output (as shown in Figure 5(b)), and thus the transitive dependence relationships leading to the computation of the index would be included in the summary. If the index is a constant value (i.e., its computation does not depend on any method input), this constant value is recorded in the summary. In our example, index t used in line 23 of Figure 3(a) is abstracted by a symbolic name, $p_0.arr.index$, which is dependent on the symbolic location $p_0.size$, as portrayed in Figure 5(b). When the abstract summary is applied, or reused, for a future invocation of the same method, we will be able to obtain the run-time value of $p_0.size$ (before an invocation to `void add(int i)`) and identify the array element that is accessed during the invocation.

Challenge 3: Accounting for Varying Method Behavior. A set of concrete summaries for a given method (e.g., `void add(int i)`) essentially represents the behavior of as many individual invocations of the method in question. As such, the abstraction of such a set of concrete summaries will result in an equal number of abstract dynamic dependence summaries—one abstract dynamic dependence summary for one specific invocation of the method in question. If all abstract dynamic dependence summaries for the given method model the same set of abstract inputs, abstract outputs, and the dynamic dependencies between those abstract inputs and outputs, then it is safe to say that the method exhibits similar external heap data effects, or what we call *method behavior*, for each of its different invocations. In other words, if all invocations of a given method result in the same abstract dynamic dependence summary, then the behavior of any invocation of the method can be represented with that single abstract dynamic dependence summary.

The challenge arises when different invocations of a given method exhibit different external heap data effects, thus resulting in different abstract summaries, such as differing sets of abstract inputs, abstract outputs, and dependence relations between such inputs and outputs. The challenge is to accurately model the behavior of any subsequent invocation of the given method by selecting a set of abstract inputs, abstract outputs, and dependencies between the inputs and outputs from a divergent set of abstract summaries that represent varying heap data effects of dynamically observed method invocations. In other words, the resultant dynamic dependence summary for a method should be able to model the correct set of heap data effects (as dynamic dependencies between inputs and outputs) of any invocation of the method, in general.

Dynamic dependence summaries should account for variations in method invocations if they are to be amenable for reuse, for any subsequent invocation of the given method. Variations in external heap data effects for different invocations are observed under two broad circumstances that we discuss in the following.

Challenge 3.1: Accounting for Polymorphic Methods. Different method invocations for a given method may accept objects of different types that share a common supertype as arguments. Although these objects share a common supertype, their fields can differ significantly, or they might result in the execution of entirely different method implementations, as in the event of polymorphism, resulting in different external heap data effects, and thus abstract summaries between different invocations.

We overcome this problem by additionally recording the dynamically observed type information of each input argument with the symbolic name representing the parameter. This enables the accurate modeling of the variances in method behavior due to differing input argument types. Before a summary edge is made concrete (during the dependence analysis), we first check whether the recorded type in the edge matches the type of its corresponding actual parameter in the current execution and only apply those summary edges that match.

Challenge 3.2: Accounting for Divergent Control Flow. A similar problem arises when different control flow paths are followed across different invocations of the same method, even with the same type(s) of input argument(s). It is conceivable that the resulting external heap data effects also vary from one invocation to another for such methods, thus resulting in different abstract summaries for the same method.

In this work, abstract summaries from different method invocations that share the common set of dynamically observed argument-types for a given method are *aggregated* into a single overarching summary that we refer to as an aggregated abstract dynamic dependence summary, or simply an aggregated summary. An aggregated summary is created by simply performing set-union operations on the sets of abstract inputs, abstract outputs, and the dependencies (between such inputs and outputs) from the different abstract summaries that are being aggregated. Such an aggregated summary serves as a generic model of the external heap data effects for any of the method's invocations that share the same dynamically observed argument types.

Challenge 4: Reusing Dynamic Dependence Summaries. An abstract dynamic dependence summary represents a symbolic model of the heap data effects that are a result of invoking a particular method in question. However, to be able to save subsequent costs of analyzing the effects of such methods dynamically, the abstract summary must be made concrete. In other words, the symbolic information in an abstract summary must be substituted with runtime or concrete information to enable the modeling of specific invocations of the given method, thus going beyond a generic model of the method's behavior. However, even such a concrete model of heap data effects of a specific method invocation does not represent the dependencies between those instructions that influence or depend on the method invocation. To be able to use dynamic dependence summaries (concrete or abstract) toward modeling dependencies between actual runtime instructions, it is important to transform the heap data effects, which are essentially dependencies between the inputs and outputs of a method invocation, into dependencies between runtime instructions that define and use a method invocation's inputs and outputs, respectively. We refer to such reuse of a method's dynamic dependence summary as *summary application*.

Summary application is carried out in two steps. First, symbolic names in a method summary's input and output sets are substituted or *concretized* with their respective concrete (heap or stack) locations at a method invocation. Such concrete information in the summary forms the actual (transitive or direct) dependence relationships between the method invocation's actual inputs and outputs. Second, the dependencies between the inputs and outputs of a method invocation are then used to derive runtime dependencies between instructions that define or use those inputs and outputs. If a specific output (location) of a method invocation is dependent on an input (location) to the method invocation, we then determine that any runtime instruction that uses the value from the output is dynamically dependent on any instruction that defines the value in input to the method invocation.

Example. To better illustrate the idea of deriving dependencies between runtime instructions from a method invocation's dependence summary, consider Figure 6. The figure depicts an execution trace for a sample program, both of which are presented earlier in Figure 3(a) and (b). Each line in the execution trace shows the execution of an individual program instruction that we call a *runtime instruction* or an *instruction execution event*.[3] The progress of the execution is denoted by a downward arrow on the right that depicts the execution time. In addition, Figure 6 also portrays

[3]Note: A single program instruction can be executed multiple times and can lead to multiple instruction execution events. For example, program instruction `05: if (j < num)` results in two events: 05^1 `if (j < num)` and 05^2 `if (j < num)`.

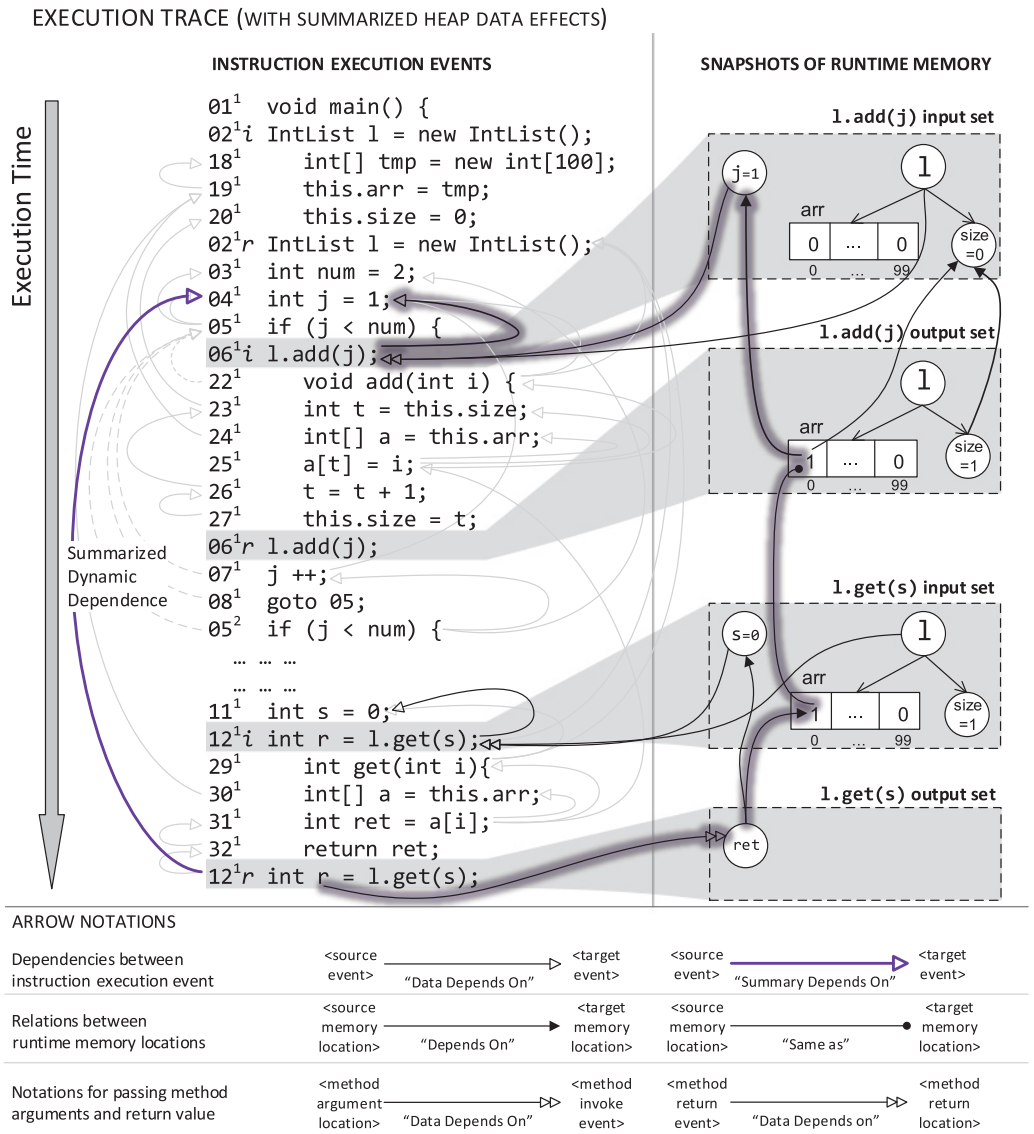


Fig. 6. Example program execution trace with summarized heap data effects (i.e., dynamic dependence summaries).

snapshots of the execution's memory on the right-hand side of the execution trace, taken at four different moments during the progress of the execution. These four snapshots of memory depict the inputs and outputs of the individual invocation of two methods: `l.add(j)` and `l.get(s)`. As such, these snapshots show the inputs to the invocation events (i.e., $06^1\,i$ `l.add(j);` and $12^1\,i$ `int r = l.get(s);`) and the outputs of the invocations just after their return events (i.e., $06^1\,r$ `l.add(j);` and $12^1\,r$ `int r = l.get(s);`). For instance, the snapshot showing the inputs to the invocation event $06^1\,i$ `l.add(j);` shows the input arguments `j` and (the receiver object) `l`. In addition, the memory locations accessible from `l` (i.e., the field `size`) and the field array `arr` (along with `arr`'s elements) are also depicted with a simple arrow going from `l` to its members (e.g., `l` → `size`). In

addition to showing the inputs and outputs to a method invocation, we also show how its outputs are dependent on the inputs with an arrow going from a location in the output set to a location in the input set. The arrow used to show such a dependency has a black-filled triangular arrowhead (e.g., `arr[0]` —▶ `j`). The execution trace and snapshots together also depict the method arguments' dependence on the invocation events and the dependence of the return event on a return value, if one exists. Such relations between the invocation and return events and the actual memory values are shown simply as data dependencies.

In Figure 6, consider the return event $\boxed{12^1\ r\ \texttt{int r = l.get(s);}}$ that is dependent on, or *uses*, the return value `ret`. The value `ret`, in turn, is part of the output set to the method invocation event $\boxed{12^1\ i}$. When we follow the chain of summarized dependencies between inputs and outputs, starting from `ret` that are highlighted in Figure 6, we obtain the following sequence of memory locations that ends with the input `j` within the input set for the method invocation $\boxed{06^1\ i\ \texttt{l.add(j);}}$:

$$\texttt{ret} \longrightarrow \texttt{l}\rightarrow\texttt{arr[0]} \longrightarrow \texttt{j}.$$

The input value `j` is in turn dependent on the invocation event $\boxed{06^1\ i\ \texttt{l.add(j);}}$, as depicted in the figure. Upon tracing the dependency for the event $\boxed{06^1\ i}$, with respect to the value `j`, we arrive at the runtime instruction $\boxed{04^1\ i\ \texttt{int j = 1;}}$ that actually defines the value `j`. In other words, the runtime instruction $\boxed{04^1\ i\ \texttt{int j = 1;}}$ defined the value of `j`, which was used to define the value at `l`→`arr[0]`, that was finally used by the runtime instruction $\boxed{12^1\ r}$ as a return value. Such a long chain of dependencies between the memory locations can be simply translated to a summarized dependency that the runtime instruction $\boxed{12^1\ r}$ has on the runtime instruction $\boxed{04^1}$. Such a *summarized dynamic dependence* is depicted on the left-hand side of the execution trace with an arrow going from $\boxed{12^1\ r\ \texttt{int r = l.get(s);}}$ to $\boxed{04^1\ \texttt{int j = 1;}}$.

4. DYNAMIC DEPENDENCE SUMMARIES

This section formally defines concrete and abstract dependence summaries, the aggregation of abstract summaries, and finally the concepts behind the application (or reuse) of dynamic dependence summaries. These concepts as a whole present our core technique that computes and uses summaries to improve the efficiency of dynamic dependence analysis. Our discussion in this section is in keeping with the conventions of object-oriented programming, as available in the Java programming language.

Method Invocations: Inputs and Outputs. The definitions of (concrete and abstract) dependence summaries, their aggregation and reuse, are rooted in the notions of inputs and outputs to method invocations and the notion of dynamic dependence. As such, we first define what we mean by inputs and outputs for a method invocation, followed by an overview of the definitions of dynamic dependence. We now introduce the concepts of inputs and outputs for a method under the framework of access paths and object graphs, which we define as follows.

Definition 1 (*Access Path (AP)*). For a member (f) (*i.e.,* field or array element), an access path is a sequence of memory locations leading to the access of the member (f), starting from a reference-typed object (o), with each preceding location in the sequence being the owner of the succeeding location and each succeeding location being the member of the preceding location.

Notationally, an access path from an object o to field f with many intermediate locations $f_1, f_2 \ldots f_n$ is denoted as $[o \rightarrow f_1 \rightarrow f_2 \rightarrow \ldots \rightarrow f_n \rightarrow f]$, or simply as $[\![o \dashrightarrow f]\!]$. In addition, an access path to an array element within an array f at index l is denoted as $[\![o \dashrightarrow f[l]]\!]$.

When stated differently, access-paths chain together a set of memory locations into a series of *owner-member* relations (between the set of memory locations), in turn informing how a certain field or array-element is accessed starting from a reference-typed object. This allows us to introduce the notion of an object graph that can be thought of as a set of access-paths that all start from a common reference-typed object.

Definition 2 (*Object Graph (OG)*). An object graph (g) is a graph rooted at a reference-typed object (o) that contains a set of memory locations as nodes that are accessible from the reference-typed object (o) via edges that represent *owner-member* relationships and connect a owner memory location (reference-typed object) and a member memory location (field or array element).

Note that the only way to access a memory location in an object graph (g) is to perform a sequence of member (field or array element) dereferences on the root object that, in turn, is represented as the access path of a member location in an object graph. As such, each member location can be expressed as an access path within an object graph. In Figure 4, we show two example object graphs reachable from parameter o^{02} for an invocation of the method `void add(int i)` before and after the execution of line 06. As an extension and generalization of the example in Figure 4, we now define the concept of a set of object graphs that are accessible from the arguments of a method invocation.

In this work, a method invocation refers to a specific instance in a series of runtime invocations of a single method within a program execution and is denoted as $c : m(a_0, a_1, \ldots, a_n)$, where

- c uniquely identifies a method invocation event,
- m is the method being invoked, and
- $a_0, a_1, \ldots, a_n$ are actual arguments to the method.

Definition 3 (*Accessible Object Graph Set (AOGS)*). For a method invocation $c : m(a_0, a_1, \ldots, a_n)$, the accessible object graph set (AOGS) is a set of object graphs that are rooted at any reference-typed objects pointed to by the arguments $(a_0, a_1, \ldots, a_n)$ and represented as $\mathcal{G}_c$.

Note: $\mathcal{G}_c^{pre}$ and $\mathcal{G}_c^{post}$ denote the AOGS immediately before and after the method invocation $c : m(a_0, a_1, \ldots, a_n)$, respectively.

Such a conceptualization of how memory locations are related and accessed during method invocations enables us to account for the situation discussed in Section 3 (Challenge 1), where a method invocation potentially uses not only the arguments to the invocation but also any memory location reachable from those arguments. Modeling accessible memory locations enables the modeling of how fields of an object that is passed as an argument are used or defined. Such a modeling of arguments, and their accessible fields, is essential in modeling standard data structures such as linked lists, trees, and graphs. Using such concepts of access paths, object graphs, and object graph sets, we now define the inputs and outputs to a method invocation.

Definition 4 (*Input and Output Sets*). For a method invocation $c : m(a_0, a_1, \ldots, a_n)$, the input set $\mathcal{I}_c$ and output set $\mathcal{O}_c$ are defined as follows:

(1) **Input set** ($\mathcal{I}_c$) is a set of memory locations that exist before the method invocation and can be read during the method invocation such that $\boxed{\mathcal{I}_c \subseteq \mathcal{G}_c^{pre} \cup \mathcal{P}_c}$, where
   - $\mathcal{G}_c^{pre}$ is the AOGS immediately before the execution of m at c, and
   - $\mathcal{P}_c$ is the set of memory locations that represent actual parameters passed into method invocation c.

(2) **Output set** ($\mathcal{O}_c$) is a set of memory locations that can be written to during the method invocation and are accessible after the completion of the method invocation such that $\boxed{\mathcal{O}_c \subseteq \mathcal{G}_c^{post} \cup \mathcal{L}_c \cup \{ret\} \cup \{g_{ret}\}}$, where
   - $\mathcal{G}_c^{post}$ is the AOGS immediately after the execution of m at c;
   - *ret* is the memory location containing the value to be returned;
   - g_{ret} is the object graph defined by *ret* (if *ret* is of reference type); and
   - $\mathcal{L}_c$ is a set of integer indices used in the accesses of the arrays referenced by memory locations in $\mathcal{G}_c^{pre}$, $\mathcal{G}_c^{post}$ or g_r.

Locations that are unreachable from a method argument via such access paths do not escape the method invocation and have no impact beyond the scope of the method invocation. Hence, such locations are discounted from a method invocation's input and output sets and eventually from the method's dependence summary. As such, the input and output sets can often be a subset of the accessible object graph sets for a method invocation, as portrayed in Definition 4. Additionally, the indices used to access the arrays in the object graphs are included in the output set. Tracking these indices is necessary for the precise handling of array accesses, as described in Section 3. Note that our approach treats static fields, which are globally accessible and used in a method invocation, as additional arguments to a method. Similarly, since Java is a call-by-value language, where the values of the actual parameters cannot be changed by the method, $\mathcal{P}_c$—locations serving as arguments to a method invocation—is not part of the output set.

Dynamic Dependencies. After defining the input and output sets for a method invocation, the next step toward defining a concrete dependence summary is to define dynamic dependencies between the inputs and outputs of a method invocation. We now present the definitions of dynamic data and control dependence between memory locations in a program execution.

In this work, we refer to the execution of a static program instruction, during the program's execution, as a runtime instruction, and it is denoted using the form a^j (i.e., the j^{th} execution of the static program instruction a).

Definition 5 (*Dynamic Dependence*). Given two memory locations, l_1 and l_2, memory location l_2 is said to be directly and dynamically dependent on memory location l_1 under the following two conditions:

(1) **Dynamic data dependence.** A runtime instruction (a^j) writes a value to the memory location l_2 that is computed using the value read from memory location l_1.
(2) **Dynamic control dependence.** A runtime instruction a^j that writes a value to the memory location l_2 and the occurrence of a^j was predicated on the value at memory location l_1.

Using Definition 5, which defines relations of direct dependence between two memory locations, transitive dependence between two memory locations (l_1 and l_2) can be easily established by following a sequence of direct dependencies from one location (l_2) to the other (l_1). Based on this definition of dynamic dependence, we can now establish relations of direct or transitive dependencies between memory locations in the input and output sets of a method invocation. Such a set of dynamic dependencies between the input and output locations of a method invocation is essentially the concrete summary of the method invocation.

Concrete Summaries. Informally, a concrete dynamic dependence summary, or simply a concrete summary, is a set of (transitive or direct) dynamically observed data or control dependencies between the inputs and outputs of a specific invocation of a given method. Using the definitions for inputs and outputs for a method invocation and dynamic data and control dependencies between memory locations, we formally define concrete summaries as follows.

Definition 6 (*Concrete Dynamic Dependence Summary*). For a method invocation event of the form $c : m(a_0, a_1, \ldots, a_n)$, the concrete dynamic dependence summary $\mathcal{S}_c$ is a Cartesian set $\mathcal{I}_c \times \mathcal{O}_c$, where each element in the set is a transitive (or direct) dynamic dependence of the locations in the output set ($\mathcal{O}_c$) upon the locations in the input set ($\mathcal{I}_c$).

Notationally, an access path–based concrete dynamic dependence summary is expressed in the form $\boxed{\bigcup\{ [\![o_i \dashrightarrow f]\!] \dashleftarrow [\![o_j \dashrightarrow g]\!] \}}$, where

- $[\![o_i \dashrightarrow f]\!] \in \mathcal{I}_c$;
- $[\![o_j \dashrightarrow g]\!] \in \mathcal{O}_c$;
- o_i and o_j are the objects pointed to by parameters a_i and a_j; and
- $[\![o_i \dashrightarrow f]\!]$ and $[\![o_j \dashrightarrow g]\!]$ are access paths for heap locations f and g, respectively.

Abstract Summaries. As described in Section 3 (Challenge 2), concrete summaries contain invocation-specific information and cannot be reused. Abstraction needs to be performed to replace concrete information with suitable abstract information so that the abstracted summaries are applicable to all other executions. The abstraction process has two steps. In the first step, we express each node in an object graph that is part the concrete summary with the corresponding root object and the access path through which the node can be reached. The result of this step is a set of access path–based concrete summary edges, as shown in the bottom part of Figure 4. In the second step, we replace each concrete parameter object or variable with a symbolic name, resulting in the final abstract summary that can be applied in other executions of the method (as shown in the bottom parts of Figure 5). Note that in our approach, these two steps are combined in one single summary generation phase. They are discussed separately in the article for clarity of presentation.

Definition 7 (*Abstract Dynamic Dependence Summary*). Given a concrete dynamic dependence summary of the form $\boxed{\bigcup\{[\![o_i \dashrightarrow f]\!] \dashleftarrow [\![o_j \dashrightarrow g]\!]\}}$, for a method invocation of the form $c : m(a_0, a_1, \ldots, a_n)$, the abstract dynamic dependence summary is the symbolic representation of the method invocation–specific runtime information in the concrete summary, and the resulting access path–based abstract dynamic dependence summary is of the form $\boxed{\bigcup\{[\![p_i \dashrightarrow f]\!] \dashleftarrow [\![p_j \dashrightarrow g]\!]\}}$, where p_i and p_j are the symbolic names for the i^{th} and j^{th} memory locations o_i and o_j that are pointed to by parameters a_i and a_j, respectively.

Note: For concrete summary edges that model array element accesses as follows, $[\![o_i \dashrightarrow f[l]]\!] \dashleftarrow [\![o_j \dashrightarrow g]\!]$
$[\![o_k \dashrightarrow h]\!] \dashleftarrow l$, where l is an actual index value within an array, and the corresponding abstract summary edges are of the form
$[\![p_i \dashrightarrow f[f.index]]\!] \dashleftarrow [\![p_j \dashrightarrow g]\!]$
$[\![p_k \dashrightarrow h]\!] \dashleftarrow f.index$, where $f.index$ is the symbolic name for the actual index value l and f is the array.

Aggregate Dependence Summary. Eventually, abstract summaries computed for all invocation events that invoke method m are combined, or aggregated, and used as m's summary for the future dependence analysis. This is done in the event where different method invocations, for the given method, result in different or varying abstract summaries as discussed in Section 3 (Challenge 3). The aggregation of different abstract summaries performs two essential functions, as showcased in Algorithm 1. First, the aggregation step creates and maintains separate aggregate summaries for method invocations with different dynamically observed input argument types. Lines 6 through 14 in Algorithm 1 are used to create a signature for the method invocation corresponding to a given abstract summary. The signature of a method invocation is simply the string concatenation of the name of the invoked method and the types of runtime arguments to the method invocation. This enables proper modeling of method behavior of polymorphic methods and their invocations as discussed in Section 3 (Challenge 3.1).

ALGORITHM 1: Aggregation of Abstract Summaries for Method Invocations

Require:
1: Map *aggregate_summaries* $\leftarrow \emptyset$ // aggregate abstract summaries mapped by method invocation argument types
2: Set *abstract_summaries* // a set of abstract summaries as input
3: **for** each abstract summary $\bigcup\{[\![p_i \dashrightarrow f]\!] \dashleftarrow [\![p_j \dashrightarrow g]\!]\} \in$ *abstract_summaries* **do**
4: Object *summary* $\leftarrow \bigcup\{[\![p_i \dashrightarrow f]\!] \dashleftarrow [\![p_j \dashrightarrow g]\!]\}$
5:
6: // a) get method invocation information associated with the abstract summary.
7: String *inovked_method_name* $\leftarrow$ getMethodName(*summary*)
8: List *argument_types* $\leftarrow$ getMethodInvokeArgumentTypes(*summary*)
9:
10: // b) begin extraction of dynamically observed method-invocation signature.
11: String *signature* $\leftarrow$ *inovked_method_name* // signature starts with invoked method's name
12: **for** each argument type *type* $\in$ *argument_types* **do**
13: *signature* $\leftarrow$ *append*(*signature*, *type*)
14: **end for**
15:
16: // c) begin aggregation of abstract summary.
17: Object *aggr_sumr* $\leftarrow$ *aggregate_summaries*[*signature*] // aggregate summary for method invocation signature
18: **for** each access path–based dependency $\{[\![p_i \dashrightarrow f]\!] \dashleftarrow [\![p_j \dashrightarrow g]\!]\} \in$ *summary* **do**
19: *aggr_sumr.inputs* $\leftarrow$ *aggr_sumr.inputs* $\bigcup \{[\![p_i \dashrightarrow f]\!]\}$ // union of abstract inputs.
20: *aggr_sumr.outputs* $\leftarrow$ *aggr_sumr.outputs* $\bigcup \{[\![p_j \dashrightarrow g]\!]\}$ // union of abstract outputs.
21: *aggr_sumr.dependencies* $\leftarrow$ *aggr_sumr.dependencies* $\bigcup \{[\![p_i \dashrightarrow f]\!] \dashleftarrow [\![p_j \dashrightarrow g]\!]\}$ // union of dependencies.
22: **end for**
23: *aggregate_summaries*[*signature*] $\leftarrow$ *aggr_sumr* // update aggregate summary for method invocation signature
24: **end for**
25: return *aggregate_summaries*

Second, for method invocations of a given method that share the same dynamically observed input arguments types, an *aggregate* summary is created that results in the generic modeling of the method's varying control flow as discussed in Section 3 (Challenge 3.2). Lines 16 through 23 in Algorithm 1 handle the aggregation of different abstract summaries that share the same input parameter types. Once the signature of the method invocation is computed, it is used retrieve an existing aggregate summary (line 17 in Algorithm 1). This is followed by lines 18 through 21 in Algorithm 1, which perform a union of all inputs, outputs, and dependence edges of the abstract summary with the existing aggregate summary, which is finally stored in line 23. It is worth noting that a node (memory location) within an object graph may have multiple access paths from the root object. For each such access path used in the method invocation, we will generate a corresponding summary edge. Note that this treatment can potentially introduce both unsoundness and imprecision and is further discussed in Section 9.

Summary Application. Aggregated abstract dynamic dependence summaries are finally used to model the behavior of any subsequent invocations of their respective methods, thus enabling the reuse of recorded method behavior. We refer to such a process of reusing method behavior as the application of a dynamic dependence summary for a method invocation, or simply summary application. Summary application for a given method invocation is composed of two steps. First, we concretize the symbolic information in the aggregated abstract summary. The dependencies within abstract dependence summaries are made concrete by substituting the abstract symbolic information that represent the method's arguments, in both the input and output sets, with concrete runtime objects. The concrete runtime objects for the substitutions in the input and output sets are collected just before and just after a method invocation, respectively. This allows us to re-create a specific concrete model of the heap data effects between the inputs and outputs of a specific method invocation. To illustrate, consider the example of applying summaries at the invocation of the method `add` (line 06 in Figure 3(a)) in a future execution of the program. The following three abstract summary edges are concretized into their respective concrete summary edges before being recorded into the trace:

(1) Substituting p_0 with o^{02} and p_1 with i;
(2) Abstract Dependencies

Abstract Dependencies	Concrete Dependencies
$\bullet\ [\![p_0 \dashrightarrow size]\!] \leftarrow [\![p_0 \dashrightarrow size]\!]$	$\bullet\ [\![o^{02} \dashrightarrow size]\!] \leftarrow [\![o^{02} \dashrightarrow size]\!]$
$\bullet\ p_1 \leftarrow [\![p_0 \dashrightarrow arr[arr.index]]\!]$	$\bullet\ i \leftarrow [\![o^{02} \dashrightarrow arr[arr.index]]\!]$
$\bullet\ [\![p_0 \dashrightarrow size]\!] \leftarrow arr.index$	$\bullet\ [\![o^{02} \dashrightarrow size]\!] \leftarrow arr.index.$

Second, we translate the dynamic dependencies between the concrete summary's inputs and outputs for a given method invocation to summarized dynamic dependencies between runtime instructions. It is important to remember that the reuse of dynamic dependence summaries is done with goal of efficient computation and modeling of dynamic dependencies between actual runtime instructions. As such, we extend the dynamic dependencies between memory locations in the input and output sets of a method invocation to the resulting dynamic dependencies between the actual runtime instructions that define and use the memory locations in the input and output sets, respectively. Such dependencies between runtime instructions derived from dynamic dependence summaries are called *summarized dynamic dependencies* and are defined as follows.

Definition 8 (*Summarized Dynamic Dependence*). Given two runtime instruction a^j and b^k and a method invocation of the form $c : m(a_0, a_1, \ldots, a_n)$, there exists a

summarized dynamic dependence on a^j from b^k (represented as $a^j \Leftarrow b^k$) under the following conditions:

- a^j writes a memory location l_1 that belongs to the input set of a method invocation, and b^k reads a memory location l_2 that belongs to the output set of the same method invocation; and
- there exists a relationship $[\![p_i \dashrightarrow f]\!] \Leftarrow [\![p_j \dashrightarrow g]\!]$ in the abstract summary of the invoked method such that
  —the location $[\![o_i \dashrightarrow f]\!]$ (before the call) is the same location as l_1,
  —and the location $[\![o_j \dashrightarrow g]\!]$ (after the call) is the same location as l_2,
  where o_i and o_j are the concrete runtime objects for the symbolic values p_i and p_j, respectively.

Note: There also exists a summary dependence edge of the form $a_j \Leftarrow b_k$ if two pairs of relationships in the abstract summary that model an array element access

- $[\![p_j \dashrightarrow g]\!] \Leftarrow [\![p_i \dashrightarrow f[f.index]]\!]$; $[\![p_k \dashrightarrow h]\!] \Leftarrow f.index$, and
- $[\![p_r \dashrightarrow v[v.index]]\!] \Leftarrow [\![p_m \dashrightarrow q]\!]$; $[\![p_t \dashrightarrow u]\!] \Leftarrow v.index$ such that

- $[\![o_m \dashrightarrow q]\!]$ is the same location as l_2,
- $[\![o_j \dashrightarrow g]\!]$ is the same location as l_1,
- $[\![o_i \dashrightarrow f]\!]$ and $[\![o_r \dashrightarrow v]\!]$ refer to the same array object, and
- values in $[\![o_k \dashrightarrow h]\!]$ and $[\![o_t \dashrightarrow u]\!]$, which represent indicies, are equal.

As discussed in Section 2, two (abstract) array slots of the form $[\![p_1 \dashrightarrow f[f.index]]\!]$ and $v.index$ are considered to be the same location in an execution if (1) the two concrete array objects in locations $[\![o_i \dashrightarrow f]\!]$ and $[\![o_r \dashrightarrow v]\!]$ are the same, and (2) the values of the inputs on which the two indices depend (i.e., $[\![o_k \dashrightarrow h]\!]$ and $[\![o_t \dashrightarrow u]\!]$ in the definition) are the same. A transitive edge is not added if one or both of the indices depend on multiple inputs, as it is unclear how the indices are computed from these inputs and how to compare their values. Note that this treatment can potentially introduce both unsoundness and imprecision and is further discussed in Section 9.

5. USING DYNAMIC DEPENDENCE SUMMARIES FOR DYNAMIC DEPENDENCE ANALYSIS

As a part of our approach, we compute dynamic dependence information and dynamic dependence summaries by analyzing program executions. Dynamic information for program executions are generated and recorded in the form of execution traces during a summary generation phase using a representative test executions. The dynamic dependencies and consequently the dynamic dependence summaries are computed from the execution traces and stored to a disk file for use in a future dynamic dependence analysis. The computation and use of the dynamic dependence summaries are performed in the following phases.

Phase I. Summary Generation Using Representative Executions. We produce the abstract summaries for the methods being summarized (these can be user specified in a configuration file by method, class, or package) by analyzing the execution traces. For each method m, we find all instances of m's execution, and for each instance in the trace, we use a worklist-based algorithm to compute an abstract summary according to the approach and definitions in Section 4. The result of the summary generation is a mapping of inputs (formal method parameters or accessible fields) to outputs (formal method parameters or accessible fields) that they influenced, expressed as abstract summary edges, within aggregated dynamic dependence summaries.

Phase II. Summary Application in Dependence Analysis. To apply the summaries to dynamic dependence analyses, the developer would choose to instrument her test case (i.e., program execution) supported by the dynamic dependence summaries generated in the first phase. In the program's execution, the instrumenter would inspect each method invocation to determine the existence of a corresponding dynamic dependence summary. If the summary is not provided for a method invocation, the program execution and its instrumentation proceed with exhaustive dependence profiling. However, if a summary does exist for the method: (1) the abstract summary edges are obtained, (2) the runtime concrete inputs and outputs are matched with the symbolic names in the corresponding summary, and (3) the concretized summary edges are recorded to the trace.

Phase III. Dependence Graph Computation and Use. A summary-based dependence analysis profiles the execution of all methods except those that have abstract dependence summaries. These dependence summaries are then used to carry out dependence analysis by building a summary-based dynamic dependence graph. Before presenting the summary-based dynamic dependence graph, we first define the regular dynamic dependence graph.

Definition 9 (*Dynamic Dependence Graph*). A dynamic dependence graph $(\mathcal{V}, \mathcal{E})$ has node set $\mathcal{V} \subseteq \mathcal{D} \times \mathcal{N}$, where each node is a static statement $(\in \mathcal{D})$ annotated with an integer i $(\in \mathcal{N})$ representing the i^{th} execution of this statement. An edge $e \in \mathcal{E}$ of the form $a^j \twoheadleftarrow b^k$ $(a, b \in \mathcal{D};\ j, k \in \mathcal{N})$ denotes that the j^{th} execution of statement a writes a (heap or stack) location that is then used by the k^{th} execution of b without an intervening write to that location.

A dynamic dependence graph essentially represents, or models, a program execution and is shown for an example program execution in Figure 3(b). Based on the definitions of summary edges and dynamic dependence graph, we give the definition of the summary-based dependence graph.

Definition 10 (*Summary-Based Dynamic Dependence Graph*). A summary-based dynamic data dependence graph $(\mathcal{V}, \mathcal{E} \cup \mathcal{T})$ is a regular dynamic data dependence graph augmented with an additional set of dependence edges $\mathcal{T}$ that denote summary dependence edges (refer to Definition 8) between any two nodes a^j and b^k $(a^j, b^k \in \mathcal{V})$.

A dynamic dependence graph is computed by recovering dependence relationships from the trace, as described in Definition 9. When (concretized) summary edges are encountered, the location matching approach described in Definition 8 is used to recover the missing relationships to build a summarized dynamic dependence graph. Using this summarized dynamic dependence graph, developers and automated techniques can perform dynamic analyses, such as interrogative software debugging, bloat and change impact analysis, and dynamic slicing, as discussed in Sections 1 and 2.

Program Instrumentation. The implementation of our summary-based dependence analysis is based on the instrumentation and analysis of executable Java class files. The goal of the instrumentation is to enable the generation of a detailed trace that records the execution of each instruction in the program and the heap/stack location it accesses. We assign a unique ID to each runtime object that is used to identify the object and its fields in the execution trace. Program instrumentation involves the addition of probe instructions within the executable code for instructions that require runtime monitoring, thus enabling the requisite analysis of the executing instructions. With the execution trace, we construct the dynamic data dependence graph, which then enables client dynamic analysis techniques such as dynamic slicing. We perform

load-time bytecode instrumentation for classes that do not belong to the Java standard library using ASM [Bruneton et al. 2002], a Java bytecode manipulation framework. In contrast, we instrument the classes in the standard Java library prior to execution—several of these classes cannot be instrumented during load time due to the technical requirements of the JVM to load them prior to the instrumenter.

6. SUMMARY ABSTRACTION ALGORITHM

The algorithm for summary abstraction starts with computing a regular dependence graph (line 3) and the transitive dependence relationships for each node on it (line 4). Initially, the input set *is* contains all incoming objects (line 5), and the output set *os* contains only the returned objects (line 6). The set *sn* contains the symbolic name for each incoming and outgoing object. The worklist-based trace processing (lines 9 through 44) iteratively identifies and adds heap and stack locations into the input and output sets and computes symbolic names for them (stored in set *sn*). After this processing is done, each transitive dependence edge between statement executions $s_1 \dashrightarrow s_2$ is retrieved from map *td* (lines 45 through 52). If s_1 reads a variable/object in the input set and s_2 writes a variable/object in the output set (line 46), we find each symbolic name p_a for a and each symbolic name p_b for b (line 48) and add an abstract summary edge $p_a \dashrightarrow p_b$ into the abstract edge set *as*. Eventually, sets *as* for all *m*'s executions (in the training phase) are combined and used as *m*'s abstract summary. Algorithm 2 shows our handling only for the two most complicated cases (i.e., array reads and array writes); the handling for all other cases is simpler and can be easily derived from the two cases shown.

7. EVALUATION

The use of dynamic dependence summaries on dynamic analysis stands to result in performance gains and accuracies losses in the underlying dynamic analysis. To study such a trade-off between the accuracy and performance of dynamic analyses when using dynamic dependence summaries, we ask the following research questions.

RQ1. What is the extent to which summaries potentially assist in cost saving?
RQ2. What are the performance cost savings with the use of dynamic dependence summaries for dynamic analyses?
RQ3. How does the use of dynamic data and aggregation of concrete dynamic dependence summaries affect the accuracy of dynamic analysis?
RQ4. How does the use of dynamic dependence summaries affect the efficiency and effectiveness of a runtime client analysis?

The first research question, *RQ1*, is designed to assess the extent of cost savings as a result of creating and using dynamic dependence summaries toward dynamic analyses. *RQ2* is designed to evaluate the actual performance cost savings, in space and time, when analyzing dynamic dependencies within program executions by using dynamic dependence summaries for designated components that are ancillary to the development of the original program. *RQ3* helps to investigate the accuracy of abstracted and aggregated dynamic dependence summaries, independent of a downstream client analysis. Finally, *RQ4* is designed to understand the cumulative impact of the "efficiency versus accuracy" trade-off when using dynamic dependence summaries for an actual runtime client analysis.

ALGORITHM 2: Generate an Abstract Summary From an Instruction-Level Execution Trace for a Method

Require: An execution trace t_m for method m
Objects $o_1, o_2, \ldots, o_n$ passed into m from the caller, and o_r returned from m
Set $as \leftarrow \emptyset$ // a set of abstract summary edges
Edge Set $td : \langle s_i \dashrightarrow s_j \rangle$ //transitive dependence relationships
Dependence graph g = *computeRegularDependenceGraph*(t_m)
td = *computeTransitiveClosureForEachNode*(g)
Set $is \leftarrow \{o_1, o_2, \ldots, o_n\}$ //input set
Set $os \leftarrow \{o_r\}$ //output set
Map $sn \leftarrow \{(o_1, \{p_0\}), (o_1, \{p_1\}), \ldots, (o_n, \{p_n\}), (o_r, \{p_r\})\}$ //initial symbolic names
List $wl \leftarrow \{o_1, o_2, \ldots, o_n, o_r\}$ //initial worklist
while $wl \neq \emptyset$ **do**
  Object $o \leftarrow wl.pop()$
  for each statement execution s in t_m **do**
    switch (s)
    case "a = b[i]"**:**
      if $o_b = o$ **then**
        for each symbolic name p in $sn(o)$ **do**
          String $p_i \leftarrow append(p, ".index")$
          String $p_a \leftarrow append(p, ".[", p_i, "]")$
          if $p_a \notin sn(o_a)$ **then**
            $sn(o_a) \leftarrow sn(o_a) \cup \{p_a\}$
            $sn(i) \leftarrow sn(i) \cup \{p_i\}$
            $wl \leftarrow wl \cup \{o_a\}$
          end if
          if $o \in is$ **then**
            $is \leftarrow is \cup \{o_a\}$
          end if
          $os \leftarrow os \cup \{o_a\} \cup \{i\}$
        end for
      end if
    case "a[i] = b"**:**
      if $o_a = o$ **then**
        for each symbolic name p in $sn(o)$ **do**
          String $p_i \leftarrow append(p, ".index")$
          String $p_b \leftarrow append(p, ".[", p_i, "]")$
          if $p_b \notin sn(o_b)$ **then**
            $sn(o_b) \leftarrow sn(o_b) \cup \{p_b\}$
            $sn(i) \leftarrow sn(i) \cup \{p_i\}$
            $wl \leftarrow wl \cup \{o_b\}$
          end if
          $os \leftarrow os \cup \{o_b\} \cup \{i\}$
        end for
      end if
    end switch
  end for
end while
for each transitive dependence edge $s_1 \dashrightarrow s_2 \in td$ **do**
  if s_1 reads from a location a AND s_2 writes into a location b AND $o_a \in is$ AND $o_b \in os$ **then**
    // a and b can be both variables and field locations
    for each $p_a \in sn(a)$, each $p_b \in sn(b)$ **do**
      $as \leftarrow as \cup \{p_a \dashrightarrow p_b\}$
    end for
  end if
end for
return as

Experimental Subjects. We implemented our technique to perform three experiments and carry out a case study to answer the research questions. The evaluation was performed on a quad-core Intel i7 3.07GHz machine running a 64-bit version of OpenJDK 7 JVM with a maximal heap size of 2GB.

This evaluation employs eight client programs: NanoXML (>2.6 KSLOCs), PL-241 (>6.9 KSLOCS), Jtopas (>10 KSLOCs), Antlr (>35 KSLOC), Bloat (>41 KSLOC), PMD (>60 KSLOC), FOP (>102 KSLOC), and Jython (>245 KSLOC). As part of this empirical investigation, we summarized all methods in the Java standard library (i.e., `rt.jar`) that were executed by the different runs of the eight client programs. Further, for each subject program, we considered the client code to be parts of the application that were not contained within the Java standard library (`rt.jar`).

Multiple executions were monitored for NanoXML (20 executions), PL-241 (13 executions), and Jtopas (12 executions), each with a different test input. Antlr, Bloat, PMD, FOP, and Jython were executed and monitored using a single test input for each program from the DaCapo benchmarks' "small" configurations. We monitored, generated, and stored (to disk) the execution traces for the resulting 40 executions across the eight client programs. As such, fewer test executions were used for the client programs, as the scale of their executions increased in terms of trace size (on disk) and execution times.

Antlr, Bloat, PMD, FOP, and Jython, with their test inputs, were obtained from the DaCapo benchmarking suite [Blackburn et al. 2006]. NanoXML and Jtopas and its test cases are obtained by the Subject-artifact Infrastructure Repository [Do et al. 2005]. PL-241 is an SSA-based optimizing compiler, and its development and test inputs are a product of a graduate-level course on advanced compiler construction at the University of California, Irvine.

The client programs were chosen such that they were real-world subjects carrying out nontrivial computations with system-level tests. It was important to select subjects with system-wide tests to enable the execution of significant portions of the subjects, thus resulting in the execution of a wide range of methods from the Java standard library.

Independent Variable. We used the following independent variable for all experiments: the dynamic dependence analysis used to detect data and control flows. We performed our dynamic dependence analysis in two ways:

Treatment 1: Exhaustive Analysis. All components that are executed, whether core or external, are instrumented and analyzed.

Treatment 2: Summary-Based Analysis. Only the program under test is instrumented and analyzed, and summaries from all external components are reused.

Answering RQ1: Potential Cost Savings Experiment. In this experiment, we assess the potential cost savings that can be gained due to summarization during dynamic monitoring and analysis of software executions. We present statistics of the recorded executions of instructions, in whole program execution traces, as a distribution between client code and third-party library code. Through the means of such a distribution, the goal is to demonstrate the extent of library code execution within the execution of a software system and thus the potential efficiency gains during summary-based runtime monitoring of program executions.

Cost savings experiment dependent variables. We use the following two metrics to assess the distribution of executions between client and library code:

Metric 1: Percentage of Client-Code Instructions (C). Number of recorded instances of all client code instruction as a percentage of net recorded instances of instructions in a whole program execution trace. This is computed as

$$\mathbf{C} = \frac{(\textit{total instances of } \textbf{\textit{client instructions}})}{(\textit{total instances of } \textbf{\textit{all instructions}})} \times 100.$$

Table I. Distribution of Whole Execution Traces Across Client and Library Instructions

Subject	**Whole Execution Trace Size** (instructions. $\times 10^3$)	Percentage of **Client Instructions (C)**	Percentage of **Library Instructions (L)**
NanoXML	319.8	18.69	81.31
PL-241	1,110.7	26.43	73.57
Jtopas	1,467.5	69.56	30.40
PMD	18,860.0	50.19	49.81
FOP	110,981.3	48.81	51.19
Antlr	221,849.8	98.03	1.97
Bloat	391,041.4	38.94	61.06
Jython	1,734,670.3	4.40	95.60

Metric 2: Percentage of Library-Code Instructions (L). Number of recorded instances of all library code instructions as a percentage of net recorded instances of all instructions in a whole program execution trace. This is formulated as

$$\mathbf{L} = \frac{(\textit{total instances of } \textbf{library instructions})}{(\textit{total instances of } \textbf{\textit{all instructions}})} \times 100.$$

Whole execution traces were produced for all eight client programs (NanoXML, PL-241, Jtopas, Antlr, Bloat, PMD, FOP, and Jython) with the exhaustive instrumentation and analysis of all components. A single whole execution trace corresponds to a single run of any given subject program. Instances of all recorded instructions within an execution were designated to client code unless they belonged in the Java standard library (*rt.jar*), where they were deemed as library code. The results of this experiment are summarized in Table I.

Table I reports, for each client program, the average number of recorded instruction instances in the whole execution traces obtained from each test execution of the client program. Thus, for NanoXML, 20 different execution traces were captured, which on an average contained more than 319,000 records of instruction executions. Table I also reports the distribution of the whole program execution traces as percentages of client and library code for the executions for a given subject. For instance, in an average execution, NanoXML's code accounted for a little more than 18% of all instruction executions, whereas 80% of the execution trace was due to the execution of instructions in external libraries.

Cost savings experiment results. The results in Table I show seven subjects, with the exception of Antlr rely heavily on the Java standard library. Only Jtopas, PMD, and Antlr, for the executions that we studied, rely more on the application code than on the code from the Java standard library. However, in the cases of PMD and Jtopas, the contribution of library instructions is still substantial—30.4% instructions on average in the executions of Jtopas and 49.81% of instructions in the case of PMD. Even in the case of Antlr, the 1.97% instructions from Antlr constitute the execution of a little more than 4.3 million instructions from the Java libraries. The two most library-reliant client programs are Jython and NanoXML, with nearly 96% and 81% of the executed instructions from the Java standard libraries, respectively.

Answering RQ2: Efficiency Experiment. In this experiment, we investigated the impact of reused dependence summaries on the costs involved in performing dynamic dependency analysis. As such, we employed the following two metrics (presented as this experiment's dependent variables) to assess the costs of using dynamic summaries.

Efficiency experiment dependent variables. For the two forms of dynamic analysis (i.e., exhaustive and summarized), we measured the following metrics:

Table II. Efficiency Experiment Results

		Exhaustive			Summary			
Subject	T_O (sec)	T_E (sec)	$(T_E - T_O)/T_O$	S_E (# instr. $\times 10^4$)	T_S (sec)	$(T_S - T_O)/T_O$	S_S (# instr. $\times 10^4$)	$\frac{(S_E - S_S)}{S_E}$
NanoXML	0.07	527.74	7,538.14	32	1.70	23.29	7	0.76
PL-241	0.16	2,167.82	13,547.88	111	5.20	31.50	43	0.61
Jtopas	0.12	2,533.02	21,107.50	147	8.44	69.33	113	0.23
Antlr	0.22	1,463.94	6,653.29	22,185	1,078.38	4,900.74	21,748	0.02
Bloat	0.99	2,258.63	2,282.75	39,104	762.31	769.79	15,227	0.61
PMD	0.25	131.86	537.22	1,886	50.58	205.47	947	0.50
FOP	0.50	703.04	1,416.42	11,098	271.39	546.15	5,417	0.51
Jython	0.29	17,531.19	60,451.36	173,467	668.58	2,304.45	7,632	0.96
Average			**14,191.82**			**1,106.34**		**0.53**

Note: T_O is the mean original running time of the program, and $T_{E|S}$ and $S_{E|S}$ are mean running times of whole execution analysis and mean trace size for each technique. $(T_{E|S} - T_O)/T_O$ shows the mean runtime overheads for each technique. $(S_E - S_S)/S_E$ shows the mean cost savings in trace sizes with summaries.

Metric 1: Execution Trace Size (S). Size of the execution trace required for each analysis.

Metric 2: Execution Time (T). Time to run the execution with each analysis.

We ran each subject program with their test inputs under exhaustive and summary-based treatments, thus resulting in a distinct whole program trace and a summarized program trace for each execution. We collected performance statistics for each execution. We measured the size of the resulting trace with a global size counter that kept track of the number of recorded runtime instructions. We also maintained a timer that kept track of the elapsed time during a program's execution. (Note: The performance metrics were collected in the same manner for both summary and exhaustive execution analyses.)

Table II summarizes the results of this experiment. For the test runs of a given client subject, Table II reports the mean execution running times and the mean execution trace sizes for both treatments (i.e., exhaustive and summary based).

Efficiency experiment results. These results show that for the experimental subjects, performing full instrumentation incurred substantial runtime and space overheads. The summary-based analysis reduced the runtime overhead,[4] on average, from 14,192× (with exhaustive analysis) to 1,106× (with summarized analysis), which shows 13× speedup in the favor of a summary-based approach, on average. (Note: These execution times include the time spent in I/O operations while storing execution traces to disk.) The summary-based approach also reduced the resulting summarized execution traces in size by 53% of the original exhaustive trace sizes. For the experimental subjects in this work, the summary-based approach successfully caused demonstrable reductions for seven subjects. However, the summary-based approach provided limited savings in the case of Antlr (2% savings in traces sizes; 1.3× speedup). That said, in absolute terms, the 2% savings in trace size amounts to saving profiling costs of more than 4.3 million instruction executions. Conversely, the usage of summaries while profiling Jython provides an overwhelming saving of nearly 96% in trace sizes and a 26× speedup.

Answering RQ3: Accuracy Experiment. Motivated by the savings found from Experiment 2, we investigated the extent to which these savings were realized at the

[4]Runtime overhead is a measure of the slowdown of the original noninstrumented program in terms of a multiplicative factor of the original time. The overhead is computed as $\frac{(\textit{running time with analysis}) - (\textit{original running time})}{(\textit{original running time})}$.

expense of accuracy losses. As discussed in Section 1, inaccuracies are introduced in dynamic dependence summaries due to (a) reliance on dynamically collected data and (b) aggregation of multiple abstract summaries into a single aggregate summary. On the one hand, different invocations of a method may exhibit possibly different external heap data effects, and thus a reliance on dynamically collected data possibly may not render unseen dependence relationships for a future method invocation. On the other hand, the aggregation of dynamic summaries, which attempts to model external effects of multiple and varying method invocations, might result in spurious dependence relationships. As such, in this accuracy experiment, we focus our investigation on the inaccuracies introduced as a result of aggregating dynamic dependence summaries.

Using the concrete summaries created by the executions of NanoXML, Jtopas, PL-241, Antlr, Bloat, FOP, PMD, and Jython, we determined the accuracy of abstract aggregated dynamic summaries. We compared each concrete summary for a given method invocation to an aggregated abstract dependence summary of the respective method. Such an aggregate summary was constructed by abstracting and aggregating other concrete summaries of the method, which model other method invocations of the given method within an individual execution. We first set aside a single concrete summary for a given method that we wish to model with an aggregated dynamic dependence summary within an execution; we will refer to such a concrete summary as the hold-out summary. The method's remaining concrete summaries from that execution were then abstracted and aggregated to create an aggregate summary for the method. Such an aggregated summary was then compared with the hold-out concrete summary in terms of the dependence relations between the method's inputs and outputs as captured in the two summaries. Multiple such aggregated summaries are created and compared to a hold-out summary by iteratively treating each concrete summary for a method as a hold-out summary. This enabled us to understand the extent of the inaccuracies, as a result of the over/underapproximations, within the aggregated dynamic dependence summary when it was used to model the dependencies between the concrete hold-out summary. To enable an appropriate comparison with the aggregated abstract summary, the hold-out concrete summary was itself abstracted.

Accuracy experiment independent variable. To understand the effect of aggregation of summaries, we created an aggregated abstract summary using only a sample of available concrete summaries for the given method. As such, for the accuracy experiment, we used the following as our independent variable:

Sample Size (SS). The number of concrete summaries for a single method, within a single program execution, selected for aggregation, as a percentage of the total concrete summaries for the given method within the single program execution. For a given method and program execution, this is computed as

$$\mathbf{SS} = \frac{|\{\textit{concrete summaries selected for aggregation}\}|}{|\{\textit{total concrete summaries}\}|} \times 100.$$

We performed our investigation for the following five sample sizes: 1%, 10%, 25%, 50%, and 100%. These percentages represent fractions of the concrete summaries for a method that are selected for creating an aggregated dynamic dependence summary.

The use of only a limited set of concrete summaries (i.e., SS = {1%, 10%, 25%, 50%}) allowed us to simulate situations where we are unaware of certain dynamic dependencies between the inputs and outputs of a method invocation because they were never observed dynamically. In other words, such a modeling would possibly underapproximate the actual dependencies in a method invocation.

Simultaneously, we also created aggregated dynamic dependence summaries by using all available concrete summaries for a method within a given execution

(i.e., SS = 100%). Such a modeling allowed us to simulate situations where we modeled every known dependency across all invocations for a given method within an execution, thus potentially leading to overapproximations with spurious dependencies.

For each sample size, we obtained a random sample from the pool of concrete summaries for a given method. For instance, a sample size of 100% would imply that we use all available concrete summaries to create the aggregated summary. Similarly, a sample size of 50% would mean that we select half of all method summaries, at random, to construct the aggregate summary.

Note that for a method's summaries to be aggregated at a sample size of 50%, at least three concrete summaries must be present, with two available for aggregation and one concrete summary acting as the hold-out summary. If for a given sampling size the minimum number of summaries are not available to select a random sample, then aggregation and comparison with the hold-out summary is abandoned, for the specific sampling size, for the method in question.

Accuracy experiment dependent variables. To assess the inaccuracies by the means of over/underapproximations within a dynamic dependence summary, we treat the hold-out concrete summary as the ground truth. The hold-out summary is then compared against the aggregated summary in terms of the dependence relations modeled by the two methods. As such, we use the following metrics as our dependent variables:

Metric 1: Precision (P). The fraction of the dependence relations in the aggregate summary that are also present in the hold-out summary. This is computed as

$$\mathrm{P} = \frac{|\{\textit{dependencies in aggregate summary}\} \cap \{\textit{dependencies in hold-out summary}\}|}{|\{\textit{dependencies in aggregate summary}\}|}.$$

Metric 2: Recall (R). The fraction of the dependence relations in the hold-out summary that are also present in the aggregate summary. This is computed as

$$\mathrm{R} = \frac{|\{\textit{dependencies in aggregate summary}\} \cap \{\textit{dependencies in hold-out summary}\}|}{|\{\textit{dependencies in hold-out summary}\}|}.$$

Ideally, both precision and recall for the aggregate summary, with respect to the hold-out summary, should be 1, suggesting a perfect modeling of the dependencies for the method invocation represented by the hold-out summary. That said, a low precision score would indicate that the aggregate summary modeled dynamic dependencies that were not part of the hold-out concrete summary (i.e., overapproximation). At the same time, a low recall score would indicate that the aggregate summary does not model all dependencies that were a part of the hold-out concrete summary, resulting in an underapproximate modeling of the method invocation's dependencies.

The results for the accuracy experiment are summarized in Figures 7 and 8 as a set of box plots that show the distribution of precision and recall scores along the vertical axis, for randomly aggregated method summaries, using specific sampling sizes. Each row in the figures contains the precision and recall box plots, shown in blue and red, respectively, for a single client subject. For instance, the first row of Figure 8 represents the precision and recall scores for methods summarized across all executions of NanoXML. It is important to note that the summaries aggregated and compared to one another belong to the same execution. The resulting scores of precision and recall for the randomly created aggregated summaries are grouped by the sampling size and shown within a single box of a box plot for a given subject program. Such a grouping of the precision and recall distributions (i.e., by sampling sizes) allows us to understand

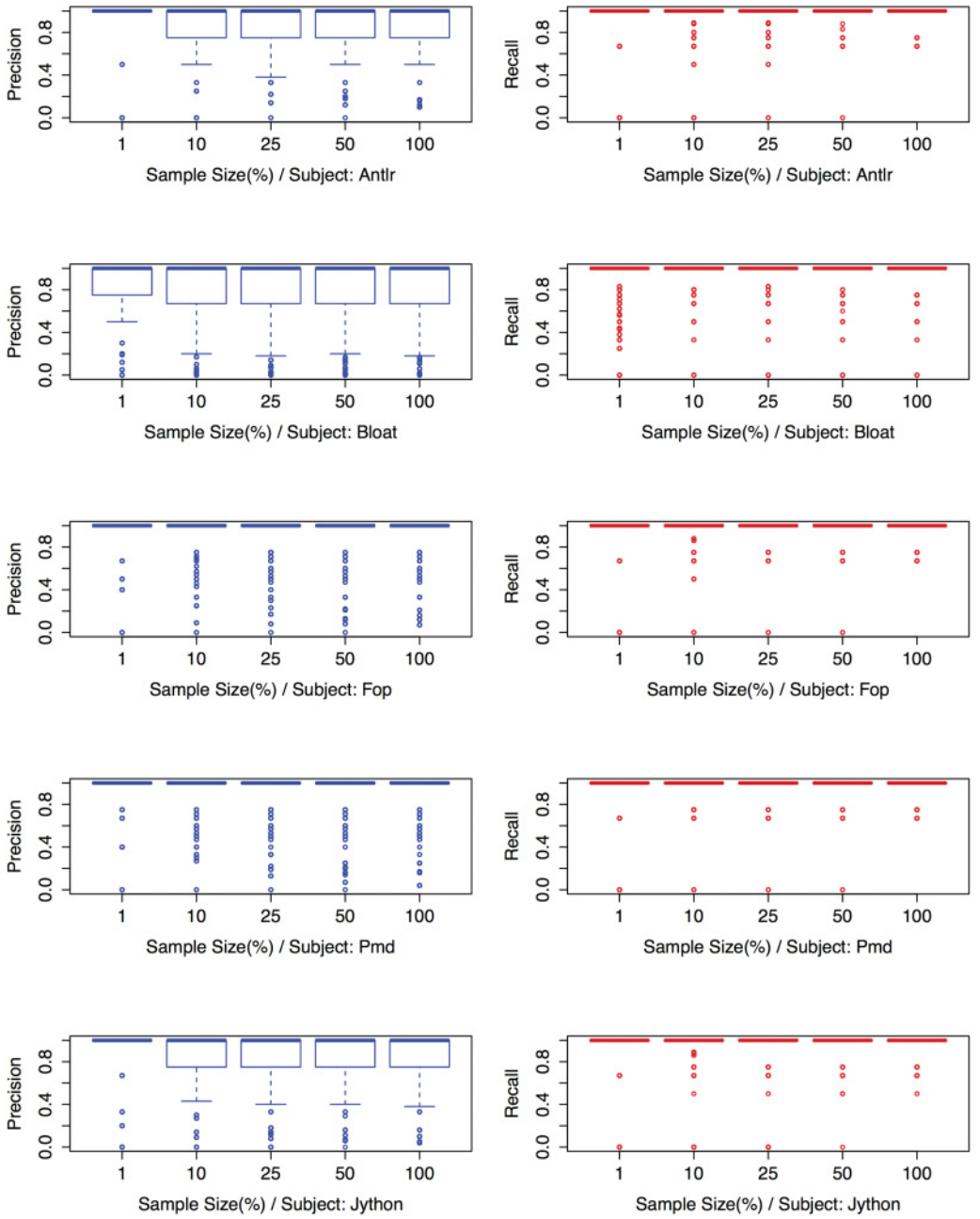


Fig. 7. Assessing the accuracy of summary aggregation with precision and recall scores for ANTLR, BLOAT, FOP, PMD, and JYTHON. Sample Size(%) represents the fraction of the concrete summaries for a given method within a single program execution that are available for summary aggregation.

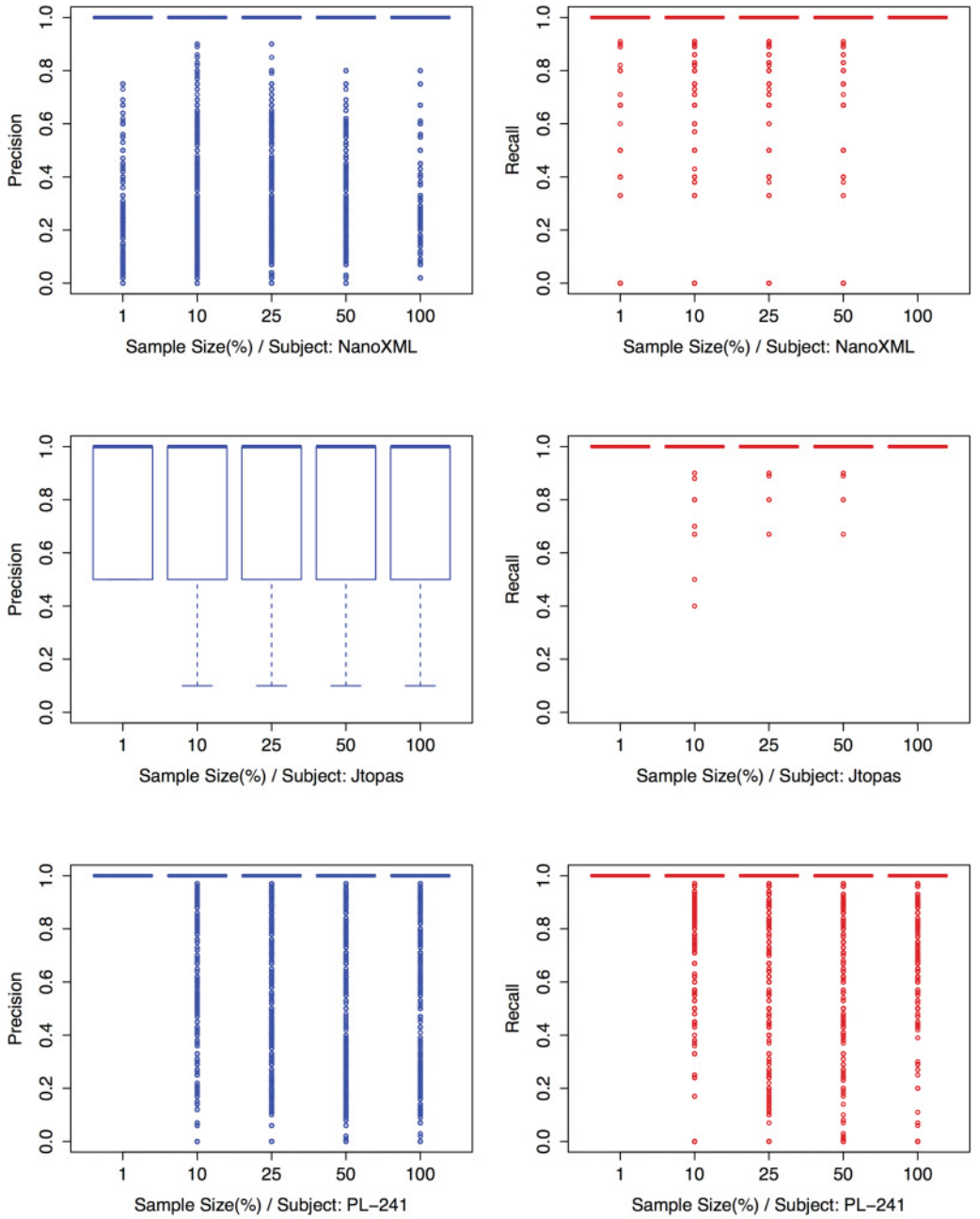


Fig. 8. Assessing the accuracy of summary aggregation with precision and recall scores for NanoXML, Jtopas, and PL-241. Sample Size(%) represents the fraction of the concrete summaries for a given method within a single program execution that are available for summary aggregation.

the impact on accuracy at varying degrees of aggregation—from SS = 1%, which implies limited aggregation, to S = 100%, which implies a high level of aggregation.

Accuracy experiment results.

Recall scores. The box plots in Figures 7 and 8 depict high median scores of recall for randomly aggregated abstract summaries across method executions and even across different subject programs, with the median recall score of 1.0 for all subjects and sampling sizes. This suggests that all dependencies for nearly every given method invocation in a single execution are being modeled. Note that a 1% sampling rate would require nearly all method invocations for their respective methods to be similar in behavior to attain a very high recall score. Such sound modeling could be attributed to a possible consistency in the behavior of methods in the Java library as used by NanoXML, Jtopas, PL-241, Antlr, Bloat, FOP, PMD, and Jython.

Precision scores. All subjects continue to maintain a high median value of 1.0 for precision scores for all of their randomly aggregated summaries with varying sampling sizes. However, unlike with recall distributions, certain subjects, particularly Antlr, Bloat, Jtopas, and Jython, exhibit wider distributions of precision scores, suggesting an overapproximation in the modeling of method invocations. Such distributions of precision scores highlight the effect of spurious relations due to aggregation.

Taken as a whole, the plots for the precision and recall scores suggest that dynamic dependence summaries can be effective at accurately modeling the external heap data effects for reused methods within all experimental subjects presented in this article. Although the results exhibit substantially high levels of precision and recall for dynamic dependence summaries, they also indicate a less than perfect modeling of dynamic dependencies for every single method invocation. This might manifest as possible inaccuracies in downstream client analyses that rely on dynamic dependence summaries, which we investigate next with a case study.

Answering RQ4: Summarized Dynamic Slicing Case Study. Motivated by the cost savings found from the efficiency experiment (*RQ2*) and the accuracy measures of dynamic dependence summaries from the accuracy experiment (*RQ3*), we next investigated the extent to which these savings are realized at the expense of accuracy for an actual client analysis. In this case study, we investigate the impact of reused dependence summaries on the accuracy of dynamic dependence analysis, specifically dynamic slicing for debugging. As discussed in Section 1, and further in Section 9, summary-based dynamic dependence analysis can be both unsound and imprecise. However, the degree to which this affects the results of an analysis in practice is yet to be known within the context of an actual software engineering technique. As such, this case study is designed to answer this question, at least for our experimental software subject, NanoXML.

Using a training input of NanoXML, and the 20 subsequent faulty inputs, we investigate this issue by performing backward dynamic slicing to determine accuracy of the slice. The training input for NanoXML, which is different from the 20 faulty inputs, is used to generate dynamic summaries for the Java library methods used by NanoXML. We injected corruptions, or "faults," in the the 20 inputs to NanoXML that resulted in corrupt and incorrect execution state early in NanoXML's execution—during the reading of the data files. We identified our slicing criterion for each corruption across the 20 inputs by observing the output stream and identifying the first point that the output from the faulty input differs from the output of the correct input. In other words, we define the slicing criterion as the output instruction (and its output data) being executed at the moment that the test case output first violates the test oracle (e.g., the first character difference). We then slice based on that instruction, the variable that was used to hold the externally observed incorrect data, and the specific

Table III. Experiment 3 Results

Metric	Exhaustive	Summary
Found Bugs (ratio)	20/20	18/20
Mean Size of the Slice (# of runtime-instructions)	5019.2	7093.1
Mean Slicing Time (seconds)	84.5	67.2
Mean Program Runing Time (seconds)	527.74	1.70
Mean Program Runtime Overhead (ratio)	7538.1	23.3

execution instance of that instruction (remember, our traces include all execution instances of each instruction). The goal was to localize the first runtime instruction in each of NanoXML's 20 executions that read in the incorrect data, which we treat as the faulty instruction instance for the purpose of this experiment. As such, the faults that we slice are "deep"—the fault execution occurs at the beginning of the execution trace during the reading of input data, the propagation of the fault's state infection spans nearly the entire execution, and the slicing criterion is placed at the output manifestation near the end of the execution.

Case study metrics. We present our results with the following metrics:

Metric 1: Found Bugs (B). The ratio of the dynamic slices that include the fault. This metric determines the degree of unsoundness that the summary-based dynamic analysis brings.

Metric 2: Size of the Slice (S_{slice}). The size of the resulting slice as a set of runtime instructions.

Metric 3: Slicing Time (T_{slice}). The time required to compute the dynamic slice.

Metric 4: Program Runtime Overhead (RO). A measure of slowdown of the original noninstrumented program in terms of a multiplicative factor of the original time. The overhead is computed as

$$\mathrm{RO} = \frac{(\textit{instrumented time}) - (\textit{noninstrumented time})}{(\textit{noninstrumented time})}$$

.

We present our results for the case study in Table III and Figures 9 and 10. Table III shows the aggregated data across all 20 bugs, whereas Figures 9 and 10 break down results per bug. Figure 9 presents the size of the resulting slices for each bug for each technique in terms of the unique source-code instructions in the slice. Figure 10 presents the time required to compute the dynamic slice.

Case study results. When examining the bugs that were being inspected, we found that 18 out of the 20 bugs were localized in the dynamic slice that we computed with the summary-based technique. Such a localization rate is attributed to faults propagating through multiple dependency chains, and as such, the ability to slice back to the fault depends on at least one such chain persisting. The exhaustive slicing technique found 100% of the bugs, as expected. The summary-based technique missed 2 out of the 20 bugs, and this was due to a missing dependency from the training of a summary.

It is also worth noting that the slice sizes on average are larger with the use of dependence summaries, suggesting an introduction of spurious dependence edges by the dynamic dependence summaries. However, certain slices are smaller when using dependence summaries, suggesting dependence edges that were not modeled by the summaries for certain methods. These effects are indicative of the unsound and imprecise nature of how dependencies are modeled by dynamic dependence summaries. Essentially, these results are congruent with the results presented in the accuracy

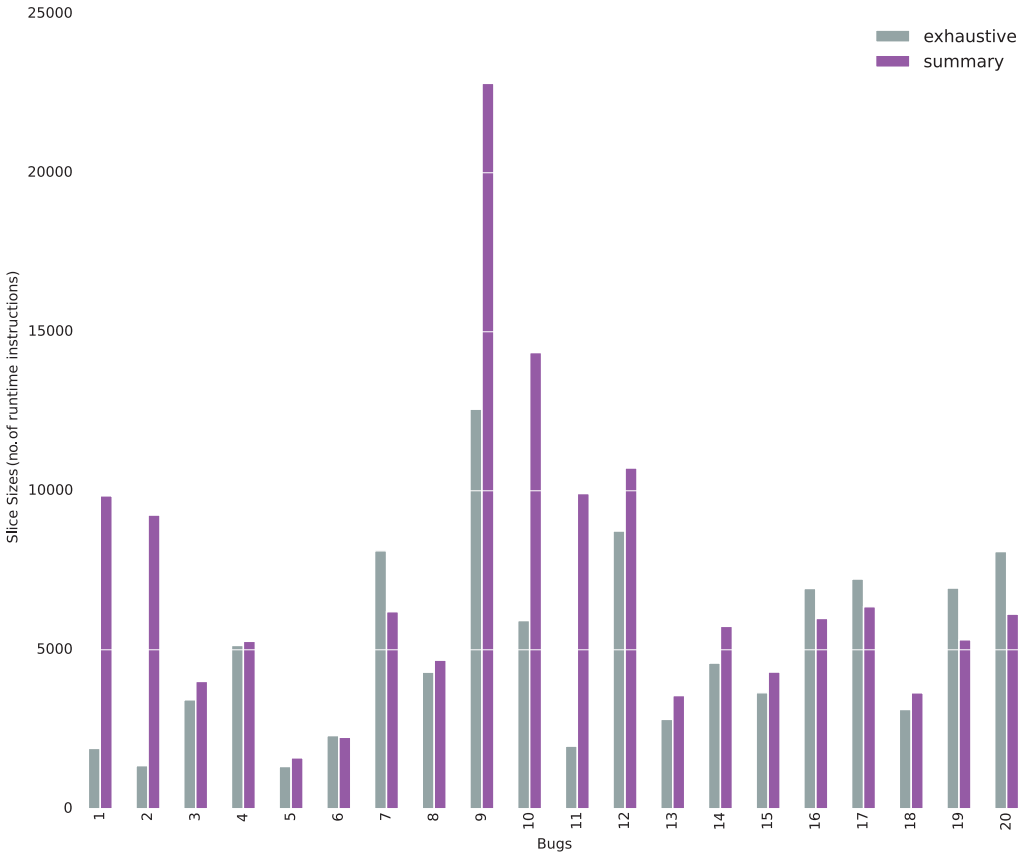


Fig. 9. Slice sizes (S_{slice}).

experiment (carried out to answer *RQ3*) that indicated small losses of accuracy on the reuse of dynamic summaries.

Considering the costs of analyzing the program's execution (indicated by the program runtime overhead) and the cost of computing the dynamic slice, the costs for exhaustive dynamic slicing are substantially more than the costs for summarized dynamic slicing. Particularly, the runtime overhead costs of exhaustive monitoring of a program's execution were dramatically greater than those of the summary-based approach—and these are costs that would be incurred for every execution to be sliced.

8. ANALYSIS AND DISCUSSION

In this section, we present a discussion of all four studies to draw more general conclusions. In addition, we discuss each research question that motivated our evaluation, in order. In our first experiment, we investigated the extent to which instructions from the Java standard library code are a part of a software execution. With seven client subject programs, there was a substantial reliance on the execution of instructions from library code, with certain executions relying on up to 95% of the execution on instructions from library code. A single execution of ANTLR showed the least reliance (1.97%) on library code. That said, even the 1.97% reliance of library code amounted to 4.3 million instructions from the Java libraries in the case of ANTLR.

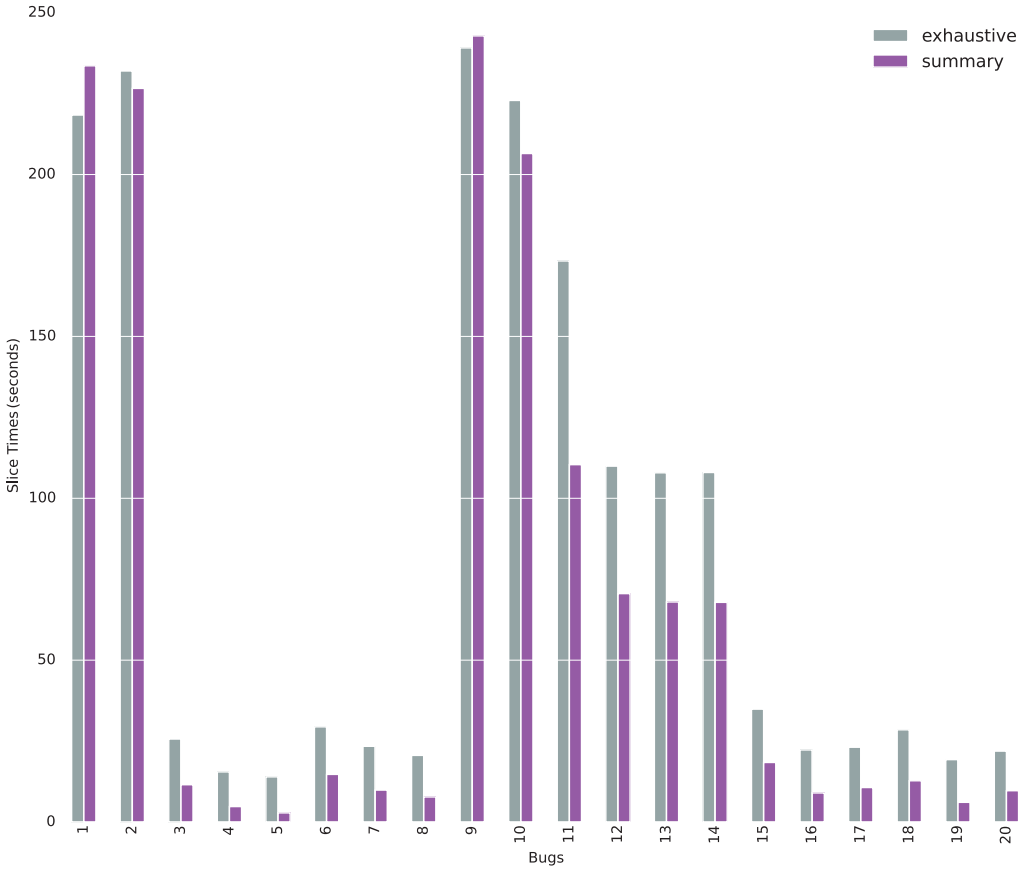


Fig. 10. Slicing time (T_{slice}).

As such, to *RQ1*, we assess: A substantial number of instructions are executed with the methods belonging to external libraries, thus suggesting potentially high cost savings.

With the efficiency experiment, we investigated the extent of performance gains with the use of a summary-based approach. In every measure of performance costs, we found that as a whole, the use of summarized dynamic dependencies substantially reduced costs. On average, for the 40 executions across the eight subjects, we observe a 13× speedup in executions and a 53% savings in execution trace sizes. When examining the results from Experiment 2, we find that the exhaustive technique often exceeded our thresholds for costs, whereas the summary-based technique was efficient, in all but one execution.

As such, to *RQ2*, we assess: The reuse of dynamic summaries caused the costs involved in performing dynamic dependency analysis to be significantly reduced.

In terms of analysis accuracy, we evaluated the effects of aggregating concrete summaries and using dynamically collected information to create dependence summaries. We aggregated random samples of concrete summaries for individual method

invocations within a single execution and compared the resulting summary to hold-out summaries. When examining the results from Experiment 3, we find that although we are utilizing summarized results from within the same execution to approximate the dependencies of external components, the summarization overapproximated dependencies between inputs and outputs of methods in some cases.

We further inspected some instances of method invocations for which precise summaries were not generated. Our anecdotal evidence revealed that imprecision in modeling summaries in some cases occurs due to exceptional control flow exhibited by corner cases. For instance, consider the `get(int index)` method in `java.util.ArrayList`, whose summary shows that its return value depends on an internal data array that is a field to the `ArrayList` object, and the integer argument `index`, as showcased earlier in Figure 6. However, certain invocations of the `get(int)` method would result in an `ArrayOutOfBounds` unchecked exception, resulting in the lack of a return value and thus exhibiting no outputs with dependencies on the method's inputs. However, such an invocation would still be modeled by a dependence summary that includes a return value as an output, resulting in an incorrect method summarization. Similarly, consider the `get(Object key)` method in `java.util.HashSet` that can either return an object or a null value if the input `key` does not exist. Although the dynamic summary for this method models both dependencies, in an actual invocation only one of the two dependencies will be exhibited: "`return` value depends on object" or "`return` value depends on `null`." Thus, resulting in an overapproximated dependence summary for the `HashMap.get`'s `return` value.

That said, in the majority of cases, the resulting summaries did exhibit sound and precise approximations of the dynamic dependencies between inputs and outputs for specific invocations of specific methods. Such accuracy results for different sampling sizes and subject programs indicate that a method's invocation may be summarized effectively by monitoring only a sample of the method's invocations instead of observing the totality of its invocations in a given execution. Based on these preliminary findings, we speculate that methods exhibit a limited set of behaviors that may be modeled by observing a limited number of invocations. As such, we envision practical applications where we are able to use one set of method inputs to generate method summaries and use the resulting summaries to model behavior of method invocations that accept entirely different inputs. The accuracy results, specifically the recall scores at 100% sampling sizes, suggest that method summaries are likely to comprehensively model method behavior with greater degrees of training. However, the extent of training necessary for generating comprehensive method summaries requires further investigation, which we discuss as future work in Section 9.

As such, to *RQ3*, we assess: Dependence summaries created using dynamically observed data resulted in the generation of sound dependence summaries within the context of a single execution, with a perceptible loss of accuracy, leading to imprecise dependence summaries in some cases.

To investigate the cumulative effects of using dynamic dependence summaries in a real-world software engineering context, we used our summary-based approach in dynamic slicing with the goal of localizing bugs in program executions. We also examined the accuracy of the summary-based dynamic slicing in the context of performance gains made as a consequence of summarization. When examining the results of our dynamic slicing case study, we find that although we are utilizing summarized results from a past execution to approximate the dependencies of external components in subsequent executions, the summarized slicing technique produced accurate results: 90% of the bugs were found with the summary-based technique.

As such, to *RQ4*, we assess: The reuse of dynamic summaries caused a small loss of accuracy as a consequence of the gains in performance.

Taken as a whole, our results suggest that the reuse of dynamic summaries can provide a way to make dynamic dependency analysis more feasible for real-world use with modest losses in accuracy. In software development projects that are able to tolerate inaccuracies (e.g., in noncritical systems), or for analyses that are more heuristic in nature (e.g., bloat analysis), our results suggest that the trade-offs favor reuse of dynamic summary information for external components.

Threats to Validity. Threats to external validity arise when the results of the experiment are unable to be generalized to other situations. In this experiment, we evaluated the impact of using dynamic dependency summaries on eight client programs, with their set of dependent external components, and thus we are unable to definitively state that our findings will hold for programs in general. However, we are confident that these results are indicative of the impacts of such summarization. The external components in this study are the Java standard library, which is a dependency that many other programs will also have and thus the effects on efficiency and effectiveness of dynamic summarization of that common library is likely to hold for those programs. Moreover, the significant gains exhibited in our results gives strong evidence that our approach, at least, shows promise for use in practice.

Threats to construct validity arise when the metrics used for evaluation do not accurately capture the concepts that they are meant to evaluate. Our experiments measured the costs involved in performing tracing and dynamic dependence analysis in terms of computational time and data storage. Although our results give an indication of the degree of such costs, our implementation can be greatly optimized in both regards. Our tracing information is verbose, and our implementation is not optimized. However, this limitation does not affect the overall result, as this same implementation was used for both treatment techniques (exhaustive and summarized)—that is, the direction and magnitude of the difference between the results should not significantly change when these factors are optimized. In addition, our experiments measured the inaccuracy introduced in method summaries as a process of abstraction and aggregation. Although these metrics give an indication of the accuracy of our results, they do not give a sense for how these will affect either developer time or client analyses that build on such results. Further studies will need to be conducted to assess the impact on such clients of these analyses.

9. LIMITATIONS AND FUTURE WORK

Unsoundness and Imprecision. It is important to note that the summary-based dynamic dependence analysis can introduce both unsoundness and imprecision. On one hand, the quality of an abstract summary relies on the coverage of the tests used to train the summary. Thus, a summary may miss certain dependence relationships due to the lack of test cases. On the other hand, the abstract summary aggregates information from multiple executions of the method. Thus, the application of the summary for a specific invocation may generate additional spurious dependence relationships that would not have been added in a regular dependence analysis.

To reduce these negative effects, it is important to find methods that are suitable for summarization. Our experience shows that API methods in large libraries may be good sources of summarizable methods because they often do not mutate client objects, their behaviors are often relatively simple, and even similar under different clients. In our evaluation (Section 7), the entire Java standard library is summarized to speed

up the dynamic slicing of a real-world application with perceptible, although limited, losses in accuracy while abstracting dynamic dependence summaries.

Future work will develop ways to assess suitability of summarization for methods. In addition, adequacy criteria will be developed to inform when the training phase has sufficiently exercised the behaviors to summarize them.

Abstracting Multiple Array Accesses. Although our technique improves on existing work in distinguishing array index accesses, this improvement is limited to method calls that access only a single index. We found that most of our studied methods accessed a single index. Even for methods that access multiple array locations (e.g., `addAll` in many of the `java.util.*` methods), they are often implemented by invoking methods that access a single array location (e.g., `add`) multiple times. Hence, our technique can precisely handle array accesses for most of the common and frequently used data structure methods.

Future work will provide yet further improvement to array summarization by developing ways to summarize access to multiple array indices. We envision techniques that rely on light-weight instrumentation to accurately identify accesses to array elements.

Using Static Analysis for Summary Generation. Our approach to generate dependence summaries improves on existing works by accounting for variance in method behavior due to polymorphism. However, it still falls short in accurately modeling variance in method behavior due to varying control flow paths followed by different executions of the same method with the same input parameter types.

Future work will investigate hybrid approaches of dependence analysis to generate accurate summaries with greater efficiency. Such a summary generation approach would use static analysis to produce static summaries for all alternatives and light-weight dynamic instrumentation to selectively choose the right summary edges from the static summaries. We envision that light-weight instrumentation will account for varying behavior due to dynamically observed control flow, accesses to individual array elements, and dynamic dispatch. In addition, we will also investigate how dynamic summaries compare to summaries computed with static analysis alone for various expensive dynamic analyses in terms of performance and accuracy.

Further Extensions to Current Evaluation. Additionally in future work, we will further expand our current evaluations on a larger set of applications and extend it to address scalability issues for specific downstream dynamic client analyses (e.g., dynamic-slicing, bloat-, and change-impact analysis). We will also study the reuse dynamic summaries at varying depths of the call stack during program executions. Such an investigation would potentially extend the scope of summarization to any given component within an application, not just library or external components.

Future studies will investigate the reuse of summaries when generated with one client program and used in a completely different client program, as against generating and using summaries with executions of the same client program. The goal of such studies will be to identify the extent of training necessary to model dependence summaries for method behavior. We will also examine a plausible correlation between the size of a method invocation, and the accuracy of the dependence summary, especially given that a wider set of dependencies would be exercised with longer method invocations due to likely issues such as more complex control flow.

10. RELATED WORK

Although there exists a body of work related to the proposed technique, discussion in this section focuses on techniques that are most closely related to ours.

Summary-Based Program Analysis. Procedural summaries have been computed and used widely in the static analysis community to achieve modularity and efficiency. Summary functions for interprocedural analysis date back to the functional approach in the work by Sharir and Pnueli [1981], with refinements for interprocedural, finite, distributive, subset (IFDS) problems from Sagiv et al. [1996] and for interprocedural distributive environment (IDE) dataflow problems from Reps et al. [1995]. Yorsh et al. [2008] use conditional microtransformers to represent and manipulate dataflow functions. Rountev and colleagues ([Rountev et al. 2006, 2008]) propose a graph representation of dataflow summary functions that generate and use summaries to reduce the cost of whole-program IDE problems. Xu et al. [2009] propose a summary-based analysis that computes access path–based summaries to speed up the context-free-language (CFL)-reachability–based points-to analysis. Dillig et al. [2011] propose a summary-based heap analysis targeted for program verification that performs strong updates to heap locations at call sites. Ranganath and Hatcliff [2004] statically analyze the reachability of heap objects from a single thread for slicing concurrent Java programs. Salcianu and Rinard [2005] propose a regular expression–based, purity and side-effect analysis for Java to characterize the externally visible heap locations that a given method mutates. Tang et al. [2015] propose a static analysis–based method to summarize library methods specifically in the presence of callbacks in the library methods. This is similar in spirit to the heap location–based dependence summarization as proposed in our work. Inspired by such static analyses, the work presented in this article is the first technique to compute and use dynamic dependence summaries. Dynamic dependence summaries can potentially better inform dependence information more precisely than static dependence summaries by leveraging information collected at runtime.

Dynamic Dependence Analysis. Since first being proposed by Agrawal and Horgan [1990], dynamic dependence analysis has inspired a large body of work on a variety of software engineering tasks from debugging to memory bloat analysis. Early work from Kamkar et al. [1992] introduces the theory behind summary-based dependence slicing for a stack-only language. Our approach attempts to be more general, in that it handles all features of a modern object-oriented language. The work by Zhang and colleagues [Zhang et al. 2003; Zhang and Gupta 2004a, 2004b; Zhang et al. 2006a; Zhang et al. 2006b] considerably improved the state of the art in dynamic dependence analysis and its applications like dynamic slicing and fault localization. This work includes, for example, a set of cost-effective dynamic slicing algorithms [Zhang et al. 2003; Zhang and Gupta 2004a], a slice-pruning analysis that computes confidence values for instructions to select those that are most related to errors [Zhang et al. 2006a], a technique that performs online compression of the execution trace [Zhang and Gupta 2004b], and an event-based approach that reduces the cost by focusing on events instead of individual instruction instances [Zhang et al. 2006b]. Apart from Zhang's work, Wang and Roychoudhury [2004] develop optimizations to compress bytecode execution traces for sequential Java programs resulting in space efficiency. Moreover, they develop a slicing algorithm applicable directly on the compressed bytecode traces. Hierarchical dynamic slicing [Wang and Roychoudhury 2007] aims at guiding the programmer through large and complex dependence chains in a dynamic slice, with debugging as the primary application. Deng and Jones [2012] propose a dynamic dependence graph that encodes frequency of inferred traversal to prioritize heavily trafficked flows. Xu et al. [2010] propose an abstraction-based approach to scale a class of dynamic analyses that need the backward traversal of execution traces. This approach employs user-provided analysis-specific information to define equivalence classes over instruction instances so that dynamic slicing can be performed over bounded abstract domains instead of concrete

instruction instances, leading to space and time reduction. Our technique achieves efficiency from a different angle—we use summaries generated for library classes to speed up general dynamic data dependence analysis, and thus all dynamic analyses that need dependence information may benefit from this technique.

11. CONCLUSIONS

This article presents a summary-based dynamic analysis approach to effectively perform dynamic analysis techniques for modern applications that use large object-oriented libraries and components. During training, summaries are produced for library methods, which are later used for dependence analysis rendering improved efficiency. To compute the summary for a library method, we first extract concrete dependence relations between a method invocation's inputs and outputs from its execution trace and then abstract them with symbolic data. This symbolic data is then used during subsequent analysis by replacing symbolic names with the new concrete data. Our experimental results on real-world software found that a heavy reliance on external components is a reality and that applying these summaries in the dependence analysis can significantly save costs. Our results show that summary abstraction does well to capture varying method behaviors despite potentially causing modest loss in accuracy.

REFERENCES

Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI'90)*. 246–256.

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. 169–190.

E. Bruneton, R. Lenglet, and T. Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems Paper presented at the meeting of Adaptable and Extensible Component Systems.

Fang Deng and James A. Jones. 2012. Weighted system dependence graph. In *Proceedings of the 2012 IEEE 5th International Conference on Software Testing, Verification, and Validation (ICST'12)*. IEEE, Los Alamitos, CA, 380–389.

Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and compact modular procedure summaries for heap manipulating programs. In *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation (PLDI'11)*. 567–577.

Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10, 4, 405–435.

Justin Holewinski, Ragavendar Ramamurthi, Mahesh Ravishankan, Naznin Fauzia, Louis-Noel Pouchet, Atanas Rountev, and P. Sadayappan. 2012. Dynamic trace-based analysis of vectorization potential of applications. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI'12)*. 371–382.

Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* 12, 1, 26–60.

Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. 1992. Interprocedural dynamic slicing. In *Programming Language Implementation and Logic Programming*. Lecture Notes in Computer Science, Vol. 631. 370–384.

Bogdan Korel and Janusz Laski. 1990. Dynamic slicing of computer programs. *Journal of Systems and Software* 13, 3, 187–195.

Doug McIlroy. 1968. Mass-produced software components. In *Proceedings of the NATO Conference on Software Engineering*. 88–98.

James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*.

Alessandro Orso, Hyunsook Do, Gregg Rothermel, Mary Jean Harrold, and David S. Rosenblum. 2007. Using component metadata to regression test component-based software: Research articles. *Journal of Software Testing, Verification and Reliability* 17, 2, 61–94.

Alessandro Orso, Mary Jean Harrold, and David S. Rosenblum. 2001. Component metadata for software engineering tasks. In *Proceedings of the International Workshop on Engineering Distributed Objects*. 129–144.

Vijay Krishna Palepu, Guoqing Xu, and James A. Jones. 2013. Improving efficiency of dynamic analysis with dynamic dependence summaries. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. 59–69.

Venkatesh Prasad Ranganath and John Hatcliff. 2004. Pruning interference and ready dependence for slicing concurrent Java programs. In *Compiler Construction*. Lecture Notes in Computer Science, Vol. 2995. Springer, 39–56.

T. Reps, S. Horwitz, and M. Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*. 49–61.

Atanas Rountev, Scott Kagan, and Thomas Marlowe. 2006. Interprocedural dataflow analysis in the presence of large libraries. In *Compiler Construction*. Lecture Notes in Computer Science, Vol. 3923. Springer, 2–16.

Atanas Rountev, Mariana Sharp, and Guoqing Xu. 2008. IDE dataflow analysis in the presence of large object-oriented libraries. In *Compiler Construction*. Lecture Notes in Computer Science, Vol. 4959. Springer, 53–68.

Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1–2, 131–170.

Alexandru Salcianu and Martin Rinard. 2005. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*. 199–215.

M. Sharir and A. Pnueli. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones (Eds.). Prentice Hall, 189–234.

Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-based context-sensitive data-dependence analysis in presence of callbacks. *ACM SIGPLAN Notices* 50, 83–95.

Tao Wang and Abhik Roychoudhury. 2004. Using compressed bytecode traces for slicing Java programs. In *Proceedings of the International Conference on Software Engineering (ICSE'04)*. 512–521.

Tao Wang and Abhik Roychoudhury. 2007. Hierarchical dynamic slicing. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)*. 228–238.

Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding low-utility data structures. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI'10)*. 174–186.

Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'09)*. 98–122.

Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating precise and concise procedure summaries. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*. 221–234.

Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006a. Pruning dynamic slices with confidence. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06)*. 169–180.

Xiangyu Zhang and Rajiv Gupta. 2004a. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*. 94–106.

Xiangyu Zhang and Rajiv Gupta. 2004b. Whole execution traces. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*. 105–116.

Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. 2003. Precise dynamic slicing algorithms. In *Proceedings of the International Conference on Software Engineering (ICE'03)*. 319–329.

X. Zhang, S. Tallam, and R. Gupta. 2006b. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the 13th Annual Fast Software Encryption Workshop (FSE'06)*. 81–91.

Received February 2015; revised June 2016; accepted July 2016