TRACERJD: Generic Trace-Based Dynamic Dependence Analysis with Fine-Grained Logging

Haipeng Cai and Raul Santelices
University of Notre Dame, Indiana, USA {hcai|rsanteli}@nd.edu

***Abstract*—We present the design and implementation of TRACERJD, a toolkit devoted to dynamic dependence analysis via fine-grained whole-program dependence tracing. TRACERJD features a *generic* framework for efficient offline analysis of dynamic dependencies, including those due to exception-driven control flows. Underlying the framework is a hierarchical trace indexing scheme by which TRACERJD maintains the relationships among execution events at multiple levels of granularity while capturing those events at runtime. Built on this framework, several application tools are provided as well, including a dynamic slicer and a performance profiler. These example applications also demonstrate the flexibility and ease with which a variety of client analyses can be built based on the framework. We tested our toolkit on four Java subjects, for which the results suggest promising efficiency of TRACERJD for its practical use in various dependence-based tasks.**

Keywords*—*Dependence analysis, tracing, slicing, profiling

I. INTRODUCTION

Analyzing program dependencies is a fundamental approach to program understanding, testing, debugging, and a wide range of other software evolution tasks [1]. Compared to static dependence analysis, which attempts to model program behavior for all possible program inputs and, thus, tends to be overly conservative, dynamic dependence analysis is usually more precise, as it focuses on specific sets of inputs used by developers in concrete task contexts. In a typical usage scenario, dynamic dependence analysis finds the subset of a program dependent on a given point of that program with respect to the concrete set of executions used by the analysis.

Various techniques have been proposed to this date for dynamic program dependence analysis (e.g., [2]–[5]), for which a common approach is to collect program execution traces such that dynamic dependence information can be later retrieved from the traces after the execution. For that purpose, many algorithms and tools exist, serving system diagnosis [6], error detection [7], [8], fault localization [9]–[12], and program understanding in general [13]–[15].

However, such solutions target either high-level system states (e.g., [6], [7]) and/or coarse runtime conditions (e.g., [14], [15]), which do not capture fine-grained dependence information, or particular tasks concerning specific program points of interest such as dynamic slicing criteria (e.g., [9], [10], [12], [16], [17]), which do not provide common dynamic information to support a broad range of dependence-based applications. More general-purpose traces that could be used for computing dynamic dependencies also exist [4], [5], yet they either lack direct tool supports for dependence computation or would require special compilation or runtime environments to do so, for Java programs in particular. A few dependence-analysis tools are available (e.g., [18], [19]), which, however, aim at *static* dependence analyses only.

Therefore, in this paper, we present TRACERJD, a fine-grained whole-program tracing tool devoted to dynamic dependence analysis of Java software, which runs on a standard JVM, thus it is widely applicable to different platforms. The core of TRACERJD consists of dedicated facilities for tracing program states at variable level and also execution events at method and statement levels, while featuring a scheme of structured logging and hierarchical trace indexing that facilitates efficient offline trace analysis. The tracing framework of TRACERJD takes a program and its set of inputs, performs a one-time whole-program analysis, and then logs sufficient information that enables retrieving data and control dependencies from traces generated at runtime, including those that result from exception-driven control flows [20].

On top of this framework, TRACERJD comes with also two major example application tools, a precise backward dynamic slicer [2] and a statement-instance-level performance profiler, that demonstrate the flexibility and ease with which various client analyses can be built on the framework. We implemented the toolkit in Java and applied it to several open-source Java subjects.[1] Our empirical results show that TRACERJD incurs reasonable time and space costs for the entire tracing process and causes relatively small execution slowdowns. The performance of the dynamic slicer also suggests promising efficiency of client analyses of the tracing framework.

II. TRACERJD ARCHITECTURE

Figure 1 depicts the architecture of TRACERJD, which consists of three major components that correspond to the three TRACERJD phases: static analysis, runtime tracing, and trace analysis. In addition, the connections among these components show the workflow process of this tool: It takes the input program P under analysis and its input set (e.g., test suite) T for the first and second phases, respectively, and outputs dynamic dependence information using the dependence querying interface in the last phase, on demand of client requests for application analyses.

The first phase (*static analysis*) inputs the Java bytecode of P, instruments P with probes for fine-grained logging, and outputs primarily the instrumented version P' of P. Additional outputs of this phase (auxiliary static data) include the interprocedural control dependence graph (ICDG) of P, a variable index, and a statement index. The static analysis is built on the Soot framework [18], which provides low-level bytecode parsing, intermediate representation (IR), and manipulation. In particular, TRACERJD uses the Jimple IR for the logging instrumentation. In the rest of this paper, We use *statement* to refer to a *Jimple statement* unless special notes are given. The ICDG construction step can be opted out by users who focus on data dependence analysis only, and generation of the two auxiliary indexes can be skipped too if users do not need them for client analyses.

[1]Downloads of TRACERJD and all supporting materials (usage demo and documentation) are available at http://sourceforge.net/projects/tracerjd/.

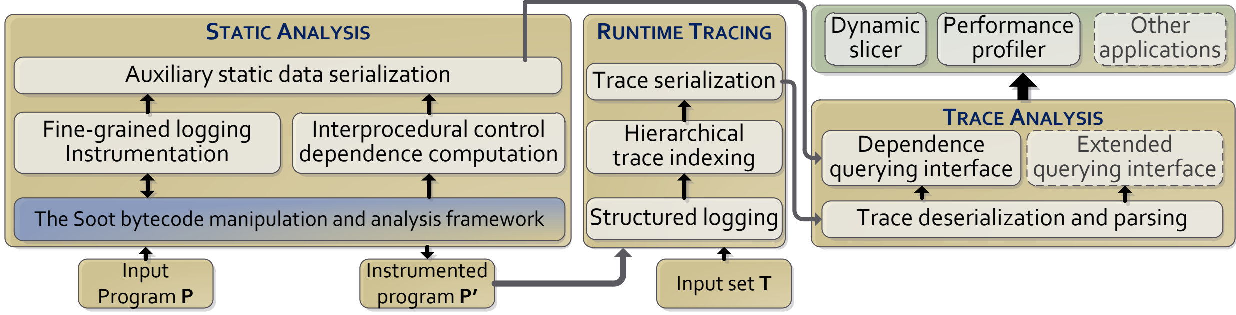

Fig. 1: The architecture of TRACERJD, which shows the composition and workflow process of our tracing framework.

The second phase (*runtime tracing*) aims to collect fine-grained dependence traces produced by the logging probes instrumented in the first phase, by running P' on input set T. For efficient trace analysis, this phase features a structured logging strategy which uses separate monitors dedicated to method-, statement-, and variable-level execution events to collect corresponding levels of runtime information, followed by a hierarchical trace indexing scheme which creates and maintains the hierarchical relationships among the multiple-level events. The output of this phase is the execution traces of P' produced and compressed during runtime.

The last phase (*trace analysis*) starts with deserializing, decompressing, and parsing the traces produced in the previous phase. On top of this trace-loading layer, the dependence querying interface consists of a group of common subroutines that retrieve dynamic dependencies from the loaded traces to be utilized by client analyses. An additional input to this layer is the (optional) auxiliary static data, where more dependence querying subroutines (*extended querying interface*) can be added depending on further application needs. On top of this trace-analysis phase is the example client analyses that are also included in the TRACERJD toolkit, for both providing readily usable tools and illustrating how to build various applications atop the core tracing framework. The dynamic slicer and performance profiler, and possible other application tools to be created by users, are not parts of the framework itself.

III. STATIC ANALYSIS

The main step of the static analysis is the instrumentation of probes for fine-grained logging, which directly effects the feasibility and complexity of later dynamic dependence querying, and the performance of the entire framework.

A. Logging Instrumentation

With TRACERJD, we aim at an efficient trace-based fine-grained dynamic dependence analysis framework. To that end, we log events at three structural levels: method, statement, and variable. Accordingly, we instrument the input program with three major categories of probes, each exclusively logging events at one of the three structural levels.

Method event probes monitor method entrance and exit events, for which three types of exit point are considered: normal return statement, entrance of catch block (if any), and entrance of finally block (if any); for each method, exactly one entrance-event probe is inserted, while each exit point is instrumented with an exit-event probe. **Statement event probes** monitor the coverage events of statements, each instrumented for one statement. **Variable event probes** monitor detailed information of all variables defined or used at a statement, each instrumented for one definition or use; the per-variable information logged includes (1) variable index with a sign indicating it being a definition or use, (2) object address of array/field base (for heap variables only), and (3) array index (for array elements only). Optionally, the variable information includes also variable values computed at runtime.

Since the correctness of subsequent dependence retrieval relies on the ordering of these events, the event probes must be inserted in an order that observes the semantics of those events at runtime. Our experience of ensuring such ordering with respect to instrumentation is to insert the probes such that (1) for variable events, a use event u happens always before the event of definition that depends on u; (2) for statement events, the coverage event happens immediately before the statement execution; and (3) for method events, all entrance and exit events happen immediately before method entrances and exits, respectively. To ensure that all exit events are captured, we wrap the entire body of each method with a *try-catch* block to catch the exit events due to any uncaught *throwables* [20]. Finally, no instrumentation should cause the instrumented program to fail the JVM bytecode verifier.

B. ICDG Construction

To reduce the size of the instrumented code and simplify the retrieval of dynamic interprocedural control dependencies (ICDs) during trace analysis, we chose to create the ICDG in the static-analysis phase instead of logging control-dependence related events and then entirely rely on the traces to recover such dependencies. Although the ICDG is static and conservative, dynamic ICDs can be precisely retrieved later using this ICDG with statement events from the traces.

To build the ICDG, we first create the control dependence graph (CDG) for each method based on the *exceptional control flow graph* (ExCFG) of Soot [18] and then transitively compute ICDs among all methods from the per-method CDGs [21]. In addition, we adapted multi-headed and multi-tailed ExCFGs by adding virtual *start* and *end* nodes joining all head and tail nodes, respectively. Also, for programs containing infinite loops (e.g., service daemons) which result in *tailless* ExCFGs, we treated all jumping statements for the outermost infinite loops as exit nodes. These treatments enabled us to directly apply existing control-dependence computation algorithms.

For space efficiency, we store the ICDG using bitvectors to encode the connections among ICD sources and targets, which are serialized to a disk file as are the traces. Finally, the two mappings (index files), from integers (as indexes) to method signatures and statement texts, respectively, are serialized to separate files as the auxiliary static data. These indexes are used in traces mainly for efficiency reasons.

IV. Runtime Tracing

To facilitate dynamic dependence querying during trace analysis, we build a hierarchical storage of traces at runtime with a structured logging strategy.

A. Structured Logging

Besides the specific order of event-logging probes considered during the instrumentation, additional steps need be taken in runtime monitors of those events to ensure correct ordering of all monitored events in the traces. To that end, we assign a unique timestamp to each event using a global time counter that starts from one and increases by one on each event occurrence. Then, we use dedicated runtime monitors to log events at the three structural levels as described before.

This logging strategy works by catching the three types of events with the corresponding event probes instrumented during static analysis. The rationale is that the *inclusion* relations among methods, statements, and variables drive those probes to log corresponding events in a structured manner, which directly supports the generation of hierarchical traces.

B. Hierarchical Trace Indexing

Hierarchical traces are produced from the structured logging with necessary communications among the separate monitors. Additional data structures are used to support such communications, including variables that keep track of method contexts (for statement events) and statement contexts (for variable events), and the global time counter.

The trace hierarchy realizes the aforementioned *inclusion* relations in the traces in that (1) statement traces are nested within the traces of their enclosing method events (preceded by the entrance and succeeded by the exit), and (2) variable traces are nested within the traces of their hosting statement events (preceded by the coverage event of the hosting statement and succeeded by that of the immediate next statement). A hierarchical global mapping then enables the hierarchical indexing of the traces: (1) all statement traces nested within the events of a method are indexed by that method; (2) all variable traces nested within the coverage events of two consecutively executed statements are indexed by the first statement.

The major benefit of the hierarchical trace storage and indexing lies in the quick retrieval of those inclusion relations to be leveraged by trace analysis for efficient dependence querying and other relevant applications. To save space costs, traces are compressed, using the legacy `GZipIOStream` APIs in the Java SDK, before serialized to disk files.

V. Trace Analysis

The first step in the trace-analysis phase is to load (and decompress) both traces and the auxiliary static data. By default, ICDG is loaded to support querying of both data and control dependencies from the traces, as are the method and statement indexes. At the core of this phase is the dependence querying interface built as part of the tracing framework. Through the dependence querying subroutines of this layer, client analyses can focus on application needs without dealing with low-level details about the execution traces.

The built-in querying interface focuses on querying data, and control if opted, dependencies. Specifically, the present interface includes subroutines for querying: (1) method activations (instances of entrance-exit event pairs) of a method and all instances of a statement, (2) all uses and definitions of a variable, at a statement instance, or within a method activation, (3) the last definition of a use and the nearest use of a definition, (4) the last control dependence of a statement instance, and (5) all uses of a definition and all definitions of a use. Other helper subroutines used for looking up method-, statement-, and variable-level dynamic information are also included. Based on the hierarchical trace structure and indexing created in the runtime, these and extended subroutines are, or can be, developed with great ease, mostly just traversing along the trace hierarchy to look up and collect, or even directly access, needed information. Figure 7 shows a screenshot of running some of these subroutines on an example program.

VI. Applications

On top of the dependence querying layer of the core tracing framework, various applications can be built. We describe here two major examples of such applications contained in the present TracerJD toolkit.

A. Dynamic Slicing

Dynamic slicing is a typical form of fine-grained dynamic dependence analysis. Particularly, we developed a *backward* dynamic slicer (closest to the NPwoC algorithm in [2]) on the basis of TracerJD. Given a statement s, this client analysis computes precise backward dynamic slices of all variables used at s for all instances of s, and then gives the union of all such slices as the eventual slice of s. Alternatively, dynamic slice for a specific variable with respect to a particular statement instance can be directly obtained as well. The implementation was straightforward when using the dependence analysis subroutines offered by TracerJD for querying last definitions and last control dependencies.

Our dynamic slicer provides an alternative to existing such tools for Java [16], [17], yet is not subjected to platform constraints as the one in [16]. More important, as we mentioned earlier, existing similar tools are usually designed for particular tasks only, thus they require considerable amount of efforts to be applied to other related applications. For example, revising the backward dynamic slicer JavaSlicer [17] to compute *forward* dynamic slices would need redo more than half of its current implementation (as we confirmed with its lead developer). In contrast, implementing a forward dynamic slicer on TracerJD becomes much easier if using the trace-analysis interface for querying dependencies forwardly. Following this way, we are developing such a forward slicer.

B. Performance Profiling

Beyond dependence analysis, TracerJD can also be employed for other types of tasks given slight extensions. By recording the time elapsed between two consecutive statement events, we have built a statement-instance-level performance profiler that measures the execution time of specified statements. This was easily done by just adding a couple more computation steps in the statement-event monitor of TracerJD, where the runtime tracing component offers an option for logging such additional data. Then, in the trace-analysis phase, performance statistics are computed from the per-statement-instance execution time. Some other statement-level, or method-level, profiling tool can be readily created too. For example, a statement coverage reporter can just collect all executed statements and then compute the coverage using the auxiliary statement index generated in the static-analysis phase (to get the total statements). In fact, this reporter has already been included in the TracerJD toolkit.

VII. Empirical Study

We tested our toolkit on four Java programs with test suites obtained from SIR [22], and used the first version of each, on a Linux workstation with a Quad-core Intel Core i5-2400 3.10GHz processor and 8GB DDR2 RAM. Table I summarizes the characteristics of these subjects in its left four columns, including the number of non-comment non-blank lines of code (*LOC*) and number of inputs (*Tests*).

TABLE I: Experimental Subjects and Slice Results

Subject	Description	LOC	Tests	#Slices	Mean slice size	Mean slicing time
Schedule1	priority scheduler	290	2,650	10	45%	0.8s
NanoXML	XML parser	3,521	214	10	8%	0.3s
XML-security	Encryption library	22,361	92	10	3%	8.1s
JMeter	Performance gauge	35,547	79	10	2%	10s

A. Efficiency

For each subject, we measured the time and space costs of each TracerJD phase separately. Figure 2 shows that TracerJD costs about 5 minutes at most for the studied subjects, with expectedly higher costs incurred with larger subjects. Also, the space costs by the auxiliary static data produced during static analysis are quite small, no more than 2.1MB for any subject, as shown in Figure 3. As the space cost of the instrumentation step, the bytecode size growth depicted in Figure 4 shows that mostly the instrumented programs are about two to three times larger than the original ones.

The runtime performance was gauged through the execution slowdowns of the instrumented programs. As can be seen in Figure 5, the execution tracing of TracerJD causes a slowdown rate of 10 to 13 in most cases (versus a factor of 19 slowdown reported in [10] which performs online dynamic dependence analysis). The space costs by execution traces are generally small, according to the trace sizes shown in Figure 6. The largest trace size was seen by Schedule1, which has far larger number of inputs utilized than the other three subjects.

As an example of the efficiency of client analyses, the last two columns of Table I show the performance of the backward dynamic slicer built atop TracerJD, for 10 randomly selected inputs and slicing criteria for each subject. The slicing time means (last column) show that the dynamic slices were computed efficiently, with mostly larger subjects incurring longer time. The average slice sizes (the fifth column), expressed as the percentage of the number of statements in slices over the total statements in the subject, show that the dynamic slices computed were not trivial.

In all, our empirical results, at least for the four non-trivial subjects of different types and sizes, show that the performance of TracerJD and its application analyses seems to be reasonable for practical uses, although we cannot generalize this conclusion based on results of this small-scale study.

B. Limitations

While the results above are promising, the subjects and inputs used here may not be representative of all those in practice. Since the framework performs whole-program instrumentation and fine-grained tracing, for programs with inputs that lead to very long executions, TracerJD can face efficiency challenges, especially in the runtime phase and probably in trace-analysis phase as well. Another limitation of TracerJD is that its current implementation does not fully support concurrent (e.g., multi-threaded) program executions: Each thread or process needs be traced separately, and dependencies among threads or processes may not be accurately retrieved from concurrent-execution traces.

VIII. Conclusion

We presented a generic tracing toolkit TracerJD devoted to fine-grained dynamic dependence analysis, which is a fundamental technique used in many software analysis and evolution tasks. The entire toolkit comes with several application tools that can be readily used and that demonstrate how client analyses can be flexibly built on the core tracing framework, while the framework itself can be utilized to develop a variety of applications, especially dependence-based ones. Our empirical results show that TracerJD incurs reasonable overheads for tracing and client analyses, thus it can be practically useful. The immediate next step for TracerJD is to build its full support for concurrent program executions. We also plan to optimize the core tracing framework to deal with large and long-running programs more efficiently. Finally, it would be also of interest to extend the framework for partial tracing—instrumenting to trace selective parts of the program or execution segments. TracerJD is open source, so it can be used for research purposes as well.

IX. Acknowledgments

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

References

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

[2] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *ICSE*, 2003, pp. 319–329.

[3] X. Zhang and R. Gupta, "Cost effective dynamic program slicing," in *PLDI*, 2004, pp. 94–106.

[4] ——, "Whole execution traces and their applications," *TACO*, vol. 2, no. 3, pp. 301–334, 2005.

[5] S. Tallam and R. Gupta, "Unified control flow and data dependence traces," *TACO*, vol. 4, no. 3, p. 19, 2007.

[6] "The DTrace Dynamic Tracing Framework," https://docs.oracle.com/cd/E23824_01/html/E22973/gkurw.html.

[7] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *ENTCS*, vol. 89, no. 2, pp. 44–66, 2003.

[8] M. Wang, Z. Li, F. Li, X. Feng, S. Bagchi, and Y.-H. Lu, "Dependence-based multi-level tracing and replay for wireless sensor networks debugging," in *LCTES*, 2011, pp. 91–100.

[9] S. Tallam, C. Tian, R. Gupta, and X. Zhang, "Enabling tracing of long-running multithreaded programs via dynamic execution reduction," in *ISSTA*, 2007, pp. 207–218.

[10] V. Nagarajan, D. Jeffrey, R. Gupta, and N. Gupta, "A system for debugging via online tracing and dynamic slicing," *Software: Practice and Experience*, vol. 42, no. 8, pp. 995–1014, 2012.

[11] S. Bhansali, W.-K. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau, "Framework for instruction-level tracing and analysis of program executions," in *VEE*, 2006, pp. 154–163.

[12] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve, "Using likely invariants for automated software fault localization," *ASPLOS*, pp. 139–152, 2013.

[13] "CodeInvestigator," http://sourceforge.net/projects/cde-investigatr/.

[14] "JTracer," http://www.onyem.com/.

[15] "JavaTracer," https://github.com/samfcmc/JavaTracer.

[16] T. Wang and A. Roychoudhury, "Using compressed bytecode traces for slicing Java programs," in *ICSE*, 2004, pp. 512–521.

[17] "JavaSlicer," https://www.st.cs.uni-saarland.de/javaslicer/.

[18] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "Soot - a Java Bytecode Optimization Framework," in *Cetus Users and Compiler Infrastructure Workshop*, 2011, pp. 1–11.

[19] R. Santelices, Y. Zhang, H. Cai, and S. Jiang, "DUA-Forensics: a fine-grained dependence analysis and instrumentation framework based on Soot," in *SIGPLAN SOAP Workshop*, 2013, pp. 13–18.

[20] H. Cai and R. Santelices, "Diver: Precise dynamic impact analysis using dependence-based trace pruning," in *ASE*, 2014, pp. 343–348.

[21] S. Sinha, M. J. Harrold, and G. Rothermel, "Interprocedural control dependence," *TOSEM*, vol. 10, no. 2, 2001.

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

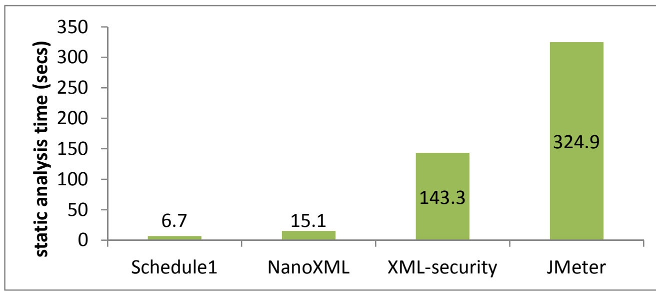

Fig. 2: Static analysis time costs of TRACERJD for the four subjects listed on the x-axis, where the y-axis indicates the computation time in seconds of the entire static-analysis phase. For example, the static analysis took 15.1 seconds on NanoXML. The data shows that the static analysis time that TRACERJD incurs tends to grow with the subject size, yet the largest such cost is about 5 minutes only. Note that for one program version, this cost needs be incurred only once for all possible runtime inputs and dependence queries subsequently.

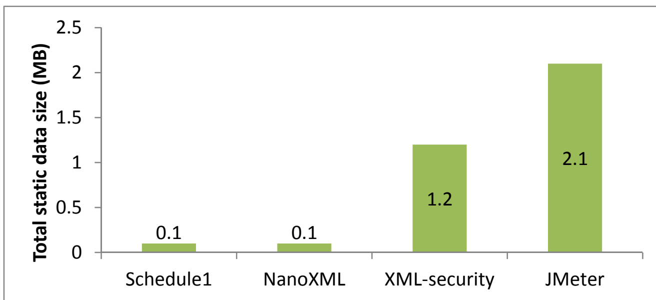

Fig. 3: Sizes of auxiliary static data generated by TRACERJD during the static-analysis phase for the four subjects shown on the x-axis, where the y-axis represents the total static data size in MB. For example, the auxiliary data generated for XML-security is 1.2MB, including the ICDG, statement index, and variable index. The result suggests that TRACERJD tends to incur higher space costs for larger programs, similar to the trend in the static analysis time costs above. Yet, the costs here are almost all negligible for today's storage resources.

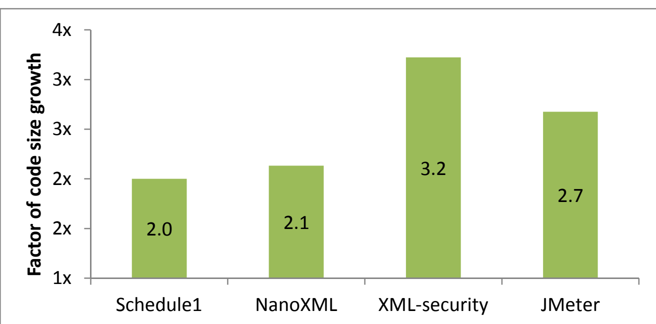

Fig. 4: Bytecode size growth caused by TRACERJD during the static-analysis phase for the four subjects shown on the x-axis, where the y-axis shows the factor of growth per subject before and after instrumentation. For example, the bytecode size of XML-security grows to 3.2 times the original program bytecode size after TRACERJD instrumentation. As expected, mostly the larger programs came with higher such growth.

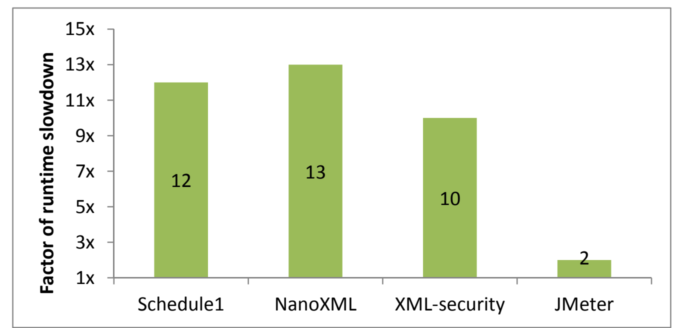

Fig. 5: Runtime overheads of TRACERJD for the four subjects shown on the x-axis, where the y-axis represents the factors of slowdown of the instrumented programs compared to the original ones. For example, the execution time of the instrumented code is 12 times that of the original for Schedule1. JMeter had a very small slowdown possibly due to its test inputs leading to short executions (i.e., having relatively low coverage), which can also be seen from its small trace size compared to the other three subjects (see Figure 6).

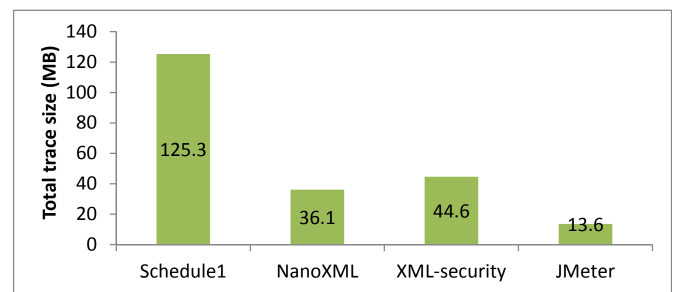

Fig. 6: Sizes of trace data produced during the runtime-tracing phase of TRACERJD for the four subjects listed on the x-axis, where the y-axis represents the total trace sizes in MB. For example, the size of all execution traces generated for NanoXML (for its totally 214 inputs) is 36.1MB.

```
 ###### Testing result from input no. 89
=========== Method Activation Querying Test ==========
All activations of method id=2, name=<ScheduleClass: void _main(ja
         enter at ts=3 and exit at ts=328

=========== Last Definition Querying Test ==========
last def of var varid=-73,vname=$r7,ts=635840 at 102^1 :
        varid=73,vname=$r7,ts=635839 at 101 ts=7

=========== All Uses of Definition Querying Test ==========
uses of the def of var varid=76,vname=r4,ts=635847 at 105^1 :
        varid=-76,vname=r4&AP%1@scanFloat,ts=637131 at 154 ts=695
        varid=-76,vname=r4,ts=637135 at 155 ts=696
        varid=-76,vname=r4&AP%1@scanFloat,ts=637824 at 154 ts=1023
        varid=-76,vname=r4,ts=637828 at 155 ts=1024
        varid=-76,vname=r4&AP%1@scanFloat,ts=637981 at 159 ts=1096
        varid=-76,vname=r4,ts=637999 at 168 ts=1102

=========== Last CD Querying Test ==========
last cd of 183^1:
         statement 181 ts=689

=========== All CD Targets Querying Test ==========
CD targets of 173^1:
         statement 174 ts=639
```

Fig. 7: A screenshot of running sample dynamic dependence querying subroutines on an example program (Schedule1) with respect to its one input, including subroutines querying method activations, the last definition of a use, all uses of a definition, the last control dependence of a statement instance, and all control dependents of a statement instance.