Towards Scalable Provenance Generation From Points-To Information: An Initial Experiment

Padmanabhan Krishnan   Štěpán Šindelář   Bernhard Scholz [*]   Raghavendra Kagalavadi Ramesh   Yi Lu

Oracle Labs, Brisbane

{paddy.krishnan,stepan.sindelar,raghavendra.kr, yi.x.lu}@oracle.com

Abstract

Points-to analysis is often used to identify potential defects in code. The usual points-to analysis does not store the justification for the presence of a specific value in the points-to relation. But for points-to analysis to meet the needs of the programmer, the analysis needs to provide the justification for its results. Programmers will use such justification to identify the cause of defect the code.

In this paper we describe an approach to generate provenance information in the context of points-to analysis. Our solution is to define an abstract notion of data-flow traces that is computed as a post-analysis using points-to information that has already been computed. We implemented our approach in conjunction with the DOOP framework that computes points-to information. We use four benchmarks derived from two versions of the JDK, and use two realistic clients to demonstrate the effectiveness of our solution. For instance, we show that the overhead to compute these data-flow traces is only 25% when compared to the time to compute the original points-to analysis. We also discuss some of the limitations of approach especially in generating precise traces.

1. Motivation

Points-to analysis [SB15] is often motivated by compiler optimisations (e.g., identifying monomorphic calls) but can also be used for finding defects in programs [LL05]. For practical adoption of such analyses, the points-to information must help developers identify the causes of the defect and improve their productivity. The analysis can, for instance, generate an abstract program execution (or a path in the control-flow graph) which can be integrated with the developer's IDE [JSMHB13, CB16]. For example, results from a taint analysis must display paths that show how the objects at the taint sources reach the taint sinks.

Because of the high cost, such path information is not stored by any existing points-to analysis. But it is not essential to store the path for all values. The user is interested only in path information for some specific objects and/or program locations, i.e., for values/constructs related to the defect that the user wants to analyse. For such cases, we want path information for values that satisfy a given property. However, it is not easy to determine the values that satisfy the property while computing the points-to information. One approach is to use staged analysis and slicing [ASK15] where the results of a points-to analysis are used by a client who identifies relevant queries. One can compute the slice and analyse it to keep track of paths. But such slices are still too large for automated path generation.

In this paper we outline the challenges faced in generating traces, and our initial approach that partially solves this problem. We explain this in the context of DOOP [BS09], which uses Datalog to express the points-to computation for Java[1] programs. We present the results of our experimentation with large codebases to illustrate the restrictions on precision imposed by scalability. Although we explain our approach in the context of Datalog, the problem of path generation (or provenance of points-to tuples) in static program analysis is not restricted to Datalog. Any flow-insensitive data-flow analysis that does not keep track of the cause of the data-flow would need modification to generate the provenance information.

Datalog engines such as LogicBlox [GAK12] and Soufflé [SJSW16] use the semi-naïve algorithm [AHV95] that is bottom-up, i.e., compute the output from the given facts till a fixed point is reached. By default these engines do not keep

[*] Current Affiliation/Contact:School of Information Technologies, University of Sydney, bernhard.scholz@sydney.edu.au

[1] Java, JDK and JRE are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

SOAP'17, .
Copyright © 2017 ACM . . . $15.00.
http://dx.doi.org/10.1145/nnnn.nnnn

track of the reasons (i.e.,provenance) for inserting a particular tuple in a relation. There are two possible approaches to retain the provenance during computation. The first is to provide direct support inside the Datalog engine by adapting the semi-naïve algorithm to store the provenance. While it is possible to store provenance for all values (e.g., by storing the delta sets), there is no easy way to store provenance for only relevant values. The second approach is to change the semi-naïve algorithm to a top-down algorithm. However, top-down algorithms are, in general, sub-optimal [AHV95] and it is not clear if such sub-optimal algorithms will scale to the codebases we analyse. In this paper we describe an approach where we develop a special Datalog specification that computes only the provenance for the given queries over the original Datalog specification. So our provenance generation is a special form of demand-driven program analysis [SGSB05].

In the next section we define the problem we solve. Our solution is described in detail in Section 3. We have implemented our solution and experimented with its efficacy on various codebases for different clients. This is described in Section 4. The data presented also illustrates some of the limitations of reusing the points-to information. We present a survey of related work in Section 5 and conclude with a summary.

2. Problem Definition

Our aim is to develop a technique to generate provenance for a class of Datalog points-to specifications:

using standard Datalog semantics so that no alteration to any Datalog engine is required;

that is client driven (i.e., provenance is generated only for tuples satisfying certain user specified conditions) and

that reuses the results of the relations that determine the client's query.

The technique must be able to handle large codebases and must be efficient when compared to computing the original results.

More formally, given an original Datalog specification D_o that computes points-to information, the aim is to write a specification D_p such that D_o composed with D_p and a query Q on relations in D_o can generate provenance of the tuples identified by Q from the results generated by executing D_o.

This technique must also be able to handle large codebases where the relations have millions of tuples. Assume that one is given a recursive relation R of large cardinality defined in D_o and a set P of user selected tuples (derived using Q) which is a small subset of R. The overheads for provenance generation for P from R must ideally be less than 100% (i.e., consume less time and memory than consumed by the actual computation of R).

From a software engineering perspective we want the specification to compute the provenance D_p to be fixed for a large class of original specifications. Otherwise, one would need a custom solution for every input Datalog program. If we can fix D_p for a given an application domain, then every D_o in that domain can use D_p to generate the provenance information.

We focus on a flow-insensitive, but object-sensitive points-to relation [SBL11]. This requires that all client queries must be expressible in terms of the points-to relation, and our technique should be able to generate provenance information only for the selected tuples. In the context of usability, generating the provenance information is not enough. Points-to analysis is often used to detect defects in programs [LL05]. Hence for every defect reported, the developer wants a program trace to identify the causes of the defect. We also need to be able to display the trace in an appropriate tool.

3. Our Approach

In this section we focus on the high-level details of generating provenance for the points-to relation. The first step is to execute the standard points-to analysis and obtain the results that can be used to define a suitable query. The user-specified query, along with the specification of the provenance, is then executed to get the results that are related to the provenance of the query. Note that there is a dependency between the original program (which we had referred to as D_o) and the provenance specification (which we had referred to as D_p). Once the provenance information is generated, the user can specify which aspects need displaying and how it needs to be displayed. Currently we support the open GraphViz format (`www.graphviz.org`) and a format suitable for the Parfait tool [CKL$^+$12] which is used by many developers in Oracle. This is computed outside of Datalog using various scripts. There are two reasons for using scripts and not Datalog for displaying traces. The first is that the provenance results produced by the Datalog engine is in relational form and difficult to interpret at the application level. In other words, the output from the Datalog computation is the set of edges in the provenance graph. The second is because of the size of the provenance information displaying the entire relation is not very useful. Hence the user can decide the necessary aspects and only this information is displayed. Particular examples of this are discussed in Section 4 where we also describe our current implementation.

As we aim to generate abstract traces from flow-insensitive points-to information, the first step is to impose an execution order based on the call-graph edges which are used to develop the traces. This idea is further developed below.

Data-flow Traces

We define our desired structure of traces or the form of the provenance information. The provenance we want to generate for the points-to relation (denoted as *VarPointsTo*) comes from the observation that $(var, alloc) \in VarPointsTo$ holds iff:

1. there is an allocation assigning *alloc* directly to *var*, or
2. given $(var', alloc) \in VarPointsTo$ at least one of the following holds:
   (a) there is a local assignment $var = var'$, where local assignments are those that occur within a method body.
   (b) there is an interprocedural assignment (from var' to var) either from actual parameter of an invocation to the formal parameter of the method, or a return variable of the method to the return value at an invocation.
   (c) there is a load from static field f to var and there is a store to static field f from var'.
   (d) there is a load from instance field f' with base variable b_l to var and there is a store to instance field f' with base variable b_s to var' such that there is an object b_o pointed-to by both b_l and b_s.

The provenance for $(var, alloc) \in VarPointsTo$ has two components. The first is a sequence of variables through which *alloc* 'flows' to *var* starting from the local variable that gets assigned *alloc* directly. The second is related to flow via loading and storing of fields using alias information. Thus our provenance computation generates a graph where variables and fields are vertices and each operation (assignment, load, store) is an edge. Our Datalog specification (D_p) has two relations to represent this information (the first is the transitive closure of rules 2a and 2b, and the second is the combination of rules 2c and 2d). Owing to space limitations we do not show the Datalog encoding here.

Three abstractions are required to the definition of the trace that enable us to scale our computation to large codebases. They relate to both the information we retain as part of the traces as well as the information we use from the points-to analysis. The first abstraction is to show the trace that includes only interprocedural assignments. This means that the local assignments, although used in the computation of the trace, are not explicitly represented in the final result. The second abstraction is related to the storing of contexts in the provenance. We do not store context information in the provenance because the presence of contexts increases the number of tuples in the provenance. The contexts are useful only during the computation phase and the traces themselves do not have contexts. The third abstraction is related to trimming methods that are deemed to not add value to the trace being generated. That is, not all methods involved in the call chain in the provenance information add value. We elide methods that do not have any side-effect (i.e., change the heap) with respect to the traced object and only return values that were passed in as parameters. Erasing such methods from the provenance reduces the size of the relation.

Before we describe our implementation of the technique we present a simple example to illustrate the different

```
1  public class Test {
2    public static class Value {}
3    public static class Holder { public Value v; }
4
5    public static void entryPoint() {
6      Holder h = new Holder();
7      setup(h);
8      read(h);
9    }
10
11    private static Value factory() {return new Value()
        ;}
12
13   private static void setup(Holder h) {h.v = factory
        ();}
14
15   private static void read(Holder h) {
16     Value x = forward(h.v);
17     target(x);
18  }
19
20    private static void target(Value v) { ... }
21
22   private static Value forward(Value v) { return v; }
23 }
```

Figure 1. Java source code for tracing: An illustrative example.

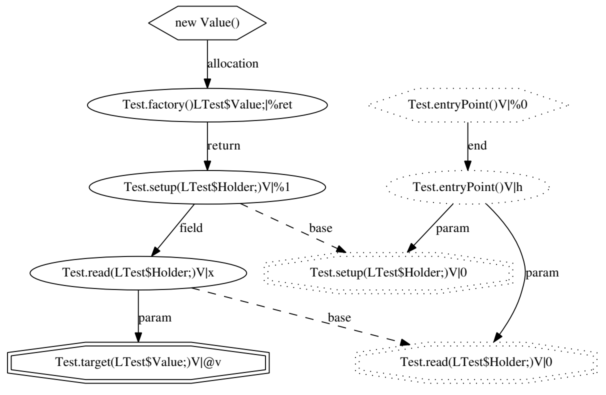
new Value()
allocation
Test.factory()LTest$Value;|%ret
Test.entryPoint()V|%0
return
end
Test.setup(LTest$Holder;)V|%1
Test.entryPoint()V|h
field
base
param
Test.read(LTest$Holder;)V|x
Test.setup(LTest$Holder;)V|0
param
param
base
Test.target(LTest$Value;)V|@v
Test.read(LTest$Holder;)V|0

Figure 2. Derived data-flow traces graph for program in Figure 1

types of edges computed. Figure 1 shows the Java source code. The query is the parameter variable `v` in the method `target` (shown on line 20) may point-to the allocation site in method `factory` which creates an object of type `Value` (shown on line 11). Figure 2 shows the computed data-flow traces graph. The method `factory` is invoked from `setup` which is invoked from `entryPoint`. The first allocation in `entryPoint` creates the base variable `h` whose field is then updated by `setup`. This effectively sets up the field value `h.v` shown by the dotted elements in the trace-graph. The method `target` is called from `read` which calls the method `forward` to set the value of `x`. As `forward` is a side-effect free method, it is not shown in the trace, and the value of `x` is directly copied from the load, i.e., `h.v`. This requires the linking of the base variable which is a parameter h to the object created in entryPoint. The dashed lines (labelled base) capture this behaviour. So there are two aspects to the trace: one captured by direct flow edges and shown via elements that are framed with a solid line in Figure 2; and the other captured by edges related to aliasing for load/store pairs and shown via elements that are framed with dotted lines. The link between the two elements is shown via dotted lines.

4. Implementation and Results

Our implementation to compute the data-flow traces graph in Datalog extends the DOOP framework. The client query is also specified as a Datalog relation that is used in the provenance computation. The results of the Datalog specification execution are stored in a SQLite3 [Owe06] database, which is useful to implement the post-processing algorithms. These include generating graphical information that can be processed by GraphViz to generate visualisation of the data, as in Figure 2, or that can used by Parfait [CKL+12]. We currently support limited features to generate specific data-flow traces, including *all* acyclic paths and sets of shortest paths. This is under the control of the user although for large codebases we do not recommend generating all acyclic paths.

To simplify the presentation of the traces we consider only one level for load/store related aliases. That is, we ignore any load/store pairs that are involved in the aliasing of other load/store pairs. Later we show that computing traces with two levels of alias pairs is possible but it does not yield useful results.

We now present the results of using our approach on large codebases. The performance is dependent on both the underlying codebase and the client queries that need provenance. For the codebases we select four benchmarks composed of two subsets of two versions of the JDK, and for each of these codebases we run the queries of two clients. We refer to the four codebases as S_iV_j for subset i from version j. The first client is taint analysis for certain security-sensitive locations of the program while the second client is escape analysis of objects created at certain security-sensitive locations. We choose these locations based on the descriptions obtained from the Java Secure Coding Guidelines [Jav14].

The taint analysis identifies objects that are created outside the library and reach the undesirable locations (sinks). The escape analysis identifies objects that are created within the library but can reach the application, say, via a return value of a public API or, written to any publicly readable field. For the taint analysis we also use the slicing technique [ASK15] but not for the the escape analysis. Again, this is to study the effect of different client behaviours.

The results we present include metrics to indicate the size of the benchmarks used, the resources consumed for both the 2O+1H context-sensitive points-to computation and the provenance generation, and the characteristics of the graph that represents the provenance.

Table 1. Sizes of input relations for each benchmark used.

Analysed code	Allocation Sites (10^3)	Invocations (10^3)	*VarPointsTo* (10^6)
S_1V_1	19	74	336
S_1V_2	15	54	95
S_2V_1	41	157	871
S_2V_2	46	179	529

Table 2. Size of the queries.

Analysed code	Traces Query	
	Size	Alloc Sites
Taint: S_1V_1	2164	117
Taint: S_1V_2	0	0
Taint: S_2V_1	9346	362
Taint: S_2V_2	12562	443
Escape: S_1V_1	64	2
Escape: S_1V_2	0	0
Escape: S_2V_1	445	16
Escape: S_2V_2	10	10

Table 1 shows the size of the various benchmarks we use in our experiments. It shows the number of allocation sites and invocations in the benchmark program and the size of the computed points-to set. Table 2 shows the result of each query for the various benchmarks. We show the total size of the query relation as well as the number of unique objects (i.e., without contexts) that need tracing as per the query. Note that the query for both taint and escape on the subset S_1V_2 has no objects. This means that there are no defect reports and hence no tracing is required. We included this subset only to ensure that the base level overheads in the tracing process are negligible. Table 3 shows the total computation time taken in minutes for the various cases. We present the time taken to just compute the results, as well as the time taken to compute the results along with generating the trace for the given query. Table 4 shows the memory consumption for the same set of executions.

Table 3. Time (minutes:seconds) for the two clients with and without tracing.

Analysed code	Taint		Escape	
	Only Results	With Tracing	Only Results	With Tracing
S_1V_1	2:26	2:33	34:31	35:12
S_2V_1	12:04	15:09	65:49	67:21
S_2V_2	25:56	32:00	29:51	30:02

Table 5 presents the characteristics of the generated graph. As expected, the graphs are generally sparse (except for the escape results on S_2V_1). However, the maximum degree for the taint traces is quite large. The reasons for these large degrees is related to the imprecision of the points-to

Table 4. Memory consumption in GB (maximum resident size).

Analysed code	Taint		Escape	
	Only Results	With Tracing	Only Results	With Tracing
S_1V_1	17.8	17.9	143	160.9
S_1V_2	33.2	33.2	38.9	38.9
S_2V_1	69	70.5	288.5	314
S_2V_2	140.4	145.2	162.2	176

results. If an object is pointed-to by numerous variables, the trace graph has to follow these variables which leads to an explosion of the graph at that program point. Such behaviours are observed when the level of context-sensitivity is not sufficient for data structures, such as containers. Table 6 presents the size and characteristics of the traces associated with the base variables in load/store pairs. In this case we consider only one level of base-tracing (i.e., we do not follow load/store pairs for load/store pairs etc.). The graph associated with taint on the larger of the two subsets of the two versions of the JDK is significantly larger than the one for the main traces. This is again caused by the imprecision points-to and data structures, such as collections. For the escape analysis, the traces do not involve any fields in the $S_2V_2$and hence no base-tracing is necessary.

Table 5. Characteristics of the trace graph.

Analysed code	Data-flow Traces			
	V	E	Avg(deg)	Max(deg)
Taint: S_1V_1	1830	2636	1.44	15
Taint: S_2V_1	4653	6774	1.46	105
Taint: S_2V_2	4287	8538	1.99	112
Escape: S_1V_1	110	1990	18	32
Escape: S_2V_1	1115	6632	5.95	33
Escape: S_2V_2	13	14	1.08	2

Graph Characteristics and A Pragmatic Solution

By examining the structure of the trace graph, we can see that one cannot sensibly display the entire graph, especially if the degree of a node is very large. Hence in practice we display only the shortest path(s) between a sink and a source. This is generated using the information computed by the Datalog engine and stored in the SQLite3 database.

In Figure 3 we show the distribution of the shortest paths of the taint traces for the S_2V_1 benchmark. Here, most of the paths are of size 20 or less and there are very few paths that are longer than 35. However, visualising paths longer 10 is challenging; and we are examining interactive mechanisms to display and navigate them. The behaviour over other non-trivial examples is similar.

We now examine the cost of some of the implementation level simplifications and abstractions. Recall that our initial implementation of traces related to aliases included only one level of load/store pairs. We have also implemented tracing of depth 2 for aliases. The taint trace generation for depth 2 takes about 10% extra time while the escape trace generation takes about 5% extra time. The extra memory consumption is also less than 5%. However, for the larger subset of the two versions, i.e., S_2V_1 and S_2V_2 the resulting graph is much larger. A spot inspection indicated that the extra information is of very little value. The reason is that the imprecision of the points-to analysis is amplified at the second level, and thus the traces contain many false positives. In summary, although the extra resources required to compute the traces at depth 2 is reasonable, the results are not very useful given our current level of precision of the points-to set.

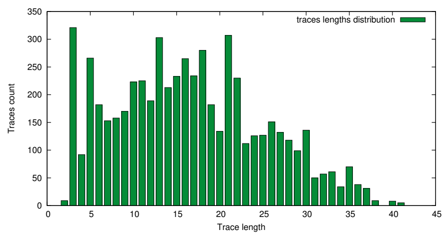

Figure 3. Distribution of shortest paths in S_2V_1.

Summary

Based on our implementation and experiments we can conclude the following. It is feasible to compute abstract traces (involving variables at method boundaries and load/store pairs and not storing context information) for large codebases with overheads of around 25%. Note that our techniques are inexpensive but scalable and thus propagate imprecision in the underlying points-to analysis. From a usability perspective, computing the base traces up to depth 1 is sufficient. However, it is still a challenge to display the entire graph in a useful fashion.

5. Related Work

In this section we review related work for points-to analysis and for provenance in general. Thresher [BCS13] is closest to our work that solves the problem of provenance in the context of points-to analysis. They are interested in the same basic problem in determining precisely if a variable can access an object using information computed by a flow-insensitive points-to information. They use very precise but expensive techniques such as symbolic execution to remove infeasible information flow paths. The main limitation is that their approach is not scalable. The largest program they analyse has 78K lines of code which takes 18 minutes to generate traces from the unannotated program. Another related work is PSE [MSA+04] for C programs, which uses backward-value flow analysis, path-sensitive analysis and symbolic ex-

Table 6. Trace for the base allocations sites of heap loads and stores.

Analysed code	Traces Query		Data-flow Traces			
	Size	Alloc Sites	V	E	Avg(deg)	Max(deg)
Taint: S_1V_1	459	114	2846	2465	1.53	8
Taint: S_2V_1	7343	529	12484	16443	1.8	35
Taint: S_2V_2	25028	531	23791	98245	4.89	136
Escape: S_1V_1	279	12	384	1479	5.23	34
Escape: S_2V_1	1280	43	7588	2818	4.4	37
Escape: S_2V_2	0	0	0	0	0	0

ecution to generates traces. These techniques are expensive and do not scale to our use cases. [GOA05] use of a subset of SQL called PTQL to express the desired set of traces. PTQL specifications are used to instrument a program, and the set of the traces is obtained from a dynamic execution. So our work is more about provenance generation for pre-computed results while their work is about instrumenting programs for dynamic traces.

6. Conclusion and Future Work

In this work we have described the need for generating provenance information for values in the points-to relation. We have described our initial solution and experimental results to show the applicability of our approach.

However, many challenges remain, including improving the precision of the provenance generation (say, via a selective flow-sensitive analysis [BCS13]), provenance for call-graph edges (e.g., the reason for resolving a virtual call to a particular target) and reducing the overheads for generating provenance. Potential other approaches include using languages such as PQL [MLL05] to generate true traces. Displaying the normal traces in combination with the traces for the aliases and integrating the generation and display of traces into the developer's workflow that is required for real adoption [JSMHB13] remains a challenge. These challenges limit the actual adoption of points-to analysis.

References

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[ASK15] N. Allen, B. Scholz, and P. Krishnan. Staged points-to analysis for large code bases. In *Compiler Construction*, LNCS 9031, pages 131–150, 2015.

[BCS13] S. Blackshear, B-Y. E. Chang, and M. Sridharan. Thresher: Precise refutations for heap reachability. In *PLDI*, pages 275–286. ACM, 2013.

[BS09] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262. ACM, 2009.

[CB16] M. Christakis and C. Bird. What developers want and need from program analysis: An empirical study. In *ASE*, pages 332–343. ACM, 2016.

[CKL$^+$12] C. Cifuentes, N. Keynes, L. Li, N. Hawes, and M. Valdiviezo. Transitioning Parfait into a development tool. *IEEE Security and Privacy*, 10(3):16–23, May/June 2012.

[GAK12] T. J. Green, M. Aref, and G. Karvounarakis. Logicblox, platform and language: A tutorial. In *Datalog in Academia and Industry*, LNCS 7494, pages 1–8. Springer, 2012.

[GOA05] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA*, pages 385–402. ACM, 2005.

[Jav14] Secure coding guidelines for Java SE. `http://www.oracle.com/technetwork/java/seccodeguide-139067.html`, 2014. Document version 5.0, updated 25 September 2014.

[JSMHB13] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why dont software developers use static analysis tools to find bugs? In *ICSE*, pages 672–681. IEEE, 2013.

[LL05] V. B. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, 2005.

[MLL05] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. In *OOPSLA*, pages 365–383, 2005.

[MSA$^+$04] R. Manevich, M. Sridharan, S. Adams, M Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis. In *FSE*, pages 63–72. ACM, 2004.

[Owe06] M. Owens. *The Definitive Guide to SQLite*. Apress, 2006.

[SB15] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[SBL11] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, pages 17–30. ACM, 2011.

[SGSB05] M. Sridharan, D. Gopan, L. Shan, and R. Bodik. Demand-driven points-to analysis for Java. In *OOPSLA*, pages 59–76. ACM, 2005.

[SJSW16] B. Scholz, H. Jordan, P. Subotić, and T. Westmann. On fast large-scale program analysis in Datalog. In *Compiler Construction*, pages 196–206. ACM, 2016.