Flow-Sensitive Points-to Analysis for Java Programs using BDDs

Conference Paper · October 2014

DOI: 10.1109/ICCKE.2014.6993367

2 authors, including:

Abbas Rasoolzadegan
Ferdowsi University Of Mashhad
51 PUBLICATIONS **478** CITATIONS

SEE PROFILE

Flow-Sensitive Points-to Analysis for Java Programs using BDDs

Hamid A. Toussi
Young Researchers and Elites Club, Mashhad Branch
Islamic Azad University, Mashhad, Iran
Email: hamidt@mshdiau.ac.ir

Abbas Rasoolzadegan
Department of Computer Engineering
Ferdowsi University of Mashhad, Mashhad, Iran
Email: rasoolzadegan@um.ac.ir

***Abstract*—Doing a flow-sensitive points-to analysis benefits many program analyses which need precise results, however, many prefer to do a flow-insensitive analysis to gain speed and overcome the memory limitations of a flow-sensitive points-to analysis. We are able to overcome these limitations by representing and manipulating points-to sets more efficiently. Binary Decision Diagrams (BDDs) have been shown to be a very efficient representation of points-to sets. A reasonable formulation of the solution is of great importance to achieve this efficiency. In this work, we formulate and employ BDDs to represent points-to sets in flow-sensitive points-to analysis for Java programs.**
Our method was compared with default points-to set in Soot program analysis framework (hybrid points-to set). The results are very promising and show the effectiveness of our method for sufficiently large programs.

Keywords-Software Engineering, Program Analysis, Points-to Analysis, Binary Decision Diagrams

I. INTRODUCTION

Points-to analysis is the problem of approximating runtime values of pointers statically or at compile time. Results of this analysis are used during compilation (e.g. for doing optimizations) or later, in various clients including program understanding, bug-detection, automatic parallelization, etc [1], [2], [3].

Having precise points-to results is very important as it enables points-to analysis clients to produce more useful (accurate) results. In particular, it greatly benefits clients which need precise points-to information such as bug detection, hardware synthesis and analysis of multi-threaded programs [4], [5]. Different factors are involved in the precision of points-to analysis including flow-sensitivity and context-sensitivity [6].

In a context-insensitive analysis, irrelevant information might propagate from one callsite to another through a common callee where in a context-sensitive analysis, this is prevented by using techniques such as cloning the callee at every callsite [7] or excluding unrealizable paths in the analysis [8],[9]. Another method to achieve context and flow sensitivity is by using value flow graph (VFG) [10]. A flow-sensitive analysis respects the order of instructions of the input program while a flow-insensitive analysis simplifies the analysis by assuming that instructions may run in any order. Because of the high performance cost of flow-sensitivity, it is a common practice to do a flow-insensitive points-to analysis. However, flow-insensitivity decreases the precision of the analysis results significantly, and thus, makes the clients less accurate.

BDDs have been used extensively in flow-insensitive points-to analysis [11], [7], [12]. One of the major reasons which contributes to the high cost of doing flow-sensitive points-to analysis is having a separate points-to graph for every program point. Previous works address this problem by using BDDs and sparse control flow representation (def-use chains) [13], [14]. If formulated reasonably, BDDs are able to represent sets and relations very compactly. Also, BDDs as implemented in BuDDy [15] share BDD nodes very effectively which leads to sharing among points-to graphs in our implementation. Flow-sensitive points-to analysis with BDDs has been focused on C/C++ programs or has not addressed Java representation specifically [13], [14], [4].
A related line of work is flow-sensitive points-to analysis by using access paths. Arnab De et al. is able to do strong updates on heap objects in Java program by exploiting access paths [24].

In this work, we formulate and implement a BDD based flow-sensitive points-to analysis for Java programs. We compared our work to hybrid points-to sets which is the default points-to set in Soot. We used Soot program analysis and optimization framework [16], [17] to implement our analysis. BDDs eliminate redundancies in the representation of points-to sets, and also, benefits from common element among points-to sets (because of sharing of BDD nodes in BuDDy). It results in analysis which is both more memory efficient and faster.

In Section II essential information which is needed for understanding our proposed method is presented. In Section III we describe proposed formulation and implementation of our analysis. Our experimental results come in Section IV, and finally, we conclude in Section V.

II. BACKGROUND

In this section, major information which is essential in understanding the overall analysis is presented. In particular, BDDs which is the underlying data-structure of our analysis is introduced, and then, we briefly present the traditional flow-sensitive framework which is the basis of our solver. Also, some aspects of Jimple representation (intermediate representation of Java byte-code in Soot) are discussed. These are the aspects which are very influential to our proposed method.

A. Binary Decision Diagrams

ROBDD (Reduced Ordered Binary Decision Diagram or

BDD for short) is a data structure for representing Boolean functions compactly [18]. A Boolean function can be interpreted as a set by interpreting inputs which produce 1-output as members of the set.

A completely unreduced BDD is shown in Figure 1 which represents the Boolean function *f = ¬x1 .x2 .x3 + x1 .¬x2 .x3 + x1 .x2 .x3.*

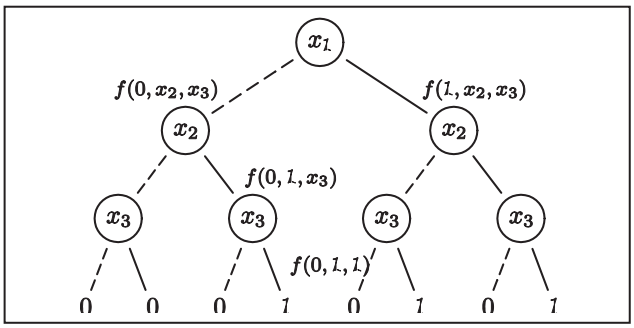

Fig. 1. A completely unreduced BDD which represents the Boolean function *f = ¬x1 .x2 .x3 + x1 .¬x2 .x3 + x1 .x2 .x3.*

ROBDDs eliminate redundancies in full BDDs by keeping them reduced, that is, there are no two sub-graphs in a reduced BDD which are isomorphic, and, low child and high child are different for any node in the reduced BDD. In addition, variables respect a given total order on all paths in a ROBDD.

Figure 2 shows the same Boolean function as in Figure 1, but in reduced form. We can also interpret this BDD and its associated function as a set which contains 011, 101 and 111 strings. A relation can be seen as a set of tuples, so, any relation can be encoded by using BDD as well.

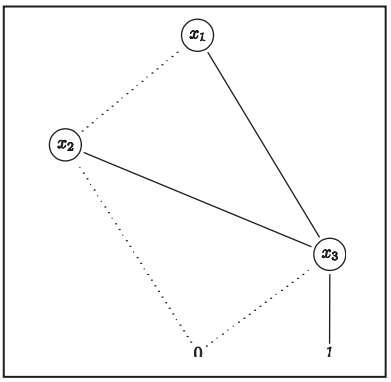

Fig. 2. The reduced form of the BDD previously shown in Figure 1.

BuDDy [15] is a library for creating and manipulating BDDs. It is written in C and also offers a C++ interface. We used this library in our BDD implementation. You can see [11], [7] for more information. Also, in a previous work we described BDDs and BuDDy in more detail [19], [25].

B. Flow-sensitive points-to analysis

Flow-sensitive points-to analysis is based on the traditional data-flow analysis framework. It works on the control-flow graph (CFG) of the input program. For every node in the CFG (program statement), an input points-to graph and an output points-to graph are computed. Points-to graphs represent relations between pointers and the abstract objects they point to.

Based on the instruction of CFG node *k*, a transfer function is assigned to the node. IN_k and OUT_k are input and output points-to graphs of the transfer function respectively. The transfer function follows the pattern below where GEN_k and $KILL_k$ are computed based on the instruction of CFG Node *k*.

$$IN_K = \bigcup_{x \in pred\,(k)} OUT_x$$
$$OUT_k = GEN_k \cup (IN_k - KILL_k)$$

To compute the solution, transfer functions are applied to their inputs iteratively until points-to graphs do not change.

In contrast to a flow-insensitive analysis, a flow-sensitive points-to analysis respects the order of instructions in the input program.

Another advantage which increases the precision of the results, is the removal of some redundant pointer information by the *KILL* set. If left-hand-side (*lhs*) of an assignment refers to a single concrete memory location, the *KILL* set would provide pointer information of the *lhs* in the *IN* set. Such an update is referred to as *strong update*. In cases where the lhs is not guaranteed to refer to a single memory location, the *KILL* set would be empty (*weak update*).

C. Jimple representation

Jimple is the main intermediate Representation in Soot. It has many advantages of SSA representation but it does not have phi-nodes. In particular, it splits variables along DU-UD webs, and as a result, gives a unique name to each definition of a variable to the extent possible (considering that there is no phi-node in Jimple) [20], [21].

This is important for us since we do a flow-sensitive pointer analysis and in this analysis, separate points-to graphs need to be kept for different points in the input program. Jimple representation would enable us to allocate only one points-to set for every variable in Jimple without losing precision for local variables, however, keeping only one points-to set for every global variable (static field in Java programs) make the flow-sensitive analysis less precise as global variables might have different points-to sets at different points of the program. Previously, Ben Hardekopf used a similar method in his analysis which was done on LLVM IR [13], [14].

Among different instructions, we are interested in instructions which may modify pointer values, in particular, allocation where a new object is allocated (new), simple assignments (copy) where a variable (reference) is assigned to another variable (v = w), field-load where a field-dereference is assigned to a variable (v = w.f), field-store (v.f = w), invocation and return instructions.

III. PROPOSED METHOD

We implemented an inter-procedural flow-sensitive points-to analysis which operates on control-flow-graphs. In the first step, nodes which contains instructions, irrelevant to pointers, are eliminated from control-flow-graphs and edges in control-flow-graphs are fixed to maintain the control-flow. Allocation-sites are used as abstract objects in our analysis. In particular, the analysis deals with the following types of instructions:

L: v = new SomeClass() *// Allocation site at L*
v = w *// Copy instruction*
v = w.f *// Load instruction*
v.f = w *// Store instruction*
v = w.f(x, y, z, ...) *// Invoke instruction*
return v *// Return instruction*

All other types of relevant instructions are interpreted as a sequence of instructions above. A flow insensitive points-to analysis (Spark [22]) is done to create a call-graph which is used to resolve call-targets at call-sites during our analysis. On the fly call-graph (creating the call-graph as we do the analysis) is not available in the current implementation and is planned for the future.

Two versions were implemented, a baseline which uses hybrid points-to sets (also known as hybrid bit-vectors), and another which uses BDDs.

There is a points-to graph for Jimple variables. It has a points-to set for every variable (*global points-to graph*). Similar to semi-sparse points-to analysis [13], [14], a single points-to graph is used for keeping variables' points-to sets. This would benefit memory usage as we no longer need to keep a separate set for every variable at every program point. The precision loss is very small as Jimple is a SSA-like representation (See Section II-C).

There is a method worklist containing methods which need to be processed. Also, every method in the input program has a node worklist. This worklist is initialized to contain all the nodes in the control-flow-graph of the method.

Every node in a control-flow-graph (CFG) has one input points-to graph and one output points-to graph. These points-to graphs hold points-to sets for *AllocField* nodes (abstract_object.field: An AllocField corresponds to a field of an abstract object). We refer to these as *AllocField points-to graphs*. Since the global points-to graph is responsible for keeping points-to sets of variables, AllocField points-to graphs do not contain variables' points-to set.

```
void solveWorklist() {
  while (!methodWorklist.isEmpty()) {
    SootMethod method = methodWorklist.pop();
    Set<DFGNode> nodeWorklist = getWorklistOf(method);
    while (!nodeWorklist.isEmpty()) {
      DFGNode dfgnode = nodeWorklist.pop();
      for (DFGNode pred : dfgnode.getPreds()) {
        IN[dfgnode] = IN[dfgnode].meet(OUT[pred]);
      }
      boolean change = false;
      /* The following boolean variable
      can be changed by hadleAlloc, handleCopy,
      handleInvoke and handleReturn routines */
      top_level_points_to_changed = false;
      switch (dfgnode.INSTRUCTION_TYPE) {
        case ALLOC:
         change = handleAlloc(dfgnode);break;
        case COPY:
         change = handleCopy(dfgnode); break;
        case LOAD:
         change = handleLoad(dfgnode); break;
        case STORE:
         change = handleStore(dfgnode);break;
        case INVOKE:
         change = handleInvoke(dfgnode);break;
        case RETURN:
         change = handleReturn(dfgnode);break;
      }
      if (change) /* ifI */
        nodeWorklist.addAll(dfgnode.successors());
      if (top_level_points_to_changed) { /* ifII */
     nodeWorklist.addAll(reachableNodesFrom(dfgnode));}
}}}
```

Fig. 3. Worklist solver

Figure 3 presents a simplified version of the main body of our analysis. It is a worklist algorithm which works on the method worklist and node worklists. Nodes in a node worklist are iterated in topological order in the analysis to speed up the propagation of points-to relationships. It is very similar to a traditional data-flow analysis solver but with a few exceptions.

In a traditional data-flow analysis, we only keep track of changes in data-flow facts (AllocField points-to graphs in our analysis) for every point in the CFG, but, here, we should also keep track of changes in the global points-to graph. This is done by the statement marked as *ifII* in the Figure. Also note that store instructions only affect AllocField points-to graphs and have no effect on the global points-to graph.

In case that the global points-to graph is changed as the result of processing node *n*, it is not enough to only add the successors of *n* to worklist as the successors may not propagate the change further, so all the nodes reachable from *n* are added to the worklist in this case.

We optimized `reachableNodesFrom` routine to speed up the solver: In cases that the *lhs* (left-hand-side of the assignment) is a variable, it is not necessary to add all the nodes reachable from *n* to the worklist, but, a subset of reachable nodes where the variable is being read is enough. In cases that the *lhs* is a field-dereference (v.f) such an optimization is not possible since pointer information which is stored to the *lhs* may modify the

points-to sets of some AllocField, and, this pointer information can be loaded via a variable other than v (i.e v might be aliased with other variables).

Routine `handleInvoke` propagates points-to information from callsite to all the possible trargets (callees) and adds any of the callees which needs to be processed as the result of the propagation, to `methodWorklist`. Routine `handleReturn` propagate points-to information from the returned value to all the callers, and similarly, add any of the callers which need to be processed to `methodWorklist`. Routines `handleAlloc`, `handleCopy`, `handleLoad` and `handleStore` handle corresponding types of instructions. As the efficient formulation and implementation of these routines are essential to the performance of the BDD implementation, they will be discussed in the next section.

Hybrid points-to sets (default points-to set in Soot) are used in the baseline bit-vector implementation (both in the global points-to graph and AllocFiled points-to graphs). We will focus on the BDD formulation and implementations in this paper as the bit-vector implementation is more straightforward to implement. Also, the BDD implementation is far more efficient than the bit-vector version.

A. BDD formulation and implementation

In this section, we describe BDD formulation of our method. The important point is to use operators which work on whole relations rather than individual members (since BDD is a compact representation for a whole relation).

The BDD version is implemented by using BuDDy [15]. The routines were implemented in C++ and then linked to Java (The solver was developed within the Soot program analysis framework, which itself, has been written in Java).

In BuDDy, BDD variables can be grouped into finite domains for easier manipulation. In our implementation of the analysis, there are three variable domains, namely, D, F and R.

An AllocField points-to graph is a relation which contains triples of the form *(AllocationSite, Field, AllocationSite)* and is defined over domain D × F × R. Every triple is a points-to relationship. For example, triple *(o1, f, o2)* shows $o2 \subset pt(o1.f)$. *o1* and *o2* are abstract objects (allocation sites in our implementation and *pt(e)* refers to points-to set of element *e*).

Major data structures which are used in the BDD implementation are:

`points_to`: It is an array of BDDs. Every BDD represents a points-to set of a variable (over domain D). Variable's number can be used to index into this array and retrieve it's pointsto set. It is the global points-to graph. In another word `points_to[v]` corresponds to *pt(v)*.

`field_pts_in`: It is an array of AllocField points-to graphs and represents IN [stmt] for every statement (CFG node) in the control flow graph. Statement's number can be used to index into this array and retrieve its input AllocField points-to graph.

`field_pts_out`: An array which represents OUT[stmt] for every statement (CFG node) in the control flow graph (similar to field_pts_in but for output points-to graphs).

`type_mask`: An array which has a set of abstract objects for every type in the input program. Every set is represented by a BDD and contains all abstract objects with type compatible to the set's associated type. A type can be used to index into this array and get its set (BDD). This array is used for doing type filtering during the analysis (online type-filtering) [23] (see Figures 5 and 7).

Figures 4, 5, 7, 6 show the routines which are called in the solver (Figure 3) to process different kinds of nodes in the CFG. These routine were implemented in C++ by using the BuDDy library, and, then linked to the solver (implemented in Java).

```
int handleAlloc(int var, int alloc, int stmt) {
  int ret = 0;
  // Copy IN[stmt] to OUT[stmt].
  if (field_pts_out[stmt] != field_pts_in[stmt]) {
    field_pts_out[stmt] = field_pts_in[stmt];
    ret = 1;
  }
  bdd old = points_to[var];
  bdd curr = old;
  curr |= fdd_ithvar(Dd, alloc);
  if (curr != old) {
    points_to[var] = curr;
    top_level_points_to_changed = 1;
  }
  return ret;
}
```

Fig. 4. Routine to handle allocation instructions (var = alloc)

Entities such as variables, statements, etc are not directly passed to these routines, instead their integer representatives in Soot have been used. In this way, our C++ module does not have to interact with Java objects directly. Different operators were overloaded in BuDDy. For example & shows an **and** operation and | shows an **or** operation. The **relational product** operation (`bdd_relprod` in BuDDy) is equivalent to an **and** followed by an **existential quantification** but the implementation of this operation in BuDDy does the job faster than the case these operations were run in sequence.

As BDDs represent whole relations, all the routines have been implemented in terms of these binary operations to gain efficiency, in particular, we never iterate through a set or relation in the BDD

implementation. Inspecting every individual member of a BDD set is a very expensive operation since it requires generation of all the reduced paths (implicit nodes) in the BDD.

```
int handleCopy(int x, int y, int x_type, int stmt)
{
  bdd old = points_to[x];
  bdd curr = old;
  bdd filtered = points_to[y] & type_mask[x_type];
  curr |= filtered;
  if (curr != old) {
    points_to[x] = curr;
    top_level_points_to_changed = 1;
  }

  if (field_pts_out[stmt] != field_pts_in[stmt]) {
    field_pts_out[stmt] = field_pts_in[stmt];
    return 1;
  }
  return 0;
}
```

Fig. 5. The routine which handles copy instructions (x = y)

```
int handleStore(int x, int y, int f, int stmt) {
  // Results in a set of AllocField nodes:
  //(D x F) x _
  bdd temp = points_to[x] & fdd_ithvar(Fd, f);
  // Move pt(y) from domain D to domain R:
  // (_ x _) x R
  bdd pt_y_R = bdd_replace(points_to[y], Dd_Rd);
  bdd gen = (temp & pt_y_R); // (D x F) x R
  bdd old = field_pts_out[stmt];
  // gen U IN[stmt]
  field_pts_out[stmt] = (gen | field_pts_in[stmt]);
  if (field_pts_out[stmt] != old) return 1;
  return 0;
}
```

Fig. 6. The routine which handles store instruction (x.f = y)

```
int handleLoad(int x, int y, int f,
int x_type, int stmt) {
  bdd field_pt = field_pts_in[stmt];
  bdd temp = bdd_relprod(points_to[y],
  (field_pt & fdd_ithvar(Fd, f)), set_DF);

  bdd old = points_to[x];
  bdd curr = old;
  bdd filtered = bdd_replace(temp, Rd_Dd)
                 & type_mask[x_type];

  curr |= filtered;
  if (curr != old) {
    points_to[x] = curr;
    top_level_points_to_changed = 1;
  }
  // OUT[stmt] = IN[stmt]
  if (field_pts_out[stmt] != field_pts_in[stmt]) {
    field_pts_out[stmt] = field_pts_in[stmt];
    return 1;
  }
  return 0;
}
```

Fig. 7. Routine which handles load instructions (x = y.f)

The major operation in routine `handleAlloc` is a set union which adds alloc to `points_to[var]`. `fdd_ithvar(D, O)` encodes O over domain D. The **or** operator is used to do the set union operation. Routine `handleCopy` first filter out incompatible abstract object by using an **and** operation, and then, adds the resulted set to `points_to[x]`.

Routine `handleStore` first obtains the gen set, and then, update the output points-to graph (see comments in Figure 6). In routine `handleLoad`, all the points-to relationships in input points-to graph corresponding to AllocFields in y.f (y may point to multiple abstract objects) is obtained first (temp set). The major operation is a relational product in this step. After, filtering out abstract objects with incompatible type, the result is added to `points_to[x]`. In an implementation with regular sets, we would need to iterate over all abstract objects in *pt(y)* to find points-to set of its corresponding AllocField, but, the relational product does this in one step in a far more efficient way.

Another routine which is used in the solver is meet which corresponds to a meet operator in the data-flow analysis framework. In the BDD version of the analysis, it is implemented as **or** Boolean operator.

B. Strong update

As it was explained in Subsection II-B, a strong update can only be done when *lhs* of an assignment refers to a single concrete memory location. In case of field-store instructions, the *lhs* refers to one or many AllocField nodes. Even when the *lhs* refers to a single AllocField node, a strong update is not possible since that one abstract object (allocation site) may correspond to multiple real objects. That explains why the KILL set in routine in Figures 6 is empty.

IV. EVALUATION AND EXPERIMENTAL RESULTS

We implemented two versions of our analysis, BDD based which was implemented based on our BDD formulation, and, bit-vector based which uses hybrid points-to sets (baseline). Hybrid points-to set is the default representation of points-to sets in Soot.

Our analysis has been programmed within Soot program analysis framework. Programs in Table I have been used in our evaluation. These are Soot 1.2.5 (an old version which compiles and run with JRE 1.3), SableCC 2.18.2 (a lexical analyzer generator), an implementation of QuickSort and Btree. QuickSort and BTree programs were obtained from http://introcs.cs.princeton.edu/java/home/ and then, slightly modified to compile with JRE 1.3 (Modified version is available at http://hamid2c.github.io/iccke2014/prince.tar.gz).

Table I shows number of classes, number of methods and number of Jimple instructions for the programs we used in our evaluation. As we do whole program analysis, Soot conservatively loads classes that might be linked with the input program. That explains why there are hundreds of classes even for small programs such as QuickSort.

We used JRE 1.3 as the library to analyze input programs. All the benchmarks were done on a computer with an Intel Pentium processor (G2030) with 4 GB of RAM (2.6 GB was allocated to JVM) running Ubuntu 14.04. We obtained very promising results, in particular, our BDD based solver completed analyzing Soot (which is the largest program) in 28.9 seconds (2 min. 48 sec for the whole running of Soot including phases other than our solver) while the baseline version (bit-vector based) went out of memory after 25 minutes without completing the analysis. You can see the running-times for different programs in Table II. Column bdd shows the running time of the BDD solver and column bitvector shows the running time of the baseline (bit-vector) solver.

As programs get larger, BDD based solver becomes faster compared to the baseline. In case of soot, the speed up is very promising and we believe it will be even bigger for larger programs.

TABLE I. PROGRAMS IN OUR BENCHMARK

program	#classes	#methods	#instructions
soot	2407	19384	208811
sablecc	906	8565	85769
btree	655	6348	55672
quicksort	651	6308	55714

We briefly explain why the solver with BDD is much faster and more memory efficient than the baseline. BDDs capture the similarities between set elements and produce a reduced and compact form while bit-vector based sets allocate one bit for every abstract object in the input program. Also, in points-to analysis, there are many sets with common members. Sharing of BDD nodes in BuDDy benefits from this and greatly improves the memory consumption. Also, note that size of a BDD depends on the coding of every one of its members, and not just number of elements in the BDD set (i.e. different elements in a BDD set correspond to different paths which results in different pattern of reductions). In addition, Our BDD formulation of the analysis takes advantage of efficient BDD operations on whole relations.

TABLE II. RUNNING TIMES OF BDD AND BITVECTOR SOLVERS FOR DIFFERENT PROGRAMS (IN SECONDS)

program	bdd	bitvector
soot	28.9	Unknown (> 1500)
sablecc	2.1	6.0
btree	0.5	0.9
quicksort	0.6	2.1

V. CONCLUSION AND FUTURE WORKS

In this work, we designed and implemented a flow-sensitive points-to analysis for Java programs by using BDDs. To our knowledge, previous BDD based works have been focused on C/C++ programs [13], [14], or, is flow insensitive (e.g. [11]). We formulated a flow-sensitive analysis for Java programs. Also, we used many state of the art techniques to make the analysis faster including elimination of irrelevant nodes from the CFG and a worklist solver. Results of our experiments are very encouraging and show that our proposed method can greatly speed up the analysis of sufficiently large programs.

In the future, we plan to experiment with the context-sensitivity [7], [12]. Also, we like to implement our method with online call-graphs (call-graphs will be created during our flow-sensitive points-to analysis).

Another interesting addition would be making the flow-sensitive points-to analysis staged as it is presented in [14]. In this scenario, a pointer analysis is performed in two or more steps. Every step is a pointer analysis itself. Each step uses the results of the previous step. Different dimensions can be used in every step (regarding flow-sensitivity, context-sensitivity, etc).

REFERENCES

[1] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural aliasing," ACM SIGPLAN Notices, vol. 27, no. 7, pp. 235–248, 1992.

[2] M. Hind and A. Pioli, "Which pointer analysis should i use?" in ACM SIGSOFT Software Engineering Notes, vol. 25, no. 5. ACM, 2000, pp. 113–123.

[3] R. Ghiya and L. Hendren, "Putting pointer analysis to work," in Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1998, pp. 121–133.

[4] J. Zhu, "Towards scalable flow and context sensitive pointer analysis," in Proceedings of the 42nd annual Design Automation Conference. ACM, 2005, pp. 831–836.

[5] A. Salcianu and M. Rinard, "Pointer and escape analysis for multithreaded programs," ACM SIGPLAN Notices, vol. 36, no. 7, pp. 12–23, 2001.

[6] M. Hind, "Pointer analysis: haven't we solved this problem yet?" in Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. ACM, 2001, pp. 54–61.

[7] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in ACM SIGPLAN Notices, vol. 39, no. 6. ACM, 2004, pp. 131–144.

[8] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1995, pp. 49–61.

[9] M. Sridharan and R. Bod´ ık, "Refinement-based context-sensitive pointsto analysis for java," ACM SIGPLAN Notices, vol. 41, no. 6, pp. 387– 400, 2006.

[10] L. Li, C. Cifuentes, and N. Keynes, "Precise and scalable context-sensitive pointer analysis via value flow graph," in ACM SIGPLAN Notices, vol. 48, no. 11. ACM, 2013, pp. 85–96.

[11] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee, "Pointsto analysis using BDDs," in Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. San Diego, California, USA: ACM Press, 2003, pp. 103–114.

[12] O. Lhotak and L. Hendren, "Context-sensitive points-to analysis: is it worth it?" in Compiler Construction. Springer, 2006, pp. 47- 64.

[13] B. Hardekopf and C. Lin, "Semi-sparse flow-sensitive pointer analysis," in ACM SIGPLAN Notices, 2009, pp. 226–238.

[14] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in Code Generation and Optimization (CGO), 9th Annual IEEE/ACM International Symposium on, 2011, pp. 289–298.

[15] J. Lind-Nielsen, "BuDDy library," http://sourceforge.net/projects/buddy/, 2002.

[16] R. Vall ´ ee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in

Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, ser. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999.
[17] P. Lam, E. Bodden, O. Lhotak, and L. Hendren, "The soot framework for java program analysis: a retrospective," in Cetus Users and Compiler Infastructure Workshop (CETUS 2011), 2011.
[18] R. E. Bryant, "Graph Based Algorithm for Boolean function manipulation," in IEEE Transactions on Computers, 1985.
[19] H. A. Toussi and B. S. Bigham, "Design, Implementation of MTBDD based Fuzzy Sets and Binary Fuzzy Relations and its Application in Fuzzy connectedness," in Proceeding of International Conference on Computer, Information Technology and Digital Media, Tehran, Iran, 2013, pp. 197–206.
[20] S. S. Muchnick, Advanced compiler design and implementation. Morgan Kaufmann, 1997.
[21] "How does Jimple preserve semantics with its partial SSA without Phi-node mechanism?" Soot mailing list, July 2012, [Online]. Available: http://www.sable.mcgill.ca/pipermail/soot-list/2012-July/004586.html
[22] O. Lhotak and L. Hendren, "Scaling java points-to analysis using Spark," in Compiler Construction. Springer, 2003, pp. 153–169.
[23] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind, "Fast online pointer analysis," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 29, no. 2, 2007.
[24] A. De and D. Deepak, "Scalable flow-sensitive pointer analysis for Java with strong updates." ECOOP 2012–Object-Oriented Programming. Springer Berlin Heidelberg, 2012. 665-687.
[25] H. A. Toussi and B. S. Bigham, "Design, Implementation and Evaluation of MTBDD based Fuzzy Sets and Binary Fuzzy Relations," preprint arXiv:1403.1279 [cs.DS], [Online]. Available: http://arxiv.org/abs/1403.1279