Alias Analysis with bddbddb

Kevin Bierhoff
kevin.bierhoff [at] cs.cmu.edu

Star Project Report
17-754 Analysis of Software Artifacts
Spring 2005

Abstract

Context-sensitive inclusion-based alias analysis is very precise but also computationally expensive. Whaley and Lam [5] recently proposed an approach based on Binary Decision Diagrams that scales to realistic Java programs. This report summarizes the approach, describes how the tool bddbddb that implements the analysis is used, and discusses how it can be employed to develop derived analyses based on alias information. In particular, a tainted analysis is developed that reveals both strengths and weaknesses of bddbddb as an analysis framework.

1 Introduction

Virtually all software written in imperative and object-oriented languages uses mutable shared data structures. Unexpected sharing can lead to common errors like race conditions or pre-condition violations. The pattern is often that one client accesses data when another client does not expect it. The goal of alias analysis is to understand sharing in a software system.

References to a particular object are called *aliases*. The *context* in which a function is called greatly influences the aliases that function will produce. Exhaustive exploration of all contexts used to be computationally infeasible. Recently, Whaley and Lam proposed a scalable context-sensitive inclusion-based alias analysis for Java [5] based on BDDs [2].

The implementation is publicly available on SourceForge under the name bddbddb. Alias analysis primarily forms the input to derived analyses. This report evaluates the tool and in particular its ability to support derived analyses. A tainted analysis is developed that reveals strengths and limitations of the tool.

The remainder of this report is organized as follows. The research context of alias analysis is summarized in section 2. Section 3 describes the approach taken in bddbddb. Installation and hands-on usage of the tool is summarized in section 4. Section 5 develops the tainted analysis, thereby discussing the usefulness of the tool. The report concludes in section 6.

2 Alias Analysis

This section gives a brief introduction to alias analysis, thereby categorizing existing work on the problem.

An alias analysis determines what references in a program *may* point to which objects. That includes local variables in functions, arguments passed between functions, and fields of objects pointing to other fields. While aliasing is reasonably easy to figure out within a single function, the problem becomes much harder when (possibly recursive) calls between functions are considered. Thus it is foremost an *interprocedural analysis*.

Alias analysis is a long-standing problem because it is computationally expensive. In order to compare previous work with the approach of Whaley and Lam [5], it is useful to categorize approaches to alias analysis along two axes.

- Unification-based vs. inclusion-based. Unification assumes that two references either point to the same set of objects or to completely distinct sets. Inclusion allows sets for different references to overlap.
- Context-insensitive vs. context-sensitive. A context-insensitive analysis treats all calls to a function identically. Context-sensitivity means that functions are analyzed depending on the calling context, which is essentially the call stack. It turns out that context sensitivity is very important to get precise results.

Context-insensitive unification-based approaches [4] easily scale to large programs, but they are also extremely imprecise. Context-insensitive inclusion-based approaches scale somewhat, but are still imprecise. A scalable technique for this approach [1] that is based on BDDs forms an important precursor to Whaley and Lam's work. Context-sensitive unification-based approaches [3] also have serious scalability restrictions and are imprecise due to inclusion. Finally, context-sensitive inclusion-based approaches [6] are more expensive on both axes. They are potentially very precise but until recently could not scale to realistic programs.

3 Context-sensitive Inclusion-based Alias Analysis with BBDs

This section gives a description of the alias analysis that `bddbddb` performs. We first discuss the approach taken, then describe its implementation, and finally report experimental results.

3.1 Database-based Approach

What makes context-sensitive approaches computationally extremely expensive is the exponential growth in the number of contexts for large programs. Inclusion-based approaches are also more expensive than unification-based ones.

Whaley and Lam proposed the first context-sensitive inclusion-based alias analysis that scales to realistic programs. They could get their analysis to work on programs with 10^{14} contexts and more [5]. Context sensitivity can be realized in two ways.

- Summary-based approaches analyze each function in the program only once and create a summary that can be parameterized with information about a particular call site, e.g. information about the arguments to the function. This approach is not so useful for alias analysis because it is hard to create a good aliasing summary for a function (making the analysis less precise).
- Cloning-based approaches analyze each function for each calling context separately. This can be seen as a truly brute-force approach where functions are conceptually cloned to match the call graph of the program one to one. Context-sensitive alias analysis then becomes algorithmically trivial: One can simply run a context-insensitive analysis on the expanded program.

bddbddb implements a cloning-based approach to context sensitivity. It uses a pre-computed call graph and a list of all possible contexts to build a *points-to* relation. The relation contains tuples (c, v, h) which means that variable v can point to object h in context c. (A similar relation is built for fields of objects.)

The underlying idea of this tool is that all information we as analysts have is represented as relations between domains like in a relational database. Domains include variables (V), objects (H), fields (F), methods (M), invocations (I), and types (T). Objects are identified by their creation site. The creation site is a *new* invocation, thus $H \subseteq I$.

The points-to relation is computed from various information about the program to be analyzed, represented as relations including (details in [5]):

- Variable creation: $vP_0 \subseteq V \times H$
- Storing a variable in a field: $S \subseteq V \times F \times V$
- Loading a variable from a field: $L \subseteq V \times F \times V$
- Context-sensitive call graph: $IE_c \subseteq C \times I \times C \times M$. (There is also a context-insensitive version IE.)
- Formal and actual parameters.
- Variable and object typing information.

These relations are computed directly from the program. The points-to relation is derived from them with Prolog-like deduction rules. The underlying Datalog language forms a superset of normal SQL queries to a database. As a simplified example, the following rules will compute the context-sensitive points-to relation vP_c for variables (ignoring fields).

$$
\begin{array}{lll}
vP_c(c, v, h) & :- & vP_0(v, h), IE_c(c, h, \_, \_). \quad (1) \\
vP_c(c_1, v_1, h) & :- & assign_c(c_1, v_1, c_2, v_2), vP_c(c_2, v_2, h). \quad (2) \\
assign_c(c_1, v_1, c_2, v_2) & :- & IE_c(c_2, i, c_1, m), formal(m, z, v_1), actual(i, z, v_2). \quad (3)
\end{array}
$$

Note that formal and actual parameters are enumerated with numbers $z = 0, 1, 2, \ldots$. The points-to relation starts with object creations in a particular context (1). (This works because objects are identified by their creation site, which counts as a method invocation.) It then builds the transitive closure over variable assignments (2). Variable assignments are computed from the call graph by matching formal and actual parameters between a method and its invocation site (3). The reader can convince himself that repeated application of these rules will reach a fix point when the two constructed relations do not change any more.

The actual alias analysis as given in [5] contains three more rules that take care of loads and stores from and to fields. It also uses typing information about variables and objects to exclude impossible assignments, thereby improving precision.

3.2 Implementation with BDDs

The relation vP_c becomes easily very large for realistic programs. Worse, the deduction rules given above have to be applied possibly millions of times, each time enlarging the relation. Thus the key to making the computation of the relation feasible is to efficiently represent, manipulate, and join relations.

bddbddb uses Binary Decision Diagrams (BDDs, [2]) to this end. Hence the name: *BBD-Based Deductive DataBase*. BBDs are an efficient way of representing and in particular manipulating binary functions. They were invented to compactly represent functions whose elements share a lot of commonalities. bddbddb uses ordered BDDs (enforcing a fixed variable ordering in the graph) which have a unique minimal form.

A relation like vP_c can be seen as a binary function in the following way: $vP_c(c, v, h) = 1$ iff $(c, v, h) \in vP_c$. If we assign (binary) numbers to the members of domains participating in a relation then we get a function $\{0, 1\}^n \to \{0, 1\}$. This function can be represented as a BDD.

All operations necessary to apply Datalog rules can be expressed with well-known operations on BBDs for which there exist efficient algorithms. bddbddb uses the BDD implementations BuDDy[1] and JavaBDD[2] to manage BDDs.

It turns out that the relations necessary for alias analysis have the property of having many similar elements. In particular, the points-to relation does depend on the context, but on the other hand the aliasing of similar contexts is often quite similar. Intuitively this means that aliasing in a function is only influenced by the last couple of functions on the call stack.

[1] www.itu.dk/research/buddy
[2] javabdd.sourceforge.net

3.3 Experimental Results

Whaley and Lam implemented their analysis for Java and ran it on the 20 most popular stand-alone Java applications developed on SourceForge[3]. These applications contain up to 145,000 variables and produce up to 4×10^{14} context-sensitive paths. bddbddb is still able to analyze each of these programs in under 20 minutes.

The authors developed a typed refinement analysis (see below) that builds on the alias analysis and can be used to measure precision of the analysis. In particular, it can compute the variables for whom the objects they can alias have multiple types. Whereas this happens with the context-insensitive analysis for an averaged 5 percent of all variables, context-sensitive analysis is able to drop this fraction to averaged 0.4 percent. Detailed results can be found in [5].

This indicates that the gain in precision from the context-sensitive analysis is on the order of a magnitude, thus justifying the increased overhead. At the same time bddbddb proves that such a precise analysis is feasible for realistic programs.

4 Using bddbddb

bddbddb[4] can be freely acquired from SourceForge using CVS. The project consists of several modules, notably the Java implementation itself, sources for an Eclipse plugin, and a number of examples including the alias analysis presented above. The author could not get the Eclipse plugin to run.

The actual implementation module comes complete with three required libraries, one of which is Whaley's BDD implementation JavaBDD. Running bddbddb then simply involves putting all these libraries in the classpath and using `net.sf.bddbddb.Solver` as the main class for the Java VM. The application takes the name of a textfile as an argument which contains the Datalog query to be executed. Note that the normal 64 MB Java VM heap are usually not enough to run the tool.

Datalog files consist of three parts. Examples can be found on the bddbddb website.

- The domains involved in the query.
- The relations involved. They can be marked as "input" or "output", or they can be unmarked which makes them an intermediate relation (like $assign_c$ in the alias query).
- The actual rules comprising the query.

Input relations are read from a file with the name of the relation. Output relations are written to a file with their name. Intermediate relations are

[3] www.sourceforge.net
[4] bddbddb.sourceforge.net

computed but not written. The usual file format for relations is a text-based representation of the BDD which is incomprehensible to the author. However, by using "outputtuples" instead of "output", a relation will be written to a text file with one tuple per line. ("inputtuples" does the obvious analogue.) A blank is used to separate the fields of the tuples. The first line in the file indicates the domains for each column. This is the only format the author could find that can be used for final, human readable analysis results. It is still quite tedious to read.

A tuple is represented with natural numbers. Each number indicates a particular element in the respective domain. Usually there exists a ".map" file that lists all elements of a domain. The author still had trouble figuring out what number represents one element. Even though each line in the map starts with a numeric identifier, this identifier does not correspond with the numbers appearing in tuples. Instead a number in a tuple indicates what *line* in the map file to look at. To make things a little more tricky, `bddbddb` assigns numbers to elements starting with zero. Thus element n appearing in a tuple can be found at line $n + 1$ in the map file. The line contains a textual description of the element which is usually quite descriptive.

4.1 Generating Domains and Initial Relations

`bddbddb` needs input in the form of existing relations to create new, derived relations that represent analysis results. In order to make sense of the results we also need a map file for each domain. As alluded to earlier, we can generate a quite substantial initial set of relations and domains. This is not accomplished with `bddbddb`, however, but with another tool written by Whaley called Joeq which is also freely available from SourceForge[5].

The module `joeq_core` contains an infrastructure for compiling, traversing and executing Java bytecode. Thus it essentially implements a pure-Java Java VM. Whaley added the class `joeq.Main.GenRelations` to `joeq_core` which takes a Java class name as an argument and traverses all files reachable from there to generate the set of domains and relations listed in section 3.1. This process takes significant time and is probably not included in the performance measurements reported above. The relations are directly written in the BDD file format that `bddbddb` uses.

If these relations are not sufficient for a specific Datalog query, it will essentially be necessary to create the missing relation directly from the program text. This can be done by either extending the functionality in `GenRelations` or by using a different analysis framework. It probably requires a traversal of the program's AST. The new relation has to be written in `bddbddb`'s BDD or tuple format.

[5] joeq.sourceforge.net

5 Writing a Tainted Analysis with bddbddb

This section demonstrates how a custom analysis can be written with bddbddb.

The alias analysis presented in section 3.1 by itself is not so useful for finding bugs in a program. However, there is a variety of analyses that rely on aliasing information to reveal possible problems. Examples include the detection of state that is shared between threads and finding security vulnerabilities like leaking out references to internal data structures. Also, a comparison between the static type of a variable and the types of the objects it can point to can reveal potential to strengthen the static variable type. A modified version of this *type refinement* analysis was used to measure the precision of the analysis as reported in section 3.3.

In order to evaluate the usefulness of bddbddb for writing interesting analyses the author attempted to implement a *tainted* analysis with it. A string is tainted when it comes from an untrusted source, in particular from user input. It is a security vulnerability to use such tainted strings for operating system calls and other sensitive operations. Tainted strings should be checked, e.g. against a regular expression, before used in this way. They essentially become untainted.

A tainted analysis with bddbddb then involves tracking references to tainted strings through assignments and the like, essentially detecting aliases to tainted strings. However, a number of issues have to be solved.

1. We only want to track strings, so other kinds of objects should be excluded from the analysis.

   To make this happen, the author used a nice feature of bddbddbwhich is to identify a certain element of a domain by its name in the map file. In this case this is the Java type for strings. So the starting point of our tainted analysis can look as follows (compare it to the alias analysis query in section 3.1).

   $$tainted(v, h) \quad :- \quad vP_0(v, h), hT(h, "java.lang.String")$$

   This essentially says that any object that is created as a string is by default tainted. Note that we are building a context-insensitive analysis here.

2. If two strings are concatenated, the result will be tainted whenever one of the original strings was. Concatenation does not involve aliasing of the original strings but instead creates a new string that has to be tracked depending on the original ones.

   To realize this, we build an intermediate relation $conc \subseteq V \times V$ that records concatenations. $conc(v1, v2)$ means that $v1$ was created by concatenating $v2$ with something else. The following rules build the relation and carry taintedness over concatenations.

$$
\begin{array}{lll}
conc(v1, v2) & :- & IE(i, "java/lang/String/java/lang/Stringconcat(java/lang/String)"), \\
 & & actual(i, \_, v2), \\
 & & Iret(i, v1). \\
tainted(v1, h) & :- & tainted(v2, h), conc(v1, v2).
\end{array}
$$

3. The following rules handle fields with a separate relation *taintedH*. They rely on (context-insensitive) alias information vP about the objects touched by store (S) and load (L) operations. For a store $v1.f = v2$, we determine what tainted strings $v2$ references and what objects $v1$ points to. Loads $v2 = v1.f$ are handled similarly.

$$
\begin{array}{lll}
taintedH(h1, f, h2) & :- & S(v1, f, v2), vP(v1, h1), tainted(v2, h2). \\
tainted(v2, h2) & :- & L(v1, f, v2), vP(v1, h1), taintedH(h1, f, h2).
\end{array}
$$

4. A single string reference can be tainted in some parts of the program and untainted in others. This in particular will happen if the string is matched against a regular expression.

   Realizing this is more trouble than just writing appropriate Datalog queries. Since we are tracking the taintedness of references, a possible approach to resolve this issue is to introduce a fresh variable and assign the matched string to it. This could be done by manipulating the source code wherever a string is matched. We also need an input relation *matched* which has only one column that indicates variables which should be considered untainted due to matching. Unfortunately, the input relations for bddbddb are generated directly from Java bytecode, and simple variable assignments are optimized away by the compiler.

   Just having the *matched* relation is good enough if it is possible to mark a "real" variable that comes after the match. This in particular can be the return value of the method in which the match is performed. This solution worked for the example program used for testing, but in general is not good enough.

   A full-featured solution would be to detect string matches statically and introduce pseudo-variables into the input relations that essentially are artificial versions of the manual variable assignments discussed above. The *matched* relation could then also be generated automatically. This however would require to change the generation of initial relations from the program, something that was beyond the scope of this report. The author instead manually created the *matched* relation and considers it upon variable assignments as follows. Essentially, assignments only propagate taintedness if the target variable is not marked as matched.

$$
tainted(v1, h) \quad :- \quad A(v1, v2), tainted(v2, h), !matched(v1).
$$

5. A tainted string is not a problem per se. Our analysis just has to point out tainted strings that are handed off to sensitive operations. These operations by definition can only be identified by the programmer. Thus we need an input relation that captures this intent. We call it *untainted* here. It is similar to *matched* except that it is not a purely technical workaround but actually captures design intent. If a variable appears in this relation it means that it better never be tainted. From that we can generate an output relation *violation* that will contain all strings that are tainted but should be untainted.

$$violation(v, h) \quad :- \quad tainted(v, h), untainted(v).$$

The full Datalog file for tainted analysis can be found in the appendix. It contains all rules shown above (augmented to observe typing information upon assignments and loads) as well as the context-insensitive version of the underlying alias analysis. The *matched* relation is also considered at creation sites to allow annotating string constants as untainted.

This analysis is good enough to find out that in the example file `TaintedExample.java` used for programming assignment 2 tainted strings can make it to the sensitive method unless the return value of *filter* is appropriately marked in the *matched* relation. The analysis takes about 12 seconds on a Pentium 4 1.7 GHz processor with 512 MB RAM.

Even though this analysis works, it is somewhat tedious in practice: The two input relations *matched* and *untainted* have to be written by hand. That involves finding the right variables in the variable map file, figuring out its number (line number minus 1) and writing this number into the relation. Fortunately, variables are described quite unambiguously in the map, which helps a lot. However, this process could be completely automated by augmenting the generation of input relations from the source as discussed above. The *untainted* relation could be generated from annotations in the source code.

The analysis output, the *violation* relation, will also contain just numbers. The user now has to go ahead and look up the variables and objects it actually references. There should be some (pretty trivial) tool support to automate this process.

6 Conclusion

Understanding program information as relations and representing those relations as Binary Decision Diagrams are the key ideas that enable `bddbddb` to perform a context-sensitive inclusion-based analysis on realistic programs with reasonable overhead and high precision [5]. As we have seen, the analysis process consists of two stages, (1) creating basic relations from the program text (or bytecode) and (2) deriving new relations using Datalog queries on the database of existing relations. The tool is efficient enough to support manipulations of relations with millions of entries.

This report investigates how new kinds of analyses that depend on aliasing information can be written with bddbddb. It turns out that Datalog is able to express interesting analyses. As long as the basic relations generated are sufficient, it is relatively painless and intuitive to write new analyses. In particular, no explicit flow functions or lattices have to be implemented. It is enough to specify an appropriate query. If existing relations are not sufficient, those have to be generated from the source text, which requires more work. However, it is probably still easier than implementing flow functions. The output format of bddbddb can undergo some minor improvements to make it more readable.

The tainted analysis implemented by the author seems to be on the border of what bddbddb can do: Ideally it would require new input relations, but as a workaround those can be provided manually. The deeper reason why tainted analysis is not entirely straightforward is that one reference can be tainted in parts of the program and untainted in others. bddbddb does not support this kind of information very well (or rather, its default input relations don't). It would be desirable to get analysis information depending on the position in the source code, which is what flow functions naturally support.

Nonetheless the scalability of bddbddb is impressive and the paradigm of treating static analyses as database queries is very intuitive and easy to use. The tool is easily installed and seems to be in a mature enough stage to be used by outsiders (like the author).

References

[1] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–114, 2003.

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(9):677–691, Aug 1986.

[3] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–263, 2000.

[4] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages*, pages 31–41, 1996.

[5] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2004.

[6] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

Appendix: Tainted Analysis Datalog File

```
### Context-insensitive tainted analysis
#
# Author: Kevin Bierhoff

.basedir "pa.joeq"

### Domains

.include "fielddomains.pa"

.bddvarorder N0_F0_I0_M1_M0_V1xV0_VC1xVC0_T0_Z0_T1_H0_H1

### joeq Relations

vP0 (v:V0, h:H0) input
S (base:V0, field:F0, src:V1) input
L (base:V0, field:F0, dest:V1) input
actual (invoke:I0, num:Z0, actualparam:V1) input
formal (method:M0, num:Z0, formalparam:V0) input
Mret (method:M0, v:V1) input
Mthr (method:M0, v:V1) input
Iret (invoke:I0, v:V0) input
Ithr (invoke:I0, v:V0) input
mI (method:M0, invoke:I0, name:N0) input
IE0 (invoke:I0, target:M0) input
vT (var:V0, type:T0) input
hT (heap:H0, type:T1) input
aT (type:T0, type:T1) input
cha (type:T1, name:N0, method:M0) input
hP0 (ha:H0, field:F0, hb:H1) input
IEfilter (ccaller:VC1, invoke:I0, ccallee:VC0, target:M0) input
cA (vcdest:VC0, dest:V0, vcsrc:VC1, source:V1) input

vPfilter (v:V0, h:H0)
IEcs (ccaller:VC1, invoke:I0, ccallee:VC0, target:M0)

### context insensitive pointer analysis

A (dest:V0, source:V1)
vP (v:V0, h:H0) output
hP (ha:H0, field:F0, hb:H1)
IE (invoke:I0, target:M0)

### tainted relations
```

```
# marks internal variables that are untainted copies of matched variables
matched (v:V0) inputtuples
# explicit untainted annotations e.g. for OS call sites
untainted (v:V0) inputtuples

# tainted variables
tainted (v:V0, h:H0) outputtuples
# tainted fields
taintedH (ha:H0, field:F0, hb:H1) outputtuples
# variables that result from concatenations of other variables
conc (dest:V0, source:V1) outputtuples
# violations of the untainted annotations (above)
violation (v:V0, h:H0) outputtuples

### context insensitive pointer rules

vP(v,h) :- vP0(v,h).
IE(i,m) :- IE0(i,m).
vPfilter(v,h) :- vT(v,tv), aT(tv,th), hT(h,th).
vP(v1,h) :- A(v1,v2), vP(v2,h), vPfilter(v1,h).
hP(h1,f,h2) :- S(v1,f,v2), vP(v1,h1), vP(v2,h2).
vP(v2,h2) :- L(v1,f,v2), vP(v1,h1), hP(h1,f,h2), vPfilter(v2,h2). split
A(v1,v2) :- formal(m,z,v1), IE(i,m), actual(i,z,v2).
A(v2,v1) :- Mret(m,v1), IE(i,m), Iret(i,v2).
A(v2,v1) :- Mthr(m,v1), IE(i,m), Ithr(i,v2).

### context insensitive tainted rules

tainted(v, h) :- vP0(v, h), hT(h, "java.lang.String"), !matched(v1).
tainted(v1, h) :- A(v1,v2), tainted(v2,h), vPfilter(v1,h), !matched(v1).
taintedH(h1,f,h2) :- S(v1,f,v2), vP(v1,h1), tainted(v2,h2).
tainted(v2,h2) :- L(v1,f,v2), vP(v1,h1), taintedH(h1,f,h2), vPfilter(v2,h2).
conc(v1, v2) :-  \
        IE(i, "java/lang/String/java/lang/Stringconcat(java/lang/String)"),  \
        actual(i,_,v2),  \
        Iret(i,v1).
tainted(v1, h) :- tainted(v2, h), conc(v1, v2).
violation(v,h) :- tainted(v,h), untainted(v).
```