Staged Points-to Analysis for Large Code Bases

Nicholas Allen, Bernhard Scholz, and Padmanabhan Krishnan

Oracle Labs
Brisbane, Australia
{nicholas.allen,bernhard.scholz,paddy.krishnan}@oracle.com

Abstract. Bug checker tools for Java require fine-grained heap abstractions including object-sensitive call graphs, field information for objects, and points-to sets for program variables to find bugs in source codes. However, heap abstractions coined commonly as points-to analysis, have high runtime-complexity especially when the points-to analysis is context-sensitive, and, hence, state-of-the-art points-to analyses do not scale for large code bases.

In this paper, we introduce a new points-to framework that facilitates the computation of context-sensitive points-to analysis for large code bases. The framework is demand-driven, i.e., a client queries the points-to information for some program variables. The novelty of our approach is a pre-analysis technique that is a combination of staged points-to analyses with program slicing and program compaction. We implemented the proposed points-to framework in Datalog for a proprietary bug checker that could identify security vulnerabilities in the OpenJDK$^{\text{TM}}$ library which has approximately 1.3 million variables and 500,000 allocation-sites. For the clients that we have chosen, our technique is able to eliminate about 73% of all variables and about 95% of allocation-sites. Thus our points-to framework scales for code bases with millions of program variables and hundreds of thousands of methods.

1 Introduction

With the wide-spread use of bug checking and productivity tools [4,8,5], the scalability of static program analysis for large code bases is imminent. Object-oriented languages heavily rely on the state of the heap and for static program analyses it is crucial to reason about the state by using a heap abstraction. For most bug checking tools one cannot consider software components in isolation [19] easily. For example, Octeau et al. [19] argue that a high-fidelity analysis of component interaction is required for a comprehensive security analysis, and hence a comprehensive heap abstraction is required. A heap abstraction over-approximates the connectivity of the objects on the heap, which program variables may point to which objects, and resolves virtual dispatches to construct a call-graph. Static program analysis for object-oriented languages relies on the precision of the heap abstraction, i.e., how the effect of heap operations are abstracted including object creations, variable references, and read/write

B. Franke (Ed.): CC 2015, LNCS 9031, pp. 131–150, 2015.
DOI: 10.1007/978-3-662-46663-6_7

operations on object fields. The heap abstraction is computed via a points-to analysis, for which there exists a cornucopia of methods [21,18,12,24,23,17].

The standard context-insensitive points-to algorithm [2] has insufficient precision for many applications including security analysis [19,11]. To improve the precision of points-to, context sensitive analyses have been introduced [15,23]. There are various notions of contexts. For instance, method invocations on different receiver objects are treated differently. One could also combine the receiver object with the caller object to create the context to distinguish invocations. In the context of computing a precise points-to relation Smaragdakis et al. [23] present a number of context-sensitive analyses and identify situations where the various context-sensitive analysis can be used. In their experimental study, the authors show that the 2-Object+1-Heap context sensitive points-to relation is the most precise for object-oriented programs.

Computing the context-sensitive points-to sets for large-scale software is not viable due to high computational costs. Scalable points-to analysis for object-oriented languages such as Java has attracted a lot of attention. To overcome the performance bottleneck of context-sensitive points-to analysis, approaches that rely on refinement, demand-driven analysis and pre-analysis have been explored [24,25,22,27]. For example, to overcome the precision versus scalability trade-off for large-scale software, demand-driven analysis [28,25] is one of the most popular approaches that computes a points-to analysis for a client. The client issues specific points-to queries and for only parts of the program that affect the points-to queries, a points-to set is computed. Other approaches include preprocessing the input [22] which may increase the efficiency of context-sensitive analysis. But the presented approach is unable to compute the 2-Object+1-Heap context-sensitive points-to relations for the programs *hsqld* and *jython* from the DaCapo benchmark suite [6] which are much smaller than real-world code bases including the source code of the JDK library. Similarly, pre-analysis to measure potential impact on the final result [20] could also increase the overall efficiency. But the results reported in the paper are on relatively small programs.

The problem we address in this paper is how to compute a precise but expensive demand-driven context-sensitive points-to analysis, such as the 2-Object+1-Heap, for very large code-bases. A client issues a query that refers to variables located in a method, for which the client queries the points-to set. Thus only parts of the program that affect the client's queries are considered. However, converting a context-sensitive points-to analysis into a demand-driven analysis is challenging even for alias analysis [27]. The main issues in converting a context-sensitive points-to analysis to a demand-driven problem stems from the nature of the problem: context-sensitivity is obtained in a forward fashion (from the program start to a location) and hence can only be converted to a backward problem for the demand-driven approach with great difficulties.

Our approach overcomes this issue by employing static program slicing and program compaction for given points-to queries. The program slicing and compaction that we employ reduces the input program to a semantically equivalent

for the points-to queries[1], for which the context-sensitive points-to analysis is exhaustively run in a forward-fashion. The program compaction is a program transformation that eliminates variables and their assignments that can be expressed by other variables. Since context-sensitive analysis are sensitive to the number of statements and variables in a method, program compaction is a key ingredients for scalable context-sensitive points-to.

However, program slicing and compaction is insufficient on its own for achieving scalability for input programs used in our experimental study. To achieve scalability for programs with millions of program variables, points-to analysis has to be performed in stages. A lightweight (context-insensitive) points-to analysis is performed on the reduced input program. The lightweight points-to analysis enables the construction of a more precise call graph, since virtual dispatches can rely on a may-points-to analysis rather than the pure syntactic type information. With the improved call-graph, another round of program slicing and compaction is performed, which further reduces the input program. We refer to the first stage, i.e., program slicing and compaction with the light-weight points-to analysis as a pre-analysis, since its points-to result is not actually used beside construction a refined reduced input program. One of the advantages of our framework is that existing state-of-the-art points-to analyses can be employed.

For our experimental case study, we use Java's OpenJDK library, that consists of approximately 1.3 millions of variables, 200 thousand methods, 600 method invocations, and 400 thousand object creation sites. We have chosen clients that produce points-to query sets for tasks related to security analysis for a proprietary security analysis tool for Java. We are able to compute context-sensitive points-to relation with our points-to framework under 8 hours. Without our staged points-to framework, deep context-sensitive points-to analysis is not computable for problem sizes in the scale of Java's OpenJDK library.

The main contributions of our work are:

- We introduce a points-to framework that can use off-the-shelf exhaustive context-sensitive points-to analysis for large-scale code bases. The framework uses a refinement approach, i.e., points-to analyses of various complexity are performed in stages in conjunction with static program slicing and compaction. A preceding stage produces points-to information to further reduce the input program by refining the call-graph.
- We introduce the notion of program compaction that compacts a flow-sensitive program representation.
- We perform experiments on a large-scale code to show that our points-to framework is feasible.

This paper is organised as follows. In Section 2 we give an overview of our approach. In Section 3 we illustrate our technique by an example. In Section 4 the details of our staged approach is explained. Our implementation of the approach as well as its usefulness is described in Section 5. We conclude the paper by

[1] Note that the reduced input program may not be executable and may not produce semantically correct information for other queries which were not specified.

comparing our work with related work in the literature and highlighting the novelty of our work in Section 6.

2 Staged Points-To Framework

Our framework produces high-precision points-to analysis results for large-scale software. To overcome the complexity issue of high-precision points-to analysis, we specialise the points-to analysis for a client that issues points-to queries. Points-to queries concern variables for which the client desires the points-to set. The specialisation is performed by using static program slicing and program compaction such that an off-the-shelf points-to analysis is performed on the reduced input program. The reduced input program produces for the points-to query set the same results as the original input. However, specialisation is not sufficient on its own. For large-scale software we observed that a refinement is necessary, i.e., the points-to analysis is performed in stages. In each stage a more refined reduced input-program is produced. The refined input-program is smaller than the previous one due to the points-to analysis of the stage such that a later points-to analysis has less work to perform.

The process to obtain a context-sensitive points-to information in stages using specialisation and refinement is illustrated in Figure 1. First, the client provides the set of points-to queries that are passed on to the step that computes the initial slice of the program. The initial slice is computed based on syntax information only rather than performing any points-to analysis. After this the slice is compacted, i.e., variables that are not of interest or are not actual/formal parameters, and return values are eliminated. The compacted slice is passed on a flow-insensitive and field-sensitive points-to analysis. The points-to analysis builds a heap-abstraction and hence field sensitivity is taken into account. A related issue is the construction of the call-graph [26]. Virtual method resolution could be done in conjunction with the points-to analysis. This leads to a mutual dependency between the points-to analysis and the construction of the call-graph. The context-insensitive analysis thus builds a call-graph that is more precise than the initial call-graph constructed using the Class Hierarchy Analysis (CHA).

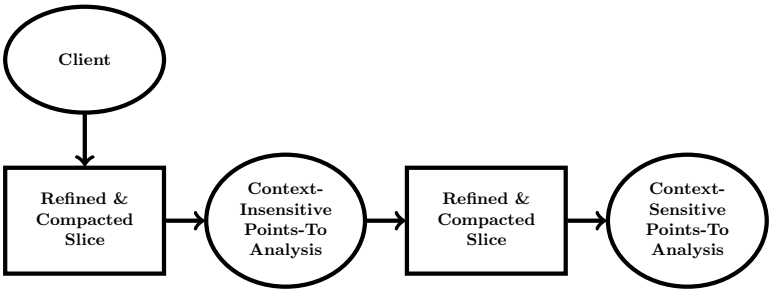


Fig. 1. Staged process of slicing and analysis for context-sensitive points-to analysis

The points-to and the call-graph relations are used to compute the final slice to refine the virtual dispatch of call-sites. The final slice is passed on to perform the context-sensitive analysis. The use of the less expensive points-to analysis and slicing can be viewed as a particular instance of pre-analysis for the final context sensitive points-to analysis. The pre-analysis enables us to compute the context-sensitive points-to set in an effective fashion. Note that we are able to use the context-insensitive points-to results to further slice the program as it is an over-approximation of the desired result. In general, we can use any points-to relation that is computable on the initial slice provided it is a sound over-approximation of the desired points-to relation. Thus, context-insensitive and 2-Object+1-Heap can be seen as an instance of our approach.

3 Motivating Example

In this section we present an example program and show the effect of the syntactic based and context-insensitive based slicing. The aim is to remove unnecessary variables and objects. This example is representative of actual code fragments on which our analysis is performed. Consider the program in Listing 1.1. It has trusted and untrusted objects which can be used in a secure or an insecure setting. The client's query is only interested in the use of untrusted objects in a secure setting. Thus non-security related actions and trusted objects are removed by our analysis.

Listing 1.1. Original Program

```
1  class SecurityApplication {
2      public static void main(String[] args) {
3          String result = setup(args);
4          System.out.println(result);
5
6           SecurityFactory uFactory = new UntrustedSecurityFactory();
7           SecurityFactory tFactory = new TrustedSecurityFactory();
8
9           SecurityObject uObject = uFactory.getSecurityObject();
10          SecurityObject tObject = tFactory.getSecurityObject();
11
12         doSecurity(uObject, tObject);
13     }
14
15     private static void doSecurity(SecurityObject secObj1,
16                                    SecurityObject secObj2) {
17         SecurityAction action1 = new SecurityAction();
18         SecurityAction action2 = new SecurityAction();
19          action1.object = secObj1;
20          action2.object = secObj2;
21
22         Object res1 = action1.invoke();
23         Object res2 = action2.invoke();
24
25         doOtherThings(res1, res2);
26     }
27
28     private static String setup(String[] args) { ··· }
29
30     private static void doOtherThings(Object result1, Object result2) { ··· }
31 }
32
```

```
33  interface SecurityFactory {
34      public SecurityObject getSecurityObject();
35  }
36
37  class UntrustedSecurityFactory implements SecurityFactory {
38      public SecurityObject getSecurityObject() {
39          SecurityObject newObj = new UntrustedSecurityObject();
40          return newObj;
41      }
42  }
43
44  class TrustedSecurityFactory implements SecurityFactory {
45      public SecurityObject getSecurityObject() {
46          SecurityObject newObj = new TrustedSecurityObject();
47          return newObj;
48      }
49  }
50
51  class SecurityAction {
52      public SecurityObject object;
53      public Object invoke() {
54          SecurityObject storedObject = this.object;
55          return invoke0(storedObject);
56      }
57      private static native Object invoke0(SecurityObject obj);
58  }
59
60  class SecurityObject {···}
61  class UntrustedSecurityObject extends SecurityObject {···}
62  class TrustedSecurityObject extends SecurityObject {···}
```

Specifically, assume that the client's query is: "whether at the invocation of `invoke0` (at line 55) the parameter `storedObject` points-to an untrusted heap object of type `UntrustedSecurityObject`." In the example, the only allocation of the untrusted object is on line 39. It can be seen that variables such as `args`, `result`, `res1`, `res2` and methods such as `setup` and `doOtherThings` have no influence on the desired result. Hence they can be removed from the initial slice.

Given the initial slice, the points-to analysis needs to determine if the variable `storedObject` at line 55 can point to the new object created at line 39. The variable `uObject` on line 9 will point to the untrusted object as it holds the return value of the invocation to the `getSecurityObject` method. The value now flows from `uObject` to `secObj1` and then to `action1.object` on line 19. The method `action1.invoke` results in the untrusted object being used in the call of `invoke0`. As there is no other value flow, we can safely ignore the other variables. That is, the context-insensitive points-to indicates that variables such as `tFactory` (on line 7), `tObject` (on line 10) and `action2` (on line 20) do not point to an untrusted heap object. Hence these variables and the allocation sites they point-to can be removed from the slice. Note that the method `doSecurity` now has only one parameter in the computed slice. The main points-to relation where the variables are in rectangular boxes and allocation-sites are in circles is shown in Figure 2. The figure also shows objects that are pruned (indicated via being crossed out) by the slicing operation. The resulting slice is shown in Listing 1.2.

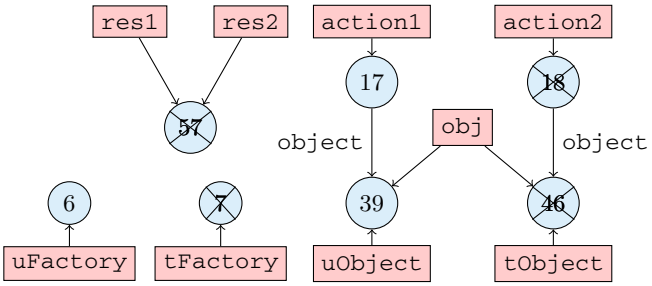


Fig. 2. Removal of Objects

Listing 1.2. Final Slice

```
1  class SecurityApplication {
2      public static void main() {
3          SecurityFactory uFactory = new UntrustedSecurityFactory();
4          SecurityObject uObject = uFactory.getSecurityObject();
5          doSecurity(uObject);
6      }
7
8      private static void doSecurity(SecurityObject secObj1) {
9          SecurityAction action1 = new SecurityAction();
10         action1.object = secObj1;
11         action1.invoke();
12     }
13 }
14
15 interface SecurityFactory {
16     public SecurityObject getSecurityObject();
17 }
18
19 class UntrustedSecurityFactory implements SecurityFactory {
20     public SecurityObject getSecurityObject() {
21         SecurityObject newUObj = new UntrustedSecurityObject();
22         return newUObj;
23     }
24 }
25
26 class SecurityAction {
27     public SecurityObject object;
28     public Object invoke() {
29         SecurityObject storedObject = this.object;
30         return invoke0(storedObject);
31     }
32     private static native Object invoke0(SecurityObject obj);
33 }
34
35 class SecurityObject {···}
36 class UntrustedSecurityObject extends SecurityObject {···}
```

4 Steps of the Staged Points-To Framework

In order to express the key aspects of our staged points-to analysis, we first define a small flow-insensitive object-oriented language that includes core Java features. The language has types so that class and sub-class relationships and interfaces can be expressed. We also use the notion of types for object-fields,

call-sites and method signatures to model virtual dispatch. We assume variables reside in a single method and are unique. A variable has attached a declared type that may not be identical to object type of the object it is referring to. Each method has a special variable *self* that denotes the instance object itself. Every method is defined in a class, and is inherited by its sub-class (if not overridden). A method may consist of one or more of the following statements:

[L1] Heap allocations: $x = new\ C()$ creates a new instance of variable C.
[L2] Assignment: $x = y$ assigns variable x the value of variable y. Note that the variable *self* is pre-defined and cannot be assigned a value.
[L3] Assignment Cast: $x = (T)y$ assigns variable x the value of variable y by casting the value to type T.
[L4] Loading of a field: $x = y.f$ loads a value from the field f from the value in variable y and assigns the loaded to the variable x.
[L5] Storing of a field: $x.f = y$ stores the value present in variable y in the field f of the value in variable x.
[L6] Return Statement: *return* u returns the value u.
[L7] Call-Site: $y = o.s(x_1, \ldots, x_k)$ calls method a method m that is declared in type of variable o or any suitable super-class.

Note that we do not consider control-flow constructs in the language, since it was shown that flow sensitive analysis is less important than a context sensitive analysis [14,16]. Note that for real implementation we need to consider static fields, static method calls, calls to super, etc. For sake of simplicity, we do not discuss them here. In addition, we do not consider reflection mechanisms either.

4.1 Computing the Initial Slice

Here we describe the computation of a slice.The initial slice is computed by tracing the dataflow of the client specified query variables backwards by the assignment relation. The slice S is a multi-set containing variables, field types, and methods of a program. We use the auxiliary function $\tau(x)$ which identifies the type of a variable. Initially, the slice contains the query variables and the methods where the variables reside in. We use a model theoretic approach to describe the set that contains the initial slice. We search for the smallest set S for which following conditions hold:

[S1] All query variables v and methods m that contain variable v are in S.
[S2] If there is an object creation $x = newC()$ and $x \in S$, then $\{C, cons_C\} \subseteq S$ where $cons_C$ is the object constructor for the type C.
[S3] If there is an assignment $x = y$ and $x \in S$, then $y \in S$.
[S4] If there is an assignment cast $x = (T)y$ and $x \in S$, then $y \in S$.
[S5] If there is a load $x = y.f$ and $x \in S$, then $\tau(y).f \in S$, and $y \in S$.
[S6] If there is a store $x.f = y$ in method m, and there is $t'.f \in S$ that is compatible with $\tau(x).f$, then $\{m, x, y\} \subseteq S$.
[S7] If there is a callsite $y = o.s(x_1, \ldots, x_k)$ residing in method m, and callsite $o.s$ is compatible with a method $m' \in S$, then $m \in S$ and $o \in S$.

[S8] If there is a callsite $y = o.s(x_1, \ldots, x_k)$ residing in method m, callsite $o.s$ is compatible with a method $m'(z_1, \ldots, z_k) \in S$, and $z_i \in S$, then $x_i \in S$.

[S9] If there is a callsite $y = o.s(x_1, \ldots, x_k)$, $y \in S$, then for all methods m that are compatible with the call-site $o.s$ and for all *return* z residing in m, then $\{m, z\} \subseteq S$.

Rule [S2] extends the slice to include types and the constructors methods of objects that are created. Rule [S3] extends the slice to the source of the assignment, if the destination of the assignment is in the slice. Rule [S4] extends the slice for assignment casts. Rule [S5] adds the field $\tau(y).f$ if the result of the load operation on a field is in the slice. Note that we add all variables whose type is compatible with the loading of the field f. Rule [S6] adds the method m and the variables x and y to the slice, if a compatible field type can be found in the slice. Rule [S7] adds the caller to the slice and the instance object. Rule [S8] adds the actual parameters to a slice, if the formal parameter of a method are in the slice. Rule [S9] adds the return variables to the slice if the result variable is in the slice.

4.2 Compaction

The compaction process eliminates variables that do not contribute directly or indirectly to the points-to query. In addition, variables that store intermediate results can be eliminated. For example, in a method without points-to queries, the only variables which should remain are variables of object-creation sites, actual/formal parameters, instance variables, and return variables. The compaction reduces the size of the data-flow graph of a method, and, hence speeds up the convergence of the points-to analysis. The compaction can be perceived as an orthogonal process to slicing.

The implementation of compaction is based on standard techniques such as reaching-definitions, copy propagation and dead and redundant code elimination schemes [3,10] suitably adapted for object-oriented programs. Since there is no control-flow, the computation reduces to the computation of equivalence classes via the assign relation. This assign relation builds a value flow graph and involves the query variables, formal arguments/actual arguments, return statements, object creation sites, receiver objects of method invocations at call-sites.

The example on the left in Figure 3 is rewritten to the example on the right. The intermediate variables `z` and `y` are eliminated as assignments to them are redundant. That is, the assignment `z = x` is not necessary as all occurrences of `z` can be replaced by `x`. Formally, the variables `x`, `y` and `z` are in the same equivalence class while the variables `a` and `b` are in another equivalence class.

In the above example only one definition reached the use. That is, only `x` reaches `z` and only `a` reaches `b`. In general, various definitions could reach a point of use. Our solution is to maintain a subset of definitions that reach a variable. The subset in then used to identify all the assignments that are necessary. An example of this situation is shown in Figure 4.

```
1  int foo(int x) {
2      z = x;
3      y = x;
4      a = goo(z);
5      b = a;
6      return b;
7  }
```

```
1  int foo(int x) {
2      a = goo(x);
3      return a;
4  }
```

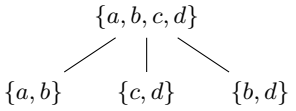
Fig. 3. Example of Compaction

```
1  int foo(int arg0, int arg1,
2          int arg2) {
3      x = arg0;
4      y = arg1;
5      z = x;
6      z = y; z:{arg0, arg1}
7      a = arg2;
8      a = z; a:{arg0, arg1, arg2}
9      goo(a);
10     z = a; z:{arg0, arg1, arg2}
11     return z;
12 }
```

```
1  int foo(int arg0, int arg1,
2          int arg2) {
3      arg0arg1arg2 = arg0;
4      arg0arg1arg2 = arg1;
5      arg0arg1arg2 = arg2;
6      goo(arg0arg1arg2);
7      return arg0arg1arg2;
8  }
```

Fig. 4. Multiple values: Compaction

In general we replace variables by reusing subsets of reaching definitions. Note that this is sound in our context as we are computing a may-point-to relation. The subsets of reaching definitions can be arranged in a Hasse-diagram representing the partial order. The assignments can then be derived by reusing the assignments used for the relevant subsets and taking into account the variables from the reaching definitions that have been covered. An example of a partial order and the generated assignments is given in Figure 5. The emphasised variables are those that are introduced by compaction whereby variables `a` to `d` are program variables of the input program. All other intermediate variables that contributed to the partial order are elided.




```
1  ab = a; ab = b;
2  cd = c; cd = d;
3  bd = b; bd = d;
4  abcd = ab; abcd = cd;
```

Fig. 5. Partial Order and Assignments

4.3 Context-Insensitive Points-To Analysis

The improvement of the context-insensitive field-sensitive point-to analysis is the heap-abstraction, i.e., fields are analysed and hence more precise information is obtained for resolving virtual method dispatches. For each instance variable, the points-to set consisting of object-creation sites is computed, which we denote by $pt(v)$. For each field of an object-creation site, a set of object-creation sites that this field may point to, which we denote by $fpt(o, f)$, is also computed. An object-creation site has associated an actual type that is used to compute more precise virtual method dispatches. We search for the smallest points-to set for the current slice such that following points-to rules hold:

[P1] If there exists a heap allocation $a : x = new\ C(\ldots)$ in a method $m \in S$, then $a \in pt(x)$.
[P2] If there exists an assignment $x = y$ in a method $m \in S$, then $pt(y) \subseteq pt(x)$.
[P3] If there exists an assignment cast $x = (T)y$ in a method $m \in S$, then $o \in pt(x)$, for all $o \in pt(y)$ and $\tau(o) \leq T$.
[P4] If there exists a load statement $x = y.f$ in a method $m \in S$, then $pt(o.f) \subseteq pt(x)$ for all $o \in pt(y)$.
[P5] If there exists a store statement $x.f = y$ in a method $m \in S$, then $pt(y) \subseteq fpt(o, f)$ for all $o \in pt(x)$.
[P6] If there exists a call-site $y = o.s(x_1, \ldots, x_k)$ in a method $m \in S$, and for all methods $m'(z_1, \ldots, z_k)$ that are compatible with s and the types in $pt(o)$, then $pt(x_i) \subseteq pt(z_i)$ for all i, $1 \leq i \leq k$, $pt(o) \subseteq pt(m'.self)$, and $pt(y) \subseteq pt(u)$ for all *return* u residing in $m'(z_1, \ldots, z_k)$.

Note that the above rules are standard [21] for a field-sensitive context-insensitive points-to analysis. The points-to relation is used to obtain the final slice. The slice produced by using the context-insensitive points-to analysis will include variables, methods, heap allocation sites along with points-to facts for the client's query. While the rules are similar in structure as the rules in the variable-based slicing, there are some subtle differences as we now have the actual object creation sites for each variable and field-sensitivity.

[S'1] If $o \in pt(v)$ such that (v, o) is part of the facts for the client's query and m is the method that contains v, then $\{(o, v), m\} \subseteq S$.
[S'2] If $(v, h) \in S$, then $\{v, h\} \subseteq S$. Similarly if $(o, f, o') \in S$ then $\{o, f, o'\} \subseteq S$.
[S'3] If there is an assignment $x = y$ and $(x, o) \in S$, and $o \in pt(y)$ then $y \in S$.
[S'4] If there is a load $x = y.f$ and $(x, o) \in S$ and $o' \in pt(y)$ such that $o \in fpt(o', f)$, then $\{(o', f, o), (y, o')\} \subseteq S$.
[S'5] If there is a store $x.f = y$ in method m, and $o \in pt(x)$, $o' \in pt(y)$, $(o, f, o') \in S$, then $\{m, (x, o), (y, o')\} \subseteq S$.

The rule [S'1] adds all the facts that are relevant to the client's query to the slice. The rule [S'2] adds all the constituent elements of the points to and field-points-to relation in the slice to the slice. The rule [S'3] extends the slice with the variable on the right hand side of the assignment provided it points-to an object

in the slice. The rule [S'4] adds the points-to facts related to a load operation to the slice provided the variable which has result of the load operation is in the slice. The rule [S'5] adds the method m and the points-to facts for the variables x and y to the slice, if a field is stored into and the field-points-to relation is part of the slice.

4.4 Context-Sensitive Points-To Analysis

The context-sensitive points-to analysis is the final stage of the analysis. It uses the last slice to compute the most precise points-to information. The contexts as well as the points-to relation is computed only on the slice and thus the analysis can use the standard techniques [23]. The construction of the slice guarantees that if o belongs to $cpt(c, x)$, then o will also belong to $pt(x)$ where cpt represents the context-sensitive points-to relation and c the context. So as long as the result of our context-sensitive analysis is a strict refinement of the results of the context-insensitive analysis, the result of the pre-analysis can be used for the context-sensitive analysis. The context-sensitive analysis further refines the call-graph. The points-to analysis will use the notion of a method being reachable in a context and use it to compute the points-to set. As noted in Section 1, one can use a method's receiver object and the object that allocates this receiver object as the context. As the resolution of the virtual method being invoked will depend on the type of the receiver object, certain methods will not be reachable in certain contexts. More details on this is available in the literature [7,23]. We use $reach(c, m)$ to indicate that method m is reachable in context c. Contexts are updated when a method is invoked or when an object is created. We use $extend(c, st)$ to identify the new context, where st is a statement that represents an invocation or an object creation. The rules for assignment and the linking of actual to formal parameters in a method invocation is shown below.

[CSP1] If there exists an assignment $x = y$ in a method $m \in S$, and $reach(c, m)$ then $cpt(c, y) \subseteq cpt(c, x)$.

[CSP2] If there exists a call-site $y = o.s(x_1, \ldots, x_k)$ in a method $m \in S$ with $reach(c, m)$ and for all methods $m'(z_1, \ldots, z_k)$ that are compatible with s and $cpt(c, o)$, then $cpt(c, x_i) \subseteq cpt(c', z_i)$ for all i, $1 \leq i \leq k$ where $c' = extend(c, o.s(x_1, \ldots, x_k))$

One can show that for any program P, if v is a variable in a client's query then $cpt_P(c, x) = cpt_S(c, x)$ where cpt_P and cpt_S represent the context-sensitive points-to set computed for the entire program P and slice S respectively. That is, our slicing technique does not lose any relevant information.

5 Implementation and Results

In this section we demonstrate both the ineffectiveness of using off-the-shelf context-sensitive points-to analysis and the effectiveness of our staged points-to

Table 1. Context Sensitive Analysis: Not computable over the JDK

Context Sensitive Analysis	Outcome
1-Call-site-sensitive	Does not terminate after 20 hours
1-Object-sensitive	Does not terminate after 20 hours
2-Call-site-sensitive+1-Heap	Does not terminate after 20 hours
2-Object-sensitive+1-Heap	Out of memory

framework using specialisation (i.e. program slicing and compaction) and refinement (i.e. staging points-to). We have implemented our technique with Datalog [1] based on the DOOP [7] framework. We use OpenJDK 7 build 147 (rt.jar) as the artefact that is subject to various analyses. We ran our experiments on an Intel Xeon E5-2660 (2.2GHz) machine with 256GB RAM using the LogicBlox engine [13].

We report results of our experiments with some of the existing context-sensitive points-to analysis that was *not* computable *in general* for the OpenJDK library without specialisation and refinement. OpenJDK 7 (rt.jar) has more than 2 million lines of Java code. Note that lines of code is not necessarily an accurate indication of the complexity for the points-to analysis. The number of variables and allocation sites is a better indication of the effort required to perform points-to analysis. The OpenJDK 7 library has close to 1.3 million variables and about 500,000 heap allocation sites and goes well beyond the largest benchmarks sizes such as DaCapo [6] mainly used for points-to research in literature.

The results of points-to without specialisation and refinement are shown in Table 1. We terminated many of the analyses after 20 hours as that was well over our self-imposed time budget of 8 hours.

Choice of Client. As results of our analysis is dependent on the client's query, we choose different security analysis clients. We choose four security analysis that relate to access control and are derived from section 9 of the Java Secure Coding guidelines (JSCG) [9]. The guidelines specify certain properties where security sensitive methods are invoked. Examples of security sensitive methods include `AccessController.doPrivileged()`, `Class.forName()` and `Class.newInstance()`. The restrictions on the invocation of such methods require appropriate permissions, use of untainted objects and escaping of results from the JDK to the application.

Typically a client is interested in identifying locations in the program that violate the Java Security Guidelines. For this purpose the client identifies invocations to the security sensitive methods that are potential violations. Points-to information is required to determine properties such as taintedness of any of the arguments and results escaping from the JDK to the application. From a security view point other invocations that do not influence the security related invocations are not relevant. Note that although security is the principal motivation, we do not report any security specific results here. The focus here is purely on client driven calculation of a suitable context-sensitive points-to set.

5.1 Staged Points-To Framework Results

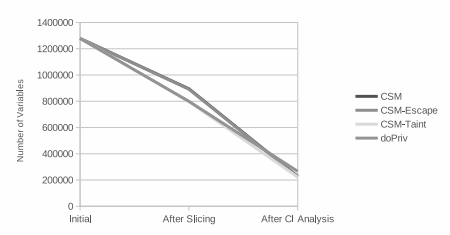
In this section we present the results of our analysis. We show that

(a) the reduction in the number of variables and allocation-sites due to program slicing and compaction,
(b) the size of the various points-to relations and
(c) the size of the call-graph

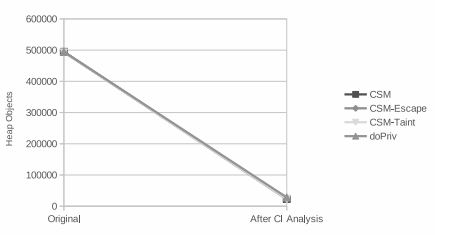
Table 2. Number of Variables

Client	Variables in Client's Query	Variables in Initial Slice	Variables after Context Insensitive Analysis
CSM	3,885	895,100	228,054
CSM-Escape	1,321	891,641	224,707
CSM-Taint	847	797,217	222,192
doPrivileged	12,202	799,484	266,558

The reduction in the number of program variables using our pre-analysis is given in Table 2. The behaviour is uniform across all clients (see Figure 6(a)) and there is no obvious link between the number of variables in the client's query and the number of variables in the various reduced input programs. The variables used either as parameters or as the base in the relevant invocations yields the variables that are in the client's query. The variable based slicing reduces the number of potentially relevant variables by approximately 30%. The context insensitive analysis on the slice further reduces the number of relevant variables by approximately 73%. This results in a slice with only about 18% of the original set of variables. The reduction in the number of allocation-sites is shown in Table 3. The variable based slice does not significantly reduce the number of allocation sites. But the context insensitive analysis marks only around 5.0% of the original set of allocation sites as relevant (shown in Figure 6(b)). This



(a) Variables



(b) Allocation Sites

Fig. 6. Size Reduction

Table 3. Number of Allocation Sites

Client	Allocation Sites in Initial Slice	Allocation Sites after Context-Sensitive Analysis
CSM	494,560	23,215
CSM-Escape	494,111	22,928
CSM-Taint	492,821	22,850
doPrivileged	494,560	28,298

Table 4. Size of Points-To Relation

Client	Context-Insensitive	Context-Sensitive	Context-Sensitive (No Context)
CSM	115,470,090	435,721,445	1,895,206
CSM-Escape	115,374,473	434,238,551	1,885,825
CSM-Taint	107,538,308	296,998,797	1,585,422
doPrivileged()	141,053,434	413,813,791	1,952,346

massive reduction in the number of allocation sites combined with a significant reduction in the number of variables is the key reason for the success of our technique.

The size of the points-to relation is shown in Table 4. For all the clients there are more than 100 million points-to relations in the context-insensitive points-to set. They are refined to more than 400 million points-to relations, except in the case of CSM-Taint where 296 million facts are generated, an increase by a factor of more than 3.5. If the contexts from the context-sensitive relation are elided, there are about 1.9 million facts, except in the case of CSM-Taint where there are about 1.6 million facts. The sheer size indicates the memory required to hold these relations.

The average number of objects a variable points to gives insight to the problem. In the context-insensitive case it is about 140 objects per variable while in the context-sensitive case it is only about 8 objects per variable. This is shown in Table 5.

Table 5. Average Number of Allocation Sites Per Variable

Client	Context-Insensitive	Context-Sensitive
CSM	129.0	8.3
CSM-Escape	129.4	8.4
CSM-Taint	134.9	7.1
doPrivileged()	176.4	7.3

The call-graph relation generated by the context-insensitive points-to analysis contains approximately 300,000 points-to relations. This is refined to more than 80 million facts, except in the case of CSM-Taint where close to 59 million facts are generated. This represents an increase in the size by a factor of 247. However, if just the call-graph edges are (without the contexts) examined, there are approximately 140,000 edges. So the context-sensitive analysis reduces the number of edges from the context-insensitive relation by about 60%.

Table 6. Size of Call-Graph Relation

Client	Context-Insensitive	Context-Sensitive	Context-Sensitive (No Context)
CSM	337,658	83,472,063	141,066
CSM-Escape	337,482	83,415,055	140,535
CSM-Taint	332,135	58,926,898	137,809
doPrivileged()	373,991	77,723,735	153,079

Despite the size of the call-graph edges and the context-sensitive points-to set, slicing enables the computation of the context-sensitive points-to and call-graph relations in under 4 hours and 45 minutes which is well under our limit of about 8 hours. The break up of the time taken by each stage is shown in Table 7. All times are given in seconds.

Table 7. Timing Information

Client	Initial Slice	Context Insensitive	Context Sensitive	Total
CSM	285	1,913	14,936	17,134 (4.76hrs)
CSM-Escape	293	1,903	14,790	16986 (4.72hrs)
CSM-Taint	233	1,694	5,959	7886 (4.05hrs)
doPriv()	256	2,508	11,844	7886 (4.05hrs)

To summarise, our experiments show that the reduction in the number of variables and allocation sites using our points-to framework. Our experiments provides insight into the sizes of the various relations that are computed on the reduced input program. From our experimentation it is hard to establish a relationship between number of variables specified by client and performance. Ultimately it depends on the size of the reduced input program, which is hard to estimate purely from the client's query.

Limitation of the Experiment

Firstly, all experiments are conducted on various versions of JDK. Although we have reported results only on the OpenJDK 7, the results on other versions of JDK are similar. All large code bases may not have the same characteristics as the JDK. Hence our analysis might produce different results. But given that we have analysed the JDK in its entirety gives us confidence that the staged approach can be applied to other large systems. The second issue relates to the security related queries. All of them were related to access control and derived from the Java Security Coding Guidelines. While the coding guidelines cover key security related situations, there are many aspects of security that we have not covered. However, an initial analysis of the JDK shows that the security sensitive operations are related to each other which is why the number of variables and allocation sites are in the initial slice are independent on the client. We believe that clients derived from other secure coding guidelines will produce similar results.

6 Related Work

The novelty of our approach is using well known demand-driven points-to analyses and the notion of slicing and combining them in the right order to obtain a scalable demand-driven refinement technique to compute context sensitive points-to relations for large systems. We have demonstrated that our approach can compute the 2-Object-sensitive+1-Heap context-sensitive points-to set for security related analysis of the JDK. No existing work has reported results on any program that is as large as the JDK.

Most of the existing works report results on programs from standard benchmarking suites such as DaCapo [6]. All the programs in these suites are much smaller than the JDK. Jython, which is part of the DaCapo suite [6] is used as an example of a typically large example for static analysis [24,23]. Smaragdakis et al. [23] report that they were *unable* to compute the entire 2-Object+1-Heap sensitive points-to relation for Jython. As the JDK has about 10 times the number of heap allocation sites and about 6 times the number of variables of Jython, the standard DOOP technique cannot be used to compute the context-sensitive points-to relation for the JDK.

It is difficult to perform an accurate comparison of our approach with other approaches. This is because the analysis depends on both the size of the input program as well as the query that is used in the demand-driven refinement process. For instance, [22] achieve a 30% reduction in the number of variables; but that is independent of any client query. But they do not reduce the number of heap objects as they do not compute a relevant slice. None of the existing work use security analysis for their refinement nor do they use slicing to get a handle on complexity.

Sridharan and Bodík [24] use refinement with cast checking and disjoint analysis of factory methods as the criteria. Yan et al. [27] do not use refinement – they compute the may alias relation directly in a demand driven fashion using

CFL reachability. They develop a specific context-sensitive analysis based on reachability and summarisation and do not compute the points-to relation.

Pre-analysis is used to selectively compute the context-sensitive points-to set [20]. They use the pre-analysis to estimate the potential benefit before they compute context-sensitive facts. Thus for the same program they have context-insensitive facts for some program fragments while other fragments have context-sensitive facts. Our approach is orthogonal to their work as we compute the context-sensitive facts for the *entire slice* where the slice is identified using a demand-driven approach. Furthermore, their approach is for C programs and it is not clear how easily it can be applied to object-oriented programs. Finally, their results are on all relatively small programs (the largest program they use is `a2ps-4.14` which has fewer than 65K lines of code).

Table 8 summarises the key differences between the different approaches.

Table 8. Comparison of Different Approaches

Approach	CFL Based Alias or Variable Reduction	Heap reduction	Client Based Reduction or Selective Contexts
Set based pre-processing [22]	✓	✗	✗
Demand-driven alias analysis [27]	✓	✗	✓
Selective context-sensitive analysis [20]	✗	✗	✓
Our Work	✓	✓	✓

7 Conclusion

In this paper we introduced a staged demand-driven points-to framework that uses specialisation and refinement. The specialisation is achieved by static program slicing and program compaction. The refinement is achieved by staging the points-to analysis, i.e., a pre-analysis refines the reduced input program for the later stage. We have implemented our technique using the DOOP framework and have presented our results on the OpenJDK version 7 build 147 using 4 security related client queries. We have observed that our technique is able to reduce the number of variables and allocation sites which enables the computation of the 2-Object+1-Heap context-sensitive points-to well within our time bound of 8 hours. Our technique produces high-precision points-to analysis information for code bases with million of program variables and thousands of invocation sites going beyond the state-of-the-art.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Ph.D. thesis, DIKU, University of Copenhagen (Fall 1994)
3. Appel, A.W.: Modern Compiler Implementation in Java. Cambridge University Press (1998)
4. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)
5. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later – using static analysis to find bugs in the real world. Comm. ACM 53, 66–75 (2010)
6. Blackburn, S.M., Garner, R., Hoffmann, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA 2006: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (2006)
7. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2009, pp. 243–262. ACM (2009), `http://doi.acm.org/10.1145/1640089.1640108`
8. Cifuentes, C., Keynes, N., Li, L., Hawes, N., Valdiviezo, M.: Transitioning Parfait into a development tool. IEEE Security and Privacy 10(3), 16–23 (2012)
9. Corporation, O.: Secure coding guidelines for java se (April 2014), `http://www.oracle.com/technetwork/java/seccodeguide-139067.html`
10. Debray, S.K., Evans, W., Muth, R., De Sutter, B.: Compiler techniques for code compaction. ACM Transactions on Programming Languages and Systems 22(2), 378–415 (2000)
11. Feng, Y., Anand, S., Dillig, I., Aiken, A.: Apposcopy: Semantics-based detection of android malware through static analysis. In: International Symposium on Foundations of Software Engineering (2014) (to appear)
12. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 240–260. Springer, Heidelberg (2006)
13. Green, T.J., Aref, M., Karvounarakis, G.: Logicblox, platform and language: A tutorial. In: Barceló, P., Pichler, R. (eds.) Datalog 2.0 2012. LNCS, vol. 7494, pp. 1–8. Springer, Heidelberg (2012)
14. Hind, M., Pioli, A.: Which pointer analysis should i use? In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 113–123. ACM (2000)
15. Lhoták, O., Hendren, L.J.: Context-sensitive points-to analysis: Is it worth it? In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 47–64. Springer, Heidelberg (2006)
16. Lhoták, O., Hendren, L.J.: Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Transactions on Software Engineering Methodology 18(1) (2008)

17. Lu, Y., Shang, L., Xie, X., Xue, J.: An incremental points-to analysis with cfl-reachability. In: Jhala, R., De Bosschere, K. (eds.) Compiler Construction. LNCS, vol. 7791, pp. 61–81. Springer, Heidelberg (2013)
18. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Transaction on Software Engineering Methodolology 14(1), 1–41 (2005), http://doi.acm.org/10.1145/1044834.1044835
19. Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Le Traon, Y.: Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In: Proceedings of the 22nd USENIX Conference on Security (SEC), pp. 543–558. USENIX Association (2013), http://dl.acm.org/citation.cfm?id=2534766.2534813
20. Oh, H., Lee, W., Heo, K., Yang, H., Yi, K.: Selective context-sensitivity guided by impact pre-analysis. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 475–484. ACM (2014)
21. Ryder, B.G.: Dimensions of precision in reference analysis of object-oriented programming languages. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 126–137. Springer, Heidelberg (2003)
22. Smaragdakis, Y., Balatsouras, G., Kastrinis, G.: Set-based pre-processing for points-to analysis. In: ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 253–270 (2013)
23. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 17–30. ACM (2011), http://doi.acm.org/10.1145/1926385.1926390
24. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2006, pp. 387–400. ACM (2006), http://doi.acm.org/10.1145/1133981.1134027
25. Sridharan, M., Gopan, D., Shan, L., Bodik, R.: Demand-driven points-to analysis for Java. In: Proceedings of the 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 59–76. ACM (2005), http://doi.acm.org/10.1145/1094811.1094817
26. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: Rosson, M.B., Lea, D. (eds.) OOPSLA 2000, pp. 281–293. ACM (2000)
27. Yan, D., Xu, G., Rountev, A.: Demand-driven context-sensitive alias analysis for Java. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA), pp. 155–165. ACM (2011), http://doi.acm.org/10.1145/2001420.2001440
28. Zheng, X., Rugina, R.: Demand-driven alias analysis for C. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 197–208 (2008), http://doi.acm.org/10.1145/1328438.1328464