NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

UNDERGRADUATE THESIS

Static Dependence Analysis for Java

Nikolaos Filippakis

Supervisors: **Yannis Smaragdakis,** Associate Professor NKUA
Georgios Balatsouras, PhD Student NKUA

ATHENS

APRIL 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Στατική Ανάλυση Εξαρτήσεων για Java

Νικόλαος Φιλιππάκης

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Αναπληρωτής Καθηγητής ΕΚΠΑ
Γεώργιος Μπαλατσούρας, Διδακτορικός Φοιτητής ΕΚΠΑ

ΑΘΗΝΑ

ΑΠΡΙΛΙΟΣ 2016

UNDERGRADUATE THESIS

Static Dependence Analysis for Java

Nikolaos Filippakis
R.N.: 1115201000103

Supervisors: **Yannis Smaragdakis,** Associate Professor NKUA
Georgios Balatsouras, PhD Student NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Στατική Ανάλυση Εξαρτήσεων για Java

Νικόλαος Φιλιππάκης
Α.Μ.: 1115201000103

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Αναπληρωτής Καθηγητής ΕΚΠΑ
Γεώργιος Μπαλατσούρας, Διδακτορικός Φοιτητής ΕΚΠΑ

ABSTRACT

Many applications use sensitive or untrusted data, such as data provided by a user. For this reason, it is useful to be able to discern the flow of such kinds of data inside an application, as well as the parts of the application whose execution is dependent on them.

We present a dependence analysis which, given some source instructions in a Java program, computes an overapproximation of the parts of the program whose execution can be influenced by these instructions. The central idea is that user-supplied data should not be able to influence critical parts of the program and that, inversely, sensitive data in the program should not leak out information to the user.

The analysis was written in the Datalog language and is based on the Doop framework. Specifically, the analysis logic was implemented in about 200 lines of Datalog code, something that demonstrates the capabilities of Doop in the creation of compact and expressive static analyses.

Subject area: Dependence Analysis

Keywords: static analysis, security, Java, Datalog, Doop

ΠΕΡΙΛΗΨΗ

Πολλές εφαρμογές λειτουργούν χρησιμοποιώντας ευαίσθητα ή αναξιόπιστα δεδομένα, όπως τα δεδομένα που παίρνουν από τον χρήστη. Για αυτόν τον λόγο, είναι χρήσιμο να μπορεί κάποιος να δει τη ροή τέτοιου είδους δεδομένων μέσα σε μια εφαρμογή, καθώς και τα μέρη της εφαρμογής των οποίων η εκτέλεση επηρεάζεται από αυτά.

Παρουσιάζουμε μία ανάλυση εξαρτήσεων, η οποία δεδομένων κάποιων εντολών εισόδου σε ένα πρόγραμμα Java, υπολογίζει μια υπερτίμηση των μερών του προγράμματος η εκτέλεση των οποίων μπορεί να επηρεαστεί από αυτή την είσοδο. Η κεντρική ιδέα είναι πως τα δεδομένα που δίνει ο χρήστης δεν θα πρέπει να επηρεάζουν κάποιο ευαίσθητο σημείο του προγράμματος, αλλά και πως, αντιστρόφως, ο χρήστης δεν θα πρέπει να λαμβάνει πληροφορίες που αφορούν ευαίσθητα δεδομένα.

Η ανάλυση γράφτηκε στην γλώσσα Datalog και βασίζεται στο Doop framework. Συγκεκριμένα, η λογική της ανάλυσης υλοποιείται σε περίπου 200 γραμμές Datalog, κάτι που είναι ενδεικτικό για τις δυνατότητες του Doop στην δημιουργία συμπαγών και εκφραστικών στατικών ανάλυσεων.

Θεματικη Περιοχη: Ανάλυση Εξαρτήσεων

Λεξεις Κλειδια: στατική ανάλυση, ασφάλεια, Java, Datalog, Doop

ACKNOWLEDGEMENTS

I would like to express my gratitude to prof. Yannis Smaragdakis for suggesting the idea of my thesis and for giving me the chance to work on such an interesting topic.

I would also like to thank both my supervisors, Yannis Smaragdakis and Georgios Balatsouras for their continuous support by sharing their expertise on the subject and providing help and suggestions throughout this project.

CONTENTS

LIST OF FIGURES

PREFACE

This thesis aims to provide the Doop framework with a dependence analysis. It has been developed as my undergraduate thesis since July 2015 at the Department of Informatics and Telecommunications of the University of Athens.

1. INTRODUCTION

An instruction can be said to be dependent on any other instruction that can affect its execution in any way. There are two major categories of such dependences: Data dependences (direct) and control dependences (indirect). For instance, an instruction that uses a variable defined by another instruction has a data dependence on that instruction, while an instruction that may or may not be executed depending on another instruction has a control dependence on that instruction.

Analyzing the way different instructions depend on each other is useful for many fields of computer science. For one, a compiler may reorder instructions that will not affect each other, or it may employ copy propagation to remove unnecessary data dependences if deemed beneficial. Another use would be the parallelization of two pieces of code that don't depend on each other.

The specific field our analysis is intended for is security. Given a set of untrusted input or output instructions, and a set of sensitive instructions, we try to detect any sort of interference between instructions in the two sets. There can also be pieces of code that serve as declassifiers, reducing the security level of some information (for example, an encryption or a sanitization method). Besides that, any instruction that may carry any information about some sensitive data will be considered a sensitive instruction.

Our approach leverages the concepts of the PDG [2] and the SDG [4] and builds on the points-to analysis provided by Doop in order to produce these graphs. The user then provides the untrusted instructions (sources) and our analysis traverses these graphs in order to find the parts of the code that are affected (sinks).

2. BACKGROUND

Datalog is a declarative logic-based programming language which is often used as a query language for deductive databases. Our analysis uses the Doop framework for Datalog [6], which provides a collection of points-to analyses (e.g. context insensitive, call-site sensitive, object sensitive). However, code built upon any such analysis, as in our case, can be oblivious to the exact choice of context (which is specified at runtime) and expressed using a generic API.

2.1 Points-To Analysis in Datalog

Doop's primary defining feature is its use of Datalog for its analyses and its explicit representation of relations as tables instead of Binary Decision Diagrams (BDDs) which have been considered necessary for scalable points-to analysis [24, 25].

Datalog is a great fit for the domain of program analysis and, as a consequence, has been extensively used both for low-level [26, 27] and for high-level [28, 29] analyses. The essence of Datalog is its ability to define recursive relations. Mutual recursion is the source of all complexity in program analysis. For a standard example, the logic for computing a call-graph depends on having points-to information for pointer expressions, which, in turn, requires a call-graph. Such recursive definitions are common in points-to analysis.

Doop currently uses a commercial Datalog engine, developed by LogicBlox Inc. A Datalog program consists of a series of rules, also known in Datalog semantics as the IDB (Intensional Database) rules, that are used to establish facts about derived relations from a conjunction of previously established facts. In the LB-Datalog syntax, the left arrow symbol (<-) separates the inferred fact (i.e. the head of the rule) from the previously established facts (i.e., the body of the rule).

Doop also uses the Soot framework [10] as a pre-processing step that takes the bytecode of a Java program and generates the input facts for an analysis, in the form of relations. Thanks to this, the original source is not needed, only the compiled classes. This is especially important, as libraries whose code is not public are used extensively. That kind of relations that directly derive from the input Java program, are also known in Datalog semantics as the EDB (Extensional Database) predicates. Our analysis also stores the information provided by the user, such as the sources of tainting and the declassifiers, as relations in the EDB.

Following the pre-processing step a simple pointer analysis can be expressed entirely in Datalog as a transitive closure computation:

```
VarPointsTo(?heap, ?var) <- AssignHeapAllocation(?heap, ?var).
VarPointsTo(?heap, ?to) <- Assign(?to, ?from), VarPointsTo(?heap, ?from).
```

This simple Datalog program consists of two rules. The first one states that, upon the allocation of a heap object, the variable that the heap object is assigned to may point to

that heap object. The second one states that, upon assigning the value of a variable to another variable, the second variable may also point to any heap object that the first one may point to. Particularly, the second rule states that if for some values of `?heap`, `?from` and `?to`, both `Assign(?to, ?from)` and `VarPointsTo(?heap, ?from)` are true, then it can be inferred that `VarPointsTo(?heap, ?to)` is also true.

2.2 Context Sensitivity in Doop

A very useful tool in improving the precision of an analysis is context sensitivity. That is the usage of context information for qualifying the program variables and possibly the heap objects, thereby collapsing the information produced by the analysis over all possible executions that result in the same context and separating that information for different contexts. The main flavors of context sensitivity are call-site-sensitivity, object-sensitivity and type-sensitivity. Context sensitivity may also be used with a certain depth, for example discerning between the last n call-sites [12].

Call-site sensitivity uses method call-sites as context elements, providing the analysis with a different context for every call-site in the program. This way, if for instance a method is called from several locations that result in different points-to sets for a variable inside that method, these points-to sets will be separated using the call-site as a qualifier, instead of considering the variable to be able to point to any of them during an invocation of the method, as would happen in a context-insensitive analysis.

Object-sensitivity qualifies contexts using the object allocation site the current method was invoked on (essentially the *this* object). Thus, the context of two invocations may differ even if their call-site is the same, if the object the method was called on was allocated in different places.

Type-sensitivity is similar to object-sensitivity but results in the merging of all allocation sites in methods declared in the same class. This yields a more scalable analysis at the cost of some precision.

```
void foo() {
    Object a = new Object();
    a = id(a);
    Object b = new Object();
    b = id(b);
}

Object id(Object o) {
    Object ret = o;
    return ret;
}
```

In the example above, the result of a context-insensitive analysis would contain a single points-to set for the variable `ret` in the method `id`, containing both the objects that `a` and

`b` point to. On the other hand, with a call-site-sensitive analysis, we would get a different points-to set for each invocation of `id`, one containing only the object pointed to by `a` and the other containing the object pointed to by `b`.

2.3 Slicing

Program slicing [13] is a well-known approach for detecting the impact of pieces of code to the rest of the program. A slice of a program with respect to a specific variable, called the slicing criterion, is the subset of instructions of the program that can affect the value of that variable at a specific point of the program. The calculation of an inter-procedural slice is based on the notion of the System Dependence Graph (SDG), which is described in the next chapter). This analysis essentially performs a forward slice, beginning from a set of instructions specified by the user and finding an overapproximation of the subset of the original program that may be affected either directly or indirectly by those instructions.

3. DEPENDENCE ANALYSIS

In this chapter, we describe the analysis that computes an over-approximation of the instructions affected by a set of starting instructions.

3.1 Program Dependence Graphs

We begin by describing the Program Dependence Graph (PDG), which is instrumental to our computing of intra-procedural dependences.

A Program Dependence Graph, first proposed by Ferrante et al. [2], is a representation of a single method in a computer program that makes explicit the data and control dependences between instructions in that method, while abstracting away details such as the ordering of the instructions in the method. The PDG is expressed as a graph, where the nodes are statements and the edges represent both the values each statement depends on and the control conditions on which the statement's execution depends. It is, effectively, a combination of a Data Dependence Graph and a Control Flow Graph. As the paths in a PDG express the information flows within a program, they are a great fit for our purposes.

Dependences are the results of two separate effects [3]. A data dependence formally exists between two statements if a variable appearing in one statement may have an incorrect value if the two statements are reversed. For example, in

```
a = 1
b = a+1
```

the second statement depends on the first one, as it defines the variable 'a' which is then used by the second statement. Reversing the two would lead into a different result. This is also called a direct dependence.

The other type of dependence is control dependence, and it arises when a statement controls another statement's execution. For example, in

```
if (a == 0)
   b = 1
```

while the second instruction is not directly dependent on the first instruction, as if it is executed the result will always be the same, it may or may not be executed depending on the first instruction. This type of dependence is called control or indirect dependence.

An example PDG is shown in figure 3.1.

```
if (a > 0)
{
  b = 1
}
else
{
  b = 2
}
c = b*2
```

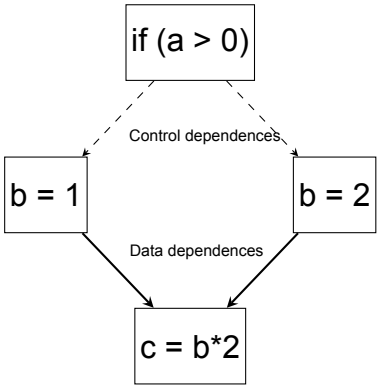

Figure 3.1: A simple piece of code and its PDG representation

3.2 Control Dependence

In order to formally define control dependence, we need the terms of Control Flow Graphs (CFG) and Post-Domination [5].

A Control Flow Graph is a directed graph augmented with a unique entry node `START` and a unique exit node `STOP`. Each node has at most 2 successors, and for each node `N` there is at least one path from `START` to `N` and from `N` to `STOP`. The nodes represent executable statements, while the edges represent potential control flow (figure 3.2).

```
while (i > 0) // S1
{
  if (cond(i)) // S2
  {
    x = x * 2 // S3
  }
  i-- // S4
}
y = 10 // S5
```

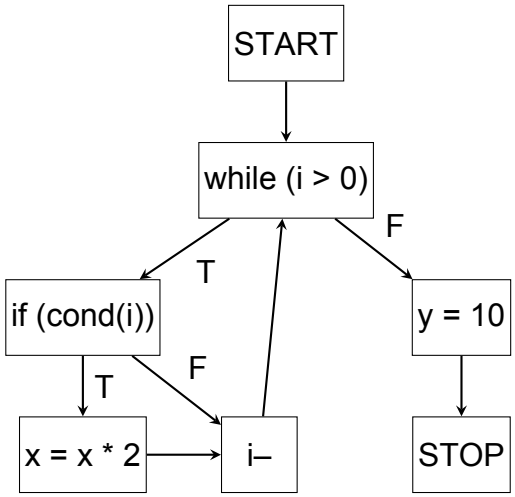

Figure 3.2: A simple CFG

We say that a node `V` is post-dominated by a node `W` if every directed path from `V` to `STOP` contains `W`. In the above example, the set of all pairs of statements `(A, B)` such that `A` is post-dominated by `B`, is `S = {(S1, S5), (S2, S1), (S2, S4), (S2, S5), (S3, S1), (S3, S4), (S3, S5), (S4, S1), (S4, S5)}`.

In our analysis, we compute post-domination in terms of basic blocks instead of statements for efficiency. This is possible because as control dependence concerns control statements and each basic block has at most one control statement by definition, we can think of each basic block as one big statement. We also use a CFG that doesn't contain the `START` and `STOP` nodes. We express post-domination in just 3 rules (figure 3.3), 2 of which compute non-post-domination:

1. Nothing post-dominates a block that is a leaf on the CFG, as the only node that could come after a leaf is the STOP node.
2. If a block `A` has an edge in the CFG to a block `B`, and block `B` is not post-dominated by a block `C`, then block `A` is also not post-dominated by block `C`, as there is a path from `A` through `B` to `STOP` that does not pass through `C`.
3. The third rule simply uses negation to finally calculate post-domination.

Intuitively, if a block `A` is not post-dominated by `B`, there is a path from `A` to `STOP` that doesn't contain `B`. We calculate for each block the set of other blocks that don't post-dominate it. For each CFG leaf, that set is all the other blocks, by definition of the CFG.

Inductively, we propagate this set to each block's ancestors. Whenever we propagate it to an ancestor, we remove than ancestor from the set and propagate the resulting set further. Finally, if we cannot reach a block `A` with another block `B` in our set by any path, this means that `A` is, in fact, post-dominated by `B`, as we didn't find a path from `STOP` to `A` that didn't contain `B`.

```
DoesNotPostDominate(?postDomCandidate, ?insn) <-
    BBHeadInMethod(?postDomCandidate, ?method),
    CFGLeaf(?insn, ?method),
    ?postDomCandidate != ?insn.

DoesNotPostDominate(?postDomCandidate, ?insn) <-
    DoesNotPostDominate(?postDomCandidate, ?otherInsn),
    MayPredecessorBBModuloThrow(?insn, ?otherInsn),
    ?insn != ?postDomCandidate.

PostDominates(?dominator, ?insn) <-
    SameMethodBBHeads(?dominator, ?insn),
    !DoesNotPostDominate(?dominator, ?insn).
```

Figure 3.3: The Datalog code for determining post-domination

The formal definition of control dependence [2] is that, given the nodes `X` and `Y` in a CFG,

`Y` is control dependent on `X` iff

- there exists a directed path from `X` to `Y` with any `Z` in `P` post-dominated by `Y` and
- `X` is not post-dominated by `Y`

For control dependences, we use both basic blocks and statements. Specifically, any control statement will be the last statement in its basic block and it will have 2 successors: The first statement of the next basic block and the control statement target, which will also be the first statement of a basic block. Since basic blocks are represented internally by their first statement, this is enough to express that a basic block has a control dependence on a statement.

Control dependence is expressed in 2 Datalog rules (figure 3.4). The first one forms the basis, finding all pairs of blocks and statements `(A, B)` where block `B` is a successor of statement `A` in the CFG but does not post-dominate it and is, therefore, control dependent on `A`.

The second rule expresses that if a block `A` post-dominates a block `B` which is control dependent on statement `C`, but `A` does not post-dominate `C`, then `A` is also control dependent on `C`. This inductive step follows from the formal definition, since `B` will only be control dependent on `C` if there is a path from `C` to `B` with `B` post-dominating all nodes in that path but not `C`. As post-domination is transitive, there will also be a path from `C` to `A` with `A` post-dominating all nodes in that path but not `C`.

```
IntraProceduralBlockControlDep(?nextBlock, ?prev) <-
    BasicBlockBegin(?nextBlock),
    MaySuccessorModuloThrow(?nextBlock, ?prev),
    BasicBlockHead[?prev] = ?prevBlockStart,
    !PostDominates(?nextBlock, ?prevBlockStart).

IntraProceduralBlockControlDep(?nextBlock, ?prev) <-
    PostDominates(?nextBlock, ?interm),
    BasicBlockHead[?prev] = ?prevBlockStart,
    !PostDominates(?nextBlock, ?prevBlockStart),
    IntraProceduralBlockControlDep(?interm, ?prev).
```

Figure 3.4: The Datalog code for determining intra-procedural control dependence

3.3 Intra-procedural Data Dependence

Data dependences can be further divided into true (read-after-write), anti (write-after-read) and output (write-after-write) dependences [1]. The latter two mostly concern the way the compiler can optimize the code: If two instructions write to the same variable, or if an instruction reads from a variable and then another instruction writes to the same variable, then the compiler should not attempt to reorder these instructions as the program will then behave differently. However, we do not care about these kind of dependences as much,

as we are simply trying to determine the places where information may flow.

Since we only care about data flow dependences, we define intra-procedural data dependences in terms of def-use chains [8].

A def-use chain identifies all possible uses of a variable, for each definition of that variable. This way, any information about the variables upon definition is directly propagated to all of their uses.

In Soot's representations for Java, each instruction may define up to one variable and use up to two. A definition of a variable is the statement that assigns a value to the variable, while a use is any statement that accesses the variable's value. Therefore, in the Data Dependence Graph, statements are nodes and def-use chains are the edges (figure 3.5).

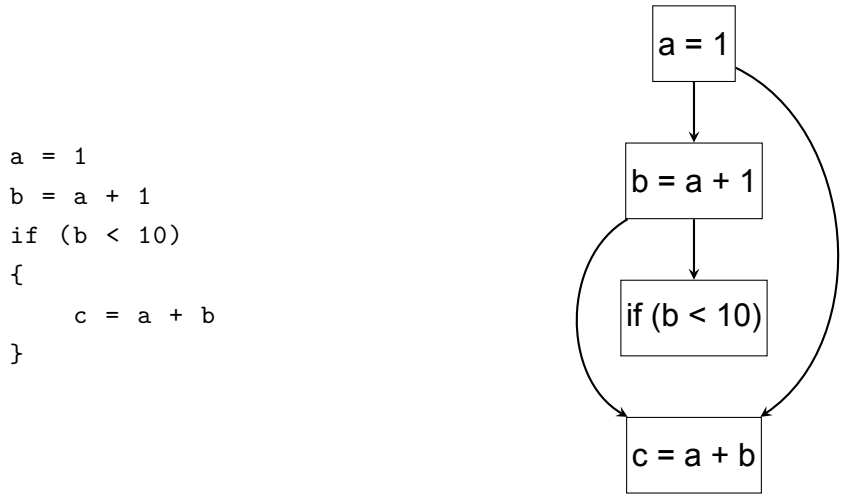

Figure 3.5: A simple DDG

A definition of a variable `X` is said to reach a use of that variable `X` if there exists a control flow path from the definition to the use without an intermediate definition of `X`.

Using Soot's Shimple representation [11], which uses an SSA form, this becomes easy to calculate. SSA introduces ϕ-nodes that combine def-use chains that have the same destination [7]. Using SSA, each variable is only defined once and may be used multiple times, so creating a DDG edge is simply a matter of finding the uses for each definition of a variable. The Datalog code for expressing this is simple as well (figure 3.6).

```
IntraProceduralDataDep(?prev, ?next) <-
    InstructionDefinesVar(?prev, ?var),
    InstructionUsesVar(?next, ?var).

/* Var is in instruction's use set */
InstructionUsesVar(?instruction, ?var) <-
    AssignLocal:From[?instruction] = ?var ;
    AssignOper:From(?instruction, ?var) ;
    AssignCast:From[?instruction] = ?var ;
    AssignInstanceOf:From[?instruction] = ?var ;
    If:Var(?instruction, ?var) ;
    VirtualMethodInvocation:Base[?instruction] = ?var ;
    Switch:Key[?instruction] = ?var ;
    ActualParam[_, ?instruction] = ?var ;
    ReturnNonvoid:Var[?instruction] = ?var ;
    Throw:Var[?instruction] = ?var ;
    LoadArrayIndex:Base[?instruction] = ?var ;
    StoreStaticField:From[?instruction] = ?var ;
    StoreInstanceField:From[?instruction] = ?var ;
    StoreArrayIndex:From[?instruction] = ?var ;
    ArrayInsnIndex[?instruction] = ?var.

/* Var is in instruction's def set */
InstructionDefinesVar(?instruction, ?var) <-
    AssignInstruction:To[?instruction] = ?var ;
    LoadArrayIndex:To(?instruction, ?var) ;
    StoreArrayIndex:Base(?instruction, ?var) ;
    LoadInstanceField:To[?instruction] =  ?var ;
    LoadStaticField:To[?instruction] =  ?var ;
    AssignReturnValue[?instruction] = ?var ;
    ReflectiveAssignReturnValue[?instruction] = ?var.
```

Figure 3.6: The Datalog code for determining intra-procedural data dependence

3.4 The System Dependence Graph

Having defined the CFG and DDG, the PDG is simply the union of the two (figure 3.7). By creating the PDG, we can traverse it beginning from any statement to perform a forward slice starting on that statement.

However, the PDG representation still only represents a single method. In order to represent an entire program, we may link the PDGs of each method into a central dependence graph, called the System Dependence Graph [4] (figure 3.8).

In the literature, to facilitate data dependence due to the passing of parameters to a method, actual-in and out and formal-in and out vertices are used. The formal-in vertices represent the values passed to the method invocation, while the actual-in vertices represent the parameters as used by the method. Similarly for return values. These in and out vertices are then connected as appropriate by edges on the SDG.

We skip this step and instead simply connect the definition of a variable passed as an argument to a method invocation and each use of that variable inside the method, essentially

```
IntraProceduralControlDep(?next, ?prev) <-
    IntraProceduralBlockControlDep(?nextBlock, ?prev),
    BasicBlockHead[?next] = ?nextBlock.

IntraProceduralDependencyBase(?next, ?prev) <-
    IntraProceduralDataDep(?prev, ?next) ;
    IntraProceduralControlDep(?next, ?prev).

/* Intra procedural dependency - base case */
IntraProceduralDependency(?prev, ?next) <-
    IntraProceduralDependencyBase(?next, ?prev).

/* Intra procedural dependency step */
IntraProceduralDependency(?prev, ?next) <-
    IntraProceduralDependency(?prev, ?inter),
    IntraProceduralDependencyBase(?next, ?inter).
```

Figure 3.7: The Datalog code for creating the PDG edges

forming an inter-procedural def-use chain for each argument.

As Java is an object-oriented language, we have to take into account polymorphism and inheritance whenever we encounter a method invocation. We have to be conservative for our analysis, so whenever we encounter a method invocation we have to consider that all of the methods that it might resolve to will be executed. This means that, if due to control flow a variable may point to different object types and a method is invoked upon that variable, we have to consider that all of the different possible implementations of that method will be executed. Thankfully, Doop takes care of this matter for us with its points-to analysis, which can tell us the possible allocation sites for each variable. Using Doop's `CallGraphEdge` predicate we can find, for each call site, all possible methods that can be called. In this way, we also leverage Doop's use of contexts for its points-to analysis.

3.5 Heap object dependences

In order to detect heap-carried dependences, we again use Doop's points-to analysis in order to determine which objects a statement may access. This part of the analysis is field-sensitive and context-sensitive, however it is not flow-sensitive across different methods. This means that, while a load may never come after a store during execution, that load may still be considered to depend on the store. While this introduces some loss of precision, many programs that would be considered for dependence analysis, such as web applications, are constantly looping and therefore this precision would make little difference. The downside is that we cannot consider any method to declassify an object field as we cannot be sure about the order of loads and stores.

Points-to analysis is also used for arrays, however array indices are not considered, so once a tainted variable is stored in an array, the entire array is considered tainted. This is because constant array indices are rarely used in Java, and we cannot reliably infer the

value of a variable index during runtime using static analysis.

Static object fields may also become tainted. Points-to analysis is not used here, as each static field has exactly one instance.

When any of the above heap-carried dependences is detected, an edge is added to the SDG from the statement that stores the possible tainted value to each statement that may load that value, similarly to how we added edges for def-use chains for intra-procedural data dependences. The difference, however, is that a def-use chain connects all the uses to the last definition, while heap-carried dependences are flow insensitive, and therefore all uses (loads) are connected to all definitions (stores). Besides the reasons stated above, another reason is that two methods that access the same parts of the heap may be executed in parallel, while a method invocation's statements will always be executed sequentially. Therefore, in order to be conservative, we don't attempt to use any kind of flow sensitivity on the inter-procedural level.

3.6 Reflection

The Doop framework also supports reflection to a certain degree. It fully supports reflective entities created using constant strings. Also, it tries to reason about the type of a reflective entity for reflective calls that use non-constant strings by trying to match constant substrings with potential class names, as well as using the type casts of reflective entities at their use sites as hints about their type. [9] Reflective calls and field operations are integrated into the appropriate relations, such as `VarPointsTo`, `CallGraphEdge` and `LoadHeapInstanceField`.

Because of this, our reflection benefits simply by enabling reflection handling, our analysis benefits by it. However, in order to handle e.g. static stores/loads and reflective call parameters, we have to introduce new rules. Therefore, our analysis should be able to soundly detect a large subset of reflection-introduced dependencies, such as those used in major Java frameworks.

3.7 Declassification

Often, a system needs some way to leak (i.e., declassify) some classified information, or process some untrusted data in a sensitive manner. For example, some user-supplied data may be considered sensitive up to the point that they reach some sanitization method, after which the data may be inserted into a database. This sanitization method may be considered a declassifier [14]. In our analysis, a user may supply the methods or statements that will be considered declassifiers. By doing that, the appropriate edges through which information flows to these methods or statements will be removed from the SDG, preventing the propagation of the tainting via these edges when the SDG is traversed. Note, however, that the issue with heap-carried dependences still applies: Since we can't determine the order in which heap loads and stores happen, a declassifier method won't

be able to un-taint a tainted field or array. Instead, a declassifier method may take some tainted arguments and return an untained value.

For an illustration, below is an example of how declassification affects one of our analysis rules. An instruction that loads a tainted heap location, represented by the `TaintedLoad` predicate, will only be considered dependent on one of the taint sources if it is not in a declassification method, or explicitly declared to be a secure instruction.

```
/* Mark dependent loads as dependent instructions across all contexts */
DependentInstruction(?ctx, ?insn) <-
    TaintedLoad(?ctx, ?insn),
    !SecureMethod(Instruction:Method[?insn]),
    !SecureInstructionInner(?insn).
```

Figure 3.8: An example of declassification

4. RESULTS

In order to evaluate our analysis, we ran it on the Securibench Micro 1.08 benchmark suite [15]. It contains test cases for different kinds of taint propagation, such as through the heap or through sessions, as well as a test group concerning declassifiers (Sanitizers group). In each test case, there are specific taint sources (usually a call to `HttpServletRequest.getParameter()`) as well as several sinks (usually calls to `System.out.println()`), for which it should be determined if the input from the source affects the output. The results are shown below.

Test Group	Detected	False Positives
Aliasing	12/12	0
Arrays	9/9	5
Basic	61/63	2
Collections	14/14	5
Data Structures	5/5	0
Factories	3/3	0
Inter	16/16	2
Pred	5/5	3
Reflection	4/4	0
Sanitizers	3/4	0
Session	3/3	1
Strong Update	1/1	2
Total	136/139	20

We detected a total of 136 out of 139 vulnerabilities (98%). The same Datalog rules were mostly used for the test cases. The biggest exceptions were the “Sanitizers” test cases, which each introduced a different declassification method and therefore needed additional rules.

Many of the false positives are due to the fact that we don’t tell apart different array indices, and heap-carried dependences are flow-insensitive.

We have also used the analysis on several of the larger programs in the Securibench benchmark suite [16] and we were able to track the parts of the code where user-supplied information, either through parameters, headers or cookies, were accessed during sensitive operations, such as writes to databases or files. This demonstrates the scalability of our analysis as the entire generation and traversal of the SDG, along with the underlying points-to analysis, took at most about 10 minutes. However, due to a lack of additional information on those programs and their vulnerabilities, we were uncertain of how many of these sinks were, in fact, security issues.

```
class Obj {
    int x;
    int y;
}
```

```
int add(int x, int y) {
    int r = x + y;
    return r;
}
```

```
void main() {
    Obj o = new Obj();
    input(o);
    int x = o.x;
    int y = o.y;
    int r = add(x, y);
    print(r);
}
```

```
void input(Obj o) {
    int x = readInt();
    int y = readInt();
    o.x = x;
    o.y = y;
}
```

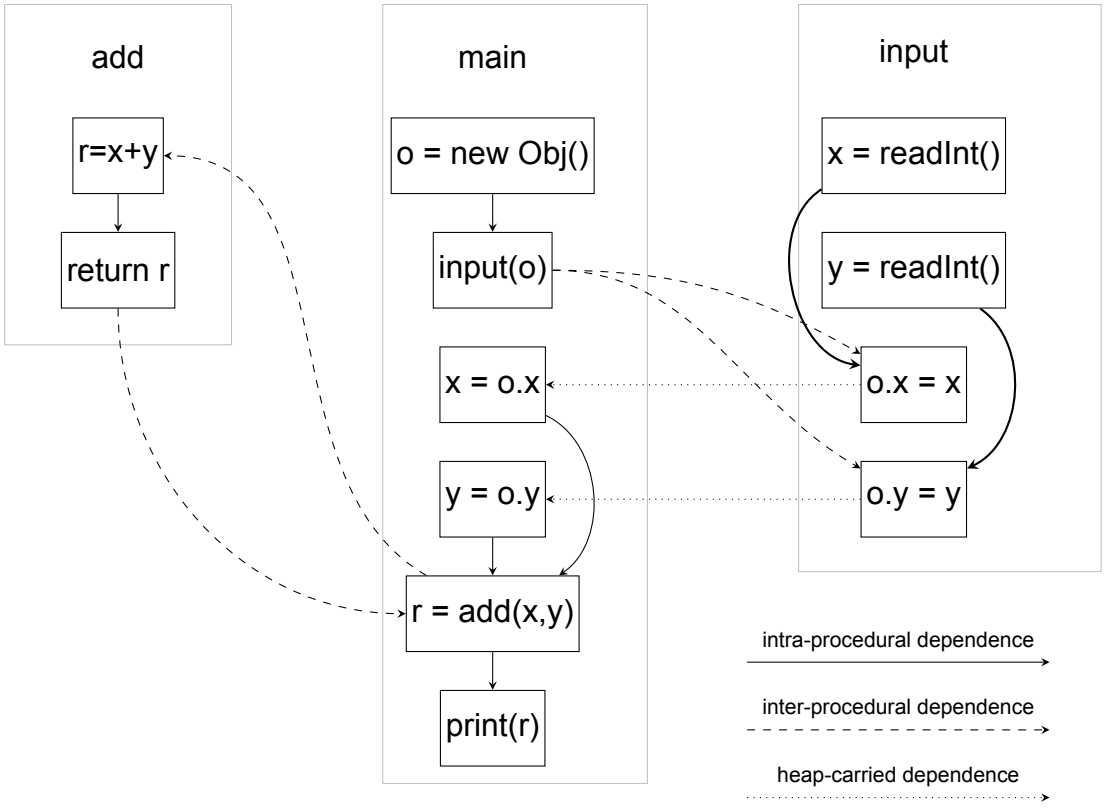

Figure 3.9: A simple program and its SDG representation

5. RELATED WORK

A similar tool that also uses the PDG is JOANA [17, 18], which comes as an add-on for the Eclipse IDE. It is object, flow and context-sensitive as well and is used for analyzing Java bytecode for noninterference and information flow control. It can also detect leaks that result from the interleaving of different threads. However, JOANA requires annotations to be present in the code during development. Like the other tools mentioned here, it does not handle reflection.

bddbddb [19] is a similar framework that uses Datalog for expressing context-sensitive analyses, however it uses BDDs instead of explicitly representing its relations as Doop does. While BDDs are a more advanced data structure than the b-trees used in Doop, thanks to Doop's declarative nature and explicit representation, with some simple enhancements a lot of redundancy is avoided, resulting in better performance and scalability [20].

Pidgin [21] uses PDGs for representing the flows of information found in Java bytecode and uses a custom PDG query language to allow users to express their own application-specific security policies, without interfering with the development of the program. Its PDG construction is based on the WALA framework.

FlowDroid [22] analyzes and detects tainted data flows within an Android apk file by using its knowledge about the Android application lifecycle. Its analysis is context, flow, field and object-sensitive and it can detect data flows from a set of sources to a set of sinks. Similarly to Doop, it uses the Soot framework and its Jimple IR [23].

6. CONCLUSIONS

Using Datalog and the Doop framework, we were able to express a compact and succinct dependence analysis, by using PDGs and SDGs to represent information and control flow within a program. In this way, we were able to combine the power of the PDG with the expressiveness of the Datalog language, as well as leverage the scalable points-to analysis offered by the Doop framework. By using the Doop framework, we were also able to decouple the implementation of the analysis from the context chosen for the points-to analysis. Also, as a result, our analysis is scalable to larger applications, thanks to Doop's explicit representation of relations instead of the traditionally used BDDs.

Users can easily define taint sources and declassifiers for the analysis and disable parts of the analysis such as the Control Flow Graph construction or change the context used. They can then query the results using the Datalog language in order to only show relevant information.

We used the Securibench Micro [15] framework to evaluate the analysis and compare it to similar methods in the field, and we were able to detect about 98% of the vulnerabilities it introduced with only a few false positives. The same set of rules was used for the benchmarks, except for the Sanitizer classes which also introduced some declassification methods.

ABBREVIATIONS

PDG	Program Dependence Graph
SDG	System Dependence Graph
CFG	Control Flow Graph
DDG	Data Dependence Graph
IR	Intermediate Representation
SSA	Static Single Assignment
BDD	Binary Decision Diagram
EDB	Extensional Database
IDB	Intentional Database

APPENDIX A: DEPENDENCE ANALYSIS CODE

```
#include "../2-object-sensitive+heap/analysis.logic"
#include "../../addons/cfg-analysis/rules.logic"



/* Made for optimizing TaintedLoad */
AllCtxDependentStoreHeapFld(?fld, ?hctx, ?heap) <-
    DependentInstruction(?ctx, ?dependent),
    StoreFldTo(?var, ?fld, ?dependent),
    VarPointsTo(?hctx, ?heap, ?ctx, ?var).

/* An instruction that loads a field that a dependent instruction wrote to
   */
TaintedLoad(?ctx, ?insn) <-
    AllCtxDependentStoreHeapFld(?fld, ?hctx, ?heap),
    LoadFldFrom(?insn, ?var2, ?fld),
    VarPointsTo(?hctx, ?heap, ?ctx, ?var2).

/* Similarly, find tainted static loads */
TaintedLoad(?ctx, ?insn) <-
    DependentInstruction(_, ?prev),
    FieldInstruction:Signature[?prev] = ?fld,
    StoreStaticField:Insn(?prev),
    LoadStaticField:Insn(?insn),
    FieldInstruction:Signature[?insn] = ?fld,
    ReachableContext(?ctx, Instruction:Method[?insn]).

/* Made for optimizing TaintedLoad */
LoadArrayHeapInsn(?ctx, ?insn, ?hctx, ?heap) <-
    LoadArrayIndex:Base[?insn] = ?var,
    VarPointsTo(?hctx, ?heap, ?ctx, ?var).

/* An instruction that loads data from an array that a dependent
   instruction wrote to */
TaintedLoad(?loadCtx, ?insn) <-
    DependentInstruction(?storeCtx, ?dependent),
    StoreArrayIndex:Base[?dependent] = ?var1,
    LoadArrayHeapInsn(?loadCtx, ?insn, ?hctx, ?heap),
    VarPointsTo(?hctx, ?heap, ?storeCtx, ?var1).

/* Find tainted reflective static loads */
TaintedLoad(?ctx, ?insn) <-
    DependentInstruction(?prevCtx, ?prevInsn),
    InstructionDefinesVar(?prevInsn, ?prevVar),
    ReflectiveStoreStaticField(?fld, ?prevCtx, ?prevVar),
    ReflectiveLoadStaticField(?ctx, ?nextVar, ?fld),
```

```
    InstructionUsesVar(?insn, ?nextVar).


/* Data dependency between the formal and actual param */
CallReturnDependency(?nextCtx, ?nextInsn, ?prevCtx, ?prevInsn) <-
    FormalParam[?paramIndex, ?meth] = ?formalParam,
    ActualParam[?paramIndex, ?callsite] = ?actParam,
    CallGraphEdge(?prevCtx, ?callsite, ?nextCtx, ?meth),
    InstructionDefinesVar(?prevInsn, ?actParam),
    InstructionUsesVar(?nextInsn, ?formalParam),
    !SecureMethod(?meth).

/* Data dependency between the formal and actual param in a reflective
   invocation */
CallReturnDependency(?nextCtx, ?nextInsn, ?prevCtx, ?prevInsn) <-
    FormalParam[_, ?meth] = ?formalParam,
    ReflectiveActualParams[?callsite] = ?actParams,
    ReflectiveCallGraphEdge(?prevCtx, ?callsite, ?nextCtx, ?meth),
    StoreArrayIndex:Base(?prevInsn, ?actParams),
    InstructionUsesVar(?nextInsn, ?formalParam),
    !SecureMethod(?meth).

/* Data dependency between the return instruction and the caller */
CallReturnDependency(?nextCtx, ?nextInsn, ?prevCtx, ?prevInsn) <-
    ReturnNonvoid:Insn(?prevInsn),
    Instruction:Method[?prevInsn] = ?meth,
    !SecureMethod(Instruction:Method[?nextInsn]),
    CallGraphEdge(?nextCtx, ?nextInsn, ?prevCtx, ?meth).


/* Dependency source */
DependentInstruction(?ctx, ?insn) <-
    Instruction:Method[?insn] = ?meth,
    ReachableContext(?ctx, ?meth),
    DependencySourceInstruction(?insn).

/* All kinds of intra-procedural dependency */
DependentInstruction(?ctx, ?insn)  <-
    DependentInstruction(?ctx, ?prevInsn),
    IntraProceduralDependencyOpt(?insn, ?prevInsn),
    !SecureInstructionInner(?insn).

/* Mark dependent loads as dependent instructions across all contexts */
DependentInstruction(?ctx, ?insn) <-
    TaintedLoad(?ctx, ?insn),
    !SecureMethod(Instruction:Method[?insn]),
    !SecureInstructionInner(?insn).
```

```
/**
 * If an instruction is inter-procedurally dependent on a dependent
    instruction
 * mark that instruction as dependent in its context
 */
DependentInstruction(?ctx, ?insn) <-
    DependentInstruction(?prevCtx, ?prevInsn),
    CallReturnDependency(?ctx, ?insn, ?prevCtx, ?prevInsn).

/**
 * If a method call has a control dependency on an instruction, every
    instruction
 * in that method is control dependent on that instruction
 */
DependentInstruction(?ctx, ?insn) <-
    DependentInstruction(?prevCtx, ?prevInsnControlDep),
    IntraProceduralControlDep(?prevInsn, ?prevInsnControlDep),
    CallGraphEdge(?prevCtx, ?prevInsn, ?ctx, ?meth),
    Instruction:Method[?insn] = ?meth.

SecureInstructionInner(?insn) <- SecureInstruction(?insn).

/* An instruction that calls a secure method is also secure */
SecureInstructionInner(?insn) <-
    CallGraphEdge(_, ?insn, _, ?meth),
    SecureMethod(?meth).

/* Insn stores to base.fld */
StoreFldTo(?base, ?fld, ?insn) <-
    FieldInstruction:Signature[?insn] = ?fld,
    StoreInstanceField:Base[?insn] = ?base.

/* Insn stores to base.fld by reflection */
StoreFldTo(?var, ?fld, ?insn) <-
    ReflectiveStoreField(?insn, ?fld, _, _),
    !FieldModifier("static", ?fld),
    java:lang:reflect:Field:set:base[?insn] = ?var.

/* Insn loads from base.fld */
LoadFldFrom(?insn, ?base, ?fld) <-
    FieldInstruction:Signature[?insn] = ?fld,
    LoadInstanceField:Base[?insn] = ?base.

/* Insn loads from base.fld by reflection */
LoadFldFrom(?insn, ?var, ?fld) <-
    ReflectiveLoadField(?insn, _, _, ?fld),
```

```
    !FieldModifier("static", ?fld),
    java:lang:reflect:Field:get:base[?insn] = ?var.

/* Var is in instruction's use set */
InstructionUsesVar(?instruction, ?var) <-
    AssignLocal:From[?instruction] = ?var ;
    AssignOper:From(?instruction, ?var) ;
    AssignCast:From[?instruction] = ?var ;
    AssignInstanceOf:From[?instruction] = ?var ;
    If:Var(?instruction, ?var) ;
    VirtualMethodInvocation:Base[?instruction] = ?var ;
    Switch:Key[?instruction] = ?var ;
    ActualParam[_, ?instruction] = ?var ;
    ReturnNonvoid:Var[?instruction] = ?var ;
    Throw:Var[?instruction] = ?var ;
    LoadArrayIndex:Base[?instruction] = ?var ;
    StoreStaticField:From[?instruction] = ?var ;
    StoreInstanceField:From[?instruction] = ?var ;
    StoreArrayIndex:From[?instruction] = ?var ;
    ArrayInsnIndex[?instruction] = ?var.

/* Var is in instruction's def set */
InstructionDefinesVar(?instruction, ?var) <-
    AssignInstruction:To[?instruction] = ?var ;
    LoadArrayIndex:To(?instruction, ?var) ;
    StoreArrayIndex:Base(?instruction, ?var) ;
    LoadInstanceField:To[?instruction] =  ?var ;
    LoadStaticField:To[?instruction] =  ?var ;
    AssignReturnValue[?instruction] = ?var ;
    ReflectiveAssignReturnValue[?instruction] = ?var.


/* An instruction is a CFG leaf if it ends with a throw */
CFGLeaf(?headInsn, ?method) <-
    Throw(?insn, _),
    BasicBlockHead[?insn] = ?headInsn,
    Instruction:Method[?headInsn] = ?method.

/* An instruction is a CFG leaf if it ends with a return */
CFGLeaf(?headInsn, ?method) <-
    ReturnInstruction(?insn),
    BasicBlockHead[?insn] = ?headInsn,
    Instruction:Method[?headInsn] = ?method.


/* Nothing post-dominates a CFG leaf */
DoesNotPostDominate(?postDomCandidate, ?insn) <-
```

```
    BBHeadInMethod(?postDomCandidate, ?method),
    CFGLeaf(?insn, ?method),
    ?postDomCandidate != ?insn.

/**
 * If A may come before B and C does not post-dominate B,
 * then C does not post-dominate A either, as after A the execution may go
    to B.
 * This is a lot more intuitive when one thinks that "DoesNotPostDominate(a
    ,b) =
 * exists path from b to some exit of the method, such that the path skips
    a".
 */
DoesNotPostDominate(?postDomCandidate, ?insn) <-
    DoesNotPostDominate(?postDomCandidate, ?otherInsn),
    MayPredecessorBBModuloThrow(?insn, ?otherInsn),
    ?insn != ?postDomCandidate.


/* If we can't prove A does not post-dominate B with the other rules, then
   A post-dominates B.
 * The definition is such that we always get reflexivity: PostDominates(x,x
    ).
 */
PostDominates(?dominator, ?insn) <-
    SameMethodBBHeads(?dominator, ?insn),
    !DoesNotPostDominate(?dominator, ?insn).


/* An intra procedural data dep exists when an instruction uses a var
   defined by another one */
IntraProceduralDataDep(?prev, ?next) <-
    InstructionDefinesVar(?prev, ?var),
    InstructionUsesVar(?next, ?var).


/* This block may follow an instruction, but it doesn't postdominate it.
 * Invariant of this relation: ?prev is the
 * last instruction of a BB, ?nextBlock is the first of a different one.
 */
IntraProceduralBlockControlDep(?nextBlock, ?prev) <-
    BasicBlockBegin(?nextBlock),
    MaySuccessorModuloThrow(?nextBlock, ?prev),
    BasicBlockHead[?prev] = ?prevBlockStart,
    !PostDominates(?nextBlock, ?prevBlockStart).

/**
```

```
 * This block postdominates a block that depends on an instruction
 * which this block doesn't postdominate
 */
IntraProceduralBlockControlDep(?nextBlock, ?prev) <-
    PostDominates(?nextBlock, ?interm),
    BasicBlockHead[?prev] = ?prevBlockStart,
    !PostDominates(?nextBlock, ?prevBlockStart),
    IntraProceduralBlockControlDep(?interm, ?prev).


/* Same as above, but per instruction */
IntraProceduralControlDep(?next, ?prev) <-
    IntraProceduralBlockControlDep(?nextBlock, ?prev),
    BasicBlockHead[?next] = ?nextBlock.


/* Intra procedural dependency base */
IntraProceduralDependencyBase(?next, ?prev) <-
    !SecureInstructionInner(?next),
    (IntraProceduralDataDep(?prev, ?next) ;
    IntraProceduralControlDep(?next, ?prev)).

/* Intra procedural dependency - base case */
IntraProceduralDependency(?prev, ?next) <-
    IntraProceduralDependencyBase(?next, ?prev).

/* Intra procedural dependency step */
IntraProceduralDependency(?prev, ?next) <-
    IntraProceduralDependency(?prev, ?inter),
    IntraProceduralDependencyBase(?next, ?inter).

/* Inverted for optimization */
IntraProceduralDependencyOpt(?next, ?prev) <- IntraProceduralDependency(?
   prev, ?next).
```

REFERENCES

[1] Jonas Skeppstedt, “An Introduction to the Theory of Optimizing Compilers”, Skeppberg AB, 2012.

[2] J. Ferrante, K. J. Ottenstein, J. D. Warren, “The program dependence graph and its use in optimization”, ACM Trans. on Programming Languages and Systems, 1987.

[3] Richard Johnson, Keshav Pingali, “Dependence-Based Program Analysis” ACM SIGPLAN’93 PLDI, June 1993.

[4] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs”, ACM Transactions on Programming Languages and Systems, vol. 12, pp. 26–60, January 1990.

[5] Frances E. Allen, “Control flow analysis”, SIGPLAN Notices, 5(7):1–19, July 1970. Proceedings of a Symposium on Compiler Optimization.

[6] Martin Bravenboer and Yannis Smaragdakis, “Strictly declarative specification of sophisticated points-to analyses”, OOPSLA ’09: 24th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications, New York, NY, USA, 2009. ACM.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”, ACM TOPLAS 13(4):451–490, Oct 1991.

[8] Alfred Aho, Jeffrey Ullman, Monica S. Lam, and Ravi Sethi, “Compilers, Principles, Techniques, and Tools”, Addison-Wesley, Reading, Mass, 1986.

[9] Y. Smaragdakis, G. Kastrinis, G. Balatsouras, and M. Bravenboer, “More Sound Static Handling of Java Reflection”, Tech. Rep., 2014.

[10] P. Lam, E. Bodden, O. Lhot´ak, and L. Hendren, “The soot framework for java program analysis: a retrospective”, In Cetus Users and Compiler Infastructure Workshop (CETUS 2011), 2011.

[11] Arni Einarsson and Janus Dam Nielsen, “A Survivor’s Guide to Java Program Analysis with Soot” BRICS, Department of Computer Science University of Aarhus, Denmark , 2008.

[12] Y. Smaragdakis, M. Bravenboer, and O. Lhotak, “Pick your contexts well: understanding object-sensitivity”, In Proc. 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2011.

[13] Weiser, M. “Programmers use slices when debugging”, Commun. ACM 25, 7 (1982), 446–452.

[14] A. Sabelfeld and D. Sands, “Declassification: Dimensions and principles”, Journal of Computer Security, 17:517–548, 2009.

[15] B. Livshits. Securibench Micro, 2006. `http://suif.stanford.edu/~livshits/work/securibench-micro/`.

[16] B. Livshits. Securibench, 2005. `http://suif.stanford.edu/~livshits/work/securibench/`.

[17] C. Hammer, “Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs”, PhD thesis, Universitat Karlsruhe (TH), Fak. f. Informatik, 2009.

[18] C. Hammer and G. Snelting, “Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs”, International Journal of Information Security, 2009.

[19] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, “Using Datalog and binary decision diagrams for program analysis”, In Proceedings of the Asian Symposium on Programming Languages and Systems, Nov. 2005.

[20] Y. Smaragdakis and M. Bravenboer, “Using Datalog for fast and easy program analysis”, In Datalog

2.0 Workshop, 2010.

[21] A. Johnson, L. Waye, S. Moore, and S. Chong, “Exploring and enforcing security guarantees via program dependence graphs”, In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 291–302, June 2015.

[22] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps”, In Proceedings of the 35th Conference on Programming Language Design and Implementation (PLDI ’14).

[23] Raja Vallee-Rai and Laurie J. Hendren, “Jimple: Simplifying Java Bytecode for Analyses and Transformations”, 1998.

[24] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam, “Using Datalog with binary decision diagrams for program analysis”, In Kwangkeun Yi, editor, APLAS, volume 3780 of Lecture Notes in Computer Science, pages 97–118. Springer, 2005.

[25] Ondrej Lhotak and Laurie Hendren, “Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation”, ACM Trans. Softw. Eng. Methodol., 18(1):1–53, 2008.

[26] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel, “Context-sensitive program analysis as database queries”, In PODS ’05: Proc. of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 1–12, New York, NY, USA, 2005. ACM.

[27] Thomas Reps, “Demand interprocedural program analysis using logic databases”, In R. Ramakrishnan, editor, Applications of Logic Databases, pages 163–196. Kluwer Academic Publishers, 1994.

[28] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini, “Defining and continuous checking of structural program dependencies”, In ICSE ’08: Proc. of the 30th int. conf. on Software engineering, pages 391–400, New York, NY, USA, 2008. ACM.

[29] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor, “Codequest: Scalable source code queries with Datalog”, In Proc. European Conf. on Object-Oriented Programming (ECOOP), pages 2–27. Spinger, 2006.