NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

UNDERGRADUATE THESIS

Security Analysis of the Java Library with Mock Objects

Konstantinos S. Triantafyllou

Supervisors: **Yannis Smaragdakis,** Associate Professor NKUA
George Kastrinis, Ph.D. Candidate NKUA

ATHENS

NOVEMBER 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανάλυση Ασφαλείας της Βιβλιοθήκης Java με Εικονικά Αντικείμενα

Κωνσταντίνος Σ. Τριανταφύλλου

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Αναπληρωτής Καθηγητής ΕΚΠΑ
Γιώργος Καστρίνης, Υποψήφιος Διδάκτορας ΕΚΠΑ

ΑΘΗΝΑ

ΝΟΕΜΒΡΙΟΣ 2016

UNDERGRADUATE THESIS

Security Analysis of the Java Library with Mock Objects

Konstantinos S. Triantafyllou
S.N.: 1115201100157

SUPERVISORS: **Yannis Smaragdakis,** Associate Professor NKUA
George Kastrinis, Ph.D. Candidate NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανάλυση Ασφαλείας της Βιβλιοθήκης Java με Εικονικά Αντικείμενα

Κωνσταντίνος Σ. Τριανταφύλλου
Α.Μ.: 1115201100157

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης,** Αναπληρωτής Καθηγητής ΕΚΠΑ
Γιώργος Καστρίνης, Υποψήφιος Διδάκτορας ΕΚΠΑ

ABSTRACT

Widely used platforms such as the Java Class Library have always attracted attackers' interest. One common exploitation scenario consists of an attacker triggering sensitive platform operations, thus letting him/her retrieve sensitive data from the results of those operations.

We present a static analysis which, considering as sources a subset of the public API of the Java Class Library, computes an over-approximation of the parts of the platform that could leak sensitive information, if triggered by an attacker. The main challenge in analysing a whole library, and not a specific program, is that we have to come up with ways to accurately fake attacker-created objects.

The analysis is based on the Doop framework which uses the Datalog language to declaratively specify pointer analysis algorithms. The analysis logic required just about 200 lines of Datalog code, which clearly shows the contribution of Doop in defining concise and expressive static analyses.

SUBJECT AREA: Static program analysis

KEYWORDS: static program analysis, security, java, mock objects, datalog

ΠΕΡΙΛΗΨΗ

Ευρέως χρησιμοποιούμενες πλατφόρμες όπως η Java Class Library έχουν πάντοτε στραμμένο πάνω τους το ενδιαφέρον των εισβολέων. Ένα κοινό σενάριο εκμετάλλευσης συνίσταται από τον επιτιθέμενο να ενεργοποιεί ευαίσθητες λειτουργίες της πλατφόρμας, επιτρέποντας του να ανακτήσει ευαίσθητα δεδομένα από τα αποτελέσματα αυτών των λειτουργιών.

Παρουσιάζουμε μια ανάλυση η οποία, θεωρώντας ως πηγές ένα υποσύνολο των μεθόδων του δημόσιου API της Java Class Library, υπολογίζει μια υπερεκτίμηση των μερών της πλατφόρμας που θα μπορούσαν να διαρρεύσουν ευαίσθητη πληροφορία αν ενεργοποιηθούν από έναν εισβολέα. Η κύρια πρόκληση της ανάλυσης μιας ολόκληρης βιβλιοθήκης, και όχι ενός συγκεκριμένου προγράμματος, είναι πως πρέπει να βρούμε τρόπους να απομιμηθούμε με ακρίβεια τα αντικείμενα που δημιουργεί ο επιτιθέμενος.

Η ανάλυση είναι βασισμένη στο Doop framework το οποίο χρησιμοποιεί τη γλώσσα Datalog για να προσδιορίζει δηλωτικά αλγορίθμους ανάλυσης δεικτών. Η λογική της ανάλυσης χρειάστηκε σχεδόν 200 γραμμές κώδικα Datalog, το οποίο είναι ενδεικτικό της συνεισφοράς του Doop στον καθορισμό περιεκτικών και εκφραστικών στατικών αναλύσεων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική ανάλυση προγράμματος

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: στατική ανάλυση προγράμματος, ασφάλεια, java, εικονικά αντικείμενα, datalog

To my family.

ACKNOWLEDGMENTS

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 George Kastrinis and also postdoctoral researcher Neville Grech and Ph.D. candidate George Balatsouras for their continuous support by sharing their expertise on the subject and providing help and suggestions throughout this project.

November 2016

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This thesis aims to research how mock objects can impact declarative static program analyses specified using the Doop framework. It was developed as my undergraduate thesis between March 2016 and October 2016 at the Department of Informatics and Telecommunications of the University of Athens.

1. INTRODUCTION

In the context of object-oriented programming, mock objects are typically created to simulate real objects and aid in the testing of other objects. Static analyses need mock objects in much the same way, as many times there is a need to mimic real objects (e.g. an object created by an attacker) in controlled ways.

This work presents an analysis in the field of security analyses and aims to find instances of the Confused Deputy Problem in the Java Class Library. In this type of vulnerability an attacker tricks the proxy possessing the necessary authority into carrying out sensitive operations on its behalf. The Java Class Library has been exploited multiple times through vulnerabilities utilizing instances of the Confused Deputy Problem.

In one common attack scenario the attacker manages to fool the platform into loading a class, which otherwise he would not have permission to load. Two of the most prominent exploits that employ this technique are described in the Common Vulnerabilities and Exposures Directory under identifiers 2012-4681 and 2013-0422. In these attacks, the attacker manages to retrieve package-restricted classes by controlling the input to a class-loading method call (e.g. Class.forName).

Defining an analysis that can find instances of the above attack scenario in the JCL requires the creation of mock objects, which mimic attacker-created objects and are provided as arguments for the source methods of the analysis. The Doop framework already implements algorithms for pointer analysis, which lets us focus mainly on the strategies of mock objects creation and their level of detail. These two variables can have significant impact on the precision and the scalability of an analysis with mock objects.

2. BACKGROUND

Datalog is a declarative logic programming language that is specifically designed to facilitate work with deductive databases. The most prominent difference between Datalog and Prolog, which heavily influenced the former, is that Datalog programs are guaranteed to terminate [14]. Our analysis depends on the Doop framework, which uses an extended version of Datalog that has been developed by Logiblox, Inc.

Doop implements a range of algorithms, including context-insensitive, call-site sensitive and object-sensitive points-to analyses. However, the modular representation of context in the framework can make code built upon any such analysis entirely oblivious to the exact choice of context (which is specified at runtime).

2.1 Points-To Analysis in Datalog

Doop's defining feature is the use of Datalog for its analyses and its explicit representation of relations as tables of a database, as it abstains from using Binary Decision Diagrams (BDDs), which have been considered necessary for scalable points-to analysis in the past [7, 8].

Datalog is a great fit for the domain of program analysis and, as a consequence, has been extensively used both for low-level [9, 10] and for high-level [11, 12] analyses. Its ability to define recursive relations solves the problem of mutual recursion, which 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's execution involves a pre-processing step, where the input facts for an analysis are generated with the help of the Soot framework [13]. Doop expects as input a Java program in bytecode form, which means that only the compiled classes and not the original source is needed, thus enabling the use of closed-source libraries. In Datalog semantics, the set of asserted facts for a program is called its EDB (Extensional Database). The generated relations that are directly produced from the input Java program, and any relation data added to the asserted facts by user defined rules, constitute the EDB predicates.

```
1 +Type:fqn(?type:?classname),
2 +ClassType(?type) <-
3   _ClassType(?classname).
```

Figure 2.1: Simple Datalog example for EDB rules

The program of Figure 2.1 consists of a single EDB rule that asserts (note the plus sign) two facts into the database. The rule creates a new `Type` entity `?type`, which is also added

to the `ClassType` relation, for each class type with name `?classname`. Each created entity is bound to an internal LB-Datalog ID.

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

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

Figure 2.2: Simple Datalog example for IDB rules

Computation in Datalog consists of monotonic logical inferences that repeatedly apply to produce more facts until a fixpoint is reached. The simple Datalog program of Figure 2.2 comprises two rules, which in Datalog semantics are known as IDB (Intensional Database) rules and are used to establish facts from a conjunction of already established facts. In the LB-Datalog syntax, a derivation rule's head (i.e. the inferred fact) is seperated by the rule's body (i.e. the previously established facts) by the left arrow symbol. For instance, the above first rule is the base case of the computation stating that, upon the assignment of an allocated heap object to a variable, this variable may point to that heap object. The second rule employs recursion to say that, a variable may point to any heap object another variable points to, if the value of the second variable is assigned to the first.

2.2 Context Sensitivity in Doop

For higher order (object-oriented and functional) languages, the key to enchancing analysis precision without sacrificing scalability has come to be context sensitivity [1, 8]. A context-sensitive analysis qualifies variables and abstract objects with context information: the analysis collapses information (e.g., "what objects this local variable can point to") over executions that map to the same context value, while separating executions that map to different contexts. Depending on the context's components, the main flavors of context sensitivity in modern pointer analysis are call-site sensitivity [3], object sensitivity [4, 5] and type sensitivity [6].

```
1  class A {
2    void bar() { ... }
3  }
4
5  Class B {
6    void foo(A a1, A a2) {
7      a1.bar();
8      a2.bar();
9    }
10 }
```

Figure 2.3: Simple Java example for context-sensitivity

Call-site sensitivity uses method call-sites as context elements. For instance, in our example, a call-site sensitive analysis will create two seperate points-to sets for the variables in method `bar`, one for each invocation on lines 7 and 8.

On the other hand, object sensitive analyses qualify contexts using the allocation site of the receiver object (i.e., the method's "this" object). In this way, the context of two method calls may differ even if they share the same call site, due to different allocation sites of the receiver objects. In the above example, an object-sensitive analysis will distinguish the calls to `bar` depending on the allocation site of the objects that variables `a1` and `a2` might point to.

Type sensitivity in Doop is analoguous to object sensitivity, yet types and not allocation sites are used to qualify contexts. Specifically, all the allocation sites in methods of the same class are merged. The goal of type sensitivity is to yield a more scalable analysis without sacrificing too much precision.

In [2], one can find a detailed description of the context-insensitive and context-sensitive analysis model in Doop. The main difference in the addition of context sensitivity is the use of constructors also known as skolem functions [16]. These functions are black boxes for the rest of the analysis and are used when we need to create a new calling context (or simply `Context`) for a variable abstraction, or a new heap context (or simply `HContext`) for a heap abstraction.

$$record : Allocation\ Site \times Context \rightarrow HContext$$
$$merge : Call\ Site \times HContext \times Context \rightarrow Context$$

The `record` function returns the creation context of an abstract object. Respectively, the `merge` function creates a calling context for every method invocation. Different flavors of context sensitivity are implemented by specifying variations of the record and merge functions.

Importantly, the addition of constructors by the LB-Datalog engine makes the language Turing-complete, i.e. programs are not guaranteed to terminate as in pure Datalog. Doop's context constructors are recursive: they return the same type of entities that they take as input, thus invalidating the property of polynomial execution. In order to restore this property we limit our attention to definitions of record and merge that create contexts in domains isomorphic to finite sets, bounded polynomially by the size of input.

3. SECURITY ANALYSIS

In this chapter, we describe the analysis that computes an over-approximation of the parts of the Java Class Library that could leak sensitive information if triggered by an attacker.

Our analysis works in a forward way: it begins at a set of sources and follows assignments until finding a vulnerable **Class.forName** method call, which we call a sink. Subsequently, the analysis tracks the returned values of sinks (which can be thought of as sources now) to check if they could leak and thus be read by an attacker.

3.1 Analysis Sources

In a real-world platform, such as the Java Class Library, that executes untrusted code, sources are all the public API methods that are callable by the untrusted code. In order to keep the analysis scalable, we regard as sources a subset of the public methods of the JCL.

We consider a public method as source if it is not abstract and has at least one parameter, whose type is interesting (or tainted, as we will call it) for our analysis. This logic is expressed in Datalog in four simple rules (Figure 3.1). As can be seen, we do not need to worry about the input facts (e.g. “which methods are public”) as Doop provides those by default. Only the facts about the tainted types need to be explicitly added into the EDB, as shown in Figure 3.2.

```
1  PublicMethod(?method) <-
2    MethodModifier("public", ?method).
3
4  AbstractMethod(?method) <-
5    MethodModifier("abstract", ?method).
6
7  InterestingMethod(?method) <-
8    PublicMethod(?method),
9    !AbstractMethod(?method).
10
11 SourceMethod(?method) <-
12   InterestingMethod(?method),
13   FormalParam[_, ?method] = ?formal,
14   Var:Type[?formal] = ?taintedtype,
15   TaintedType(?taintedtype).
```

Figure 3.1: Datalog code for determining source methods

We are aware that the decision to determine as sources the specific subset, and not all the public methods might raise some concern about the completeness of the analysis. However, experimental results of the analysis (Chapter 4) indicate that this should not be a major concern.

```
+TaintedType("java.lang.Object").
+TaintedType("java.lang.Object[]").
+TaintedType("java.lang.String").
+TaintedType("java.lang.String[]").
```

Figure 3.2: Datalog code for asserting tainted types

3.2 Mock Objects

The analysis presented in this work makes heavy use of mock objects in an attempt to mimic attacker-created objects. In theory, there should be a distinct mock object for every user-created one. In reality, in order to make the analysis scale better, we merge some of them, which naturally introduces some loss of precision.

The merge of mock objects can have many variants, depending on how one wants to handle the trade-off between scalability and precision of the analysis. For instance, an analysis that can afford some imprecision might create one mock object for every available class in the JCL and assign it where it is needed.

```
+MockHeap(?heap, ?type),
+Instruction:Value(?heap:?heapstr),
+HeapAllocation(?heap),
+HeapAllocation:Type[?heap] = ?type <-
  ClassType(?type),
  !TaintedType(?type),
  Type:fqn(?type:?typestr),
  ?heapstr = "mock-heap" + ?typestr.

+TaintedHeap(?type, ?method, ?heap),
+Instruction:Value(?heap:?heapstr),
+HeapAllocation(?heap),
+HeapAllocation:Type[?heap] = ?type <-
  MethodSignature:Value(?method:?m),
  FormalParam@previous[_, ?method] = ?formal,
  Var:Type@previous[?formal] = ?type,
  Type:fqn@previous(?type:?typestr),
  TaintedType(?type),
  ?heapstr = "tainted-heap-" + ?m + "-" + ?typestr.
```

Figure 3.3: Datalog code for creating mock objects

In our analysis, we follow a hybrid approach for mock object creation, which aims to keep the analysis scalable while maintaining good precision in specific areas, such as the tainted objects and their flow. The two EDB rules, which are responsible for the creation of the mock objects, are presented in Figure 3.3.

The first rule creates one object for every type of the JCL, except for the tainted ones.

The second one creates one object per parameter type, if the type is tainted (Figure 3.2), for every method of the JCL. The reason why the second rule does not focus only on the source methods, is that in this stage of the analysis (i.e. input facts generation) they have not been computed yet.

The objects are created using the a special type of predicated called refmode predicate `+Instruction:Value(?heap:?heapstr)`. In `+Instruction:Value(?heap:?heapstr)`, `?heap` is bound to an internal ID of the LB-Datalog engine, and `?heapstr` is the string that uniquely identifies the object. Note also the use of the `@previous` suffix, which references the population of a relation as it was immediately before the start of the transaction (i.e. at the end of the input facts generation). In this way, the infinite delta recursion, that could be introduced by the mutually recursive rules, is avoided.

The created mock objects (i.e. those stored in predicates *MockHeap* and *TaintedHeap*) need to be assigned to the parameters of the source methods as substitutes for attacker-created objects. Parameters of tainted type should point to the corresponding tainted heap objects. As for the rest of the parameters (those of non-tainted type) and the *this* variables, we choose to assign them simple mock objects (i.e. those created one per type). The Datalog code that implements this logic is presented in Figure 3.4.

```
1  VarPointsTo(?hctx, ?heap, ?ctx, ?formal) <-
2    SourceMethod(?method),
3    FormalParam[_, ?method] = ?formal,
4    Var:Type[?formal] = ?type,
5    TaintedHeap(?type, ?method, ?heap),
6    GlobalContext[] = ?ctx,
7    GlobalHContext[] = ?hctx.
8
9  VarPointsTo(?hctx, ?heap, ?ctx, ?this) <-
10   SourceMethod(?method),
11   ThisVar[?method] = ?this,
12   MethodSignature:DeclaringType[?method] = ?type,
13   MockHeap(?heap, ?type),
14   GlobalContext[] = ?ctx,
15   GlobalHContext[] = ?hctx.
16
17 VarPointsTo(?hctx, ?heap, ?ctx, ?formal) <-
18   SourceMethod(?method),
19   FormalParam[_, ?method] = ?formal,
20   Var:Type[?formal] = ?type,
21   MockHeap(?heap, ?type),
22   GlobalContext[] = ?ctx,
23   GlobalHContext[] = ?hctx.
```

Figure 3.4: Datalog code for assigning mock objects

Note that all the above `VarPointsTo` rules include in their body the relations `GlobalContext` and `GlobalHContext` to retrieve the global calling and heap context, respectively. These

two global contexts are used as the starting contexts of the analysis. Their definition depends on the flavor of context sensitivity that is used.

3.2.1 Mock Objects and Fields

Doop's analyses are field-sensitive, which means that they are able to distinguish different fields of the same abstract object, instead of lumping all fields together. Taking this into consideration, there are two main approaches to handle fields of mock objects.

1. Create mock objects for the parameters of every public method and let them flow and populate any mock object fields.
2. Manually set the fields of mock objects to point to other (or the same) mock objects.

In our analysis we follow the second approach as we want to avoid dealing with every public method of the JCL. Moreover, the abstract objects assigned to fields of mock objects and the mock objects themselves belong in the same set (i.e. the one per type mock objects). This can result in the creation of objects that are not so realistic. For instance, consider a `Node` object (Figure 3.5) which represents a node in a binary tree, but its `child1` and `child2` fields point to itself.

```
class BinaryTree {
   Node root;

   ...

   class Node {
      ...
      Node child1;
      Node child2;
      ...
   }
}
```

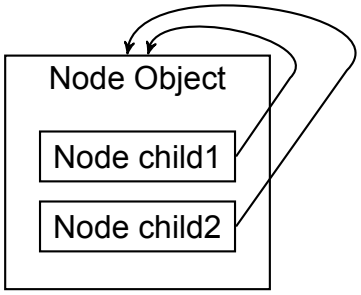

Figure 3.5: Example of mock object with fields

The assignment of mock objects to fields of mock objects requires one simple Datalog rule, which is presented in Figure 3.6. Note that Doop uses the relation InstanceFieldPoints, and not VarPointsTo, as the points-to relation for fields.

```
InstanceFieldPointsTo(?hctx, ?heap, ?signature, ?hctx, ?baseheap) <-
  MockHeap(?baseheap, ?basetype),
  ReferenceType(?basetype),
  FieldSignature:DeclaringClass[?signature] = ?basetype,
  FieldSignature:Type[?signature] = ?type,
  ReferenceType(?type),
  !FieldIsStatic(?signature),
  MockHeap(?heap, ?type),
  GlobalHContext[] = ?hctx.
```

Figure 3.6: Datalog code for assigning mock objects to mock object's fields

3.3 Analysis Sinks

In this kind of analysis, sinks are methods that perform sensitive operations. Our analysis focuses on finding vulnerable **Class.forName** method calls, and thus defines two sinks (Figure 3.7).

```
+SinkMethod(0, ?sig) <-
  MethodSignature:Value(?sig:"<java.lang.Class: java.lang.Class
      forName(java.lang.String)>") ;
  MethodSignature:Value(?sig:"<java.lang.Class: java.lang.Class
      forName(java.lang.String,boolean,java.lang.ClassLoader)>").
```

Figure 3.7: Datalog code for defining analysis sinks

Thanks to Doop's points-to analysis, which computes how heap objects flow intra- and inter-procedurally through the program, we only need to define two rules (Figure 3.8) to find tainted heaps that reach sinks. The first rule associates the first argument (0-th parameter, since forName is a static method) of a forName call with the invocation site and its context in computed relation `SinkVariable`. The second rule then uses `SinkVariable`: if the first argument `?var` of a forName call `?invocation` points to a tainted heap `?heap` from source method `?source`, then infer that an object can flow from `?source` to `?invocation`.

```
SinkVariable(?invocation, ?ctx, ?var) <-
  SinkMethod(?index, ?tomethod),
  CallGraphEdge(?ctx, ?invocation, _, ?tomethod),
  ActualParam[?index, ?invocation] = ?var.

TaintedHeapFromSourceFlowsToSink(?ctx, ?source, ?invocation) <-
  SinkVariable(?invocation, ?ctx, ?var),
  VarPointsTo(_, ?heap, ?ctx, ?var),
  TaintedHeap(_, ?source, ?heap).
```

Figure 3.8: Datalog code for computing objects flowing to sinks

3.3.1 Sanitization

Platforms such as the JCL often need to release sensitive information in a controlled manner. In our case, a user should be able to retrieve classes through calls to method **Class.forName**, as long as the requested classes do not belong in restricted packages.

The JCL implements many mechanisms to make Class.forName more secure [15]. One such mechanism is based on the idea of performing an additional security check that verifies package access prior to loading a class. This security check is performed by the method **checkPackageAccess (java.lang.String)** (of the **sun.reflect.misc.ReflectUtil** package), which raises an exception if the user does not have the authority to access the requested class.

For our analysis, user-created objects may be considered sensitive up to the point that they reach method **checkPackageAccess**: from there they are considered sanitized. The analysis computes the relation `SanitizedHeapFromSourceFlowsToSink`, which is a subset of the `TaintedHeapFromSourceFlowsToSink` relation (Figure 3.8) as it stores the sources of only the *sanitized* tainted heaps, that reach a sink. This relation is essential to compute which methods can leak sensitive information (i.e. objects returned by a non-sanitized call to forName). The Datalog code that implements the sanitization logic is presented in Figure 3.9.

```
1  SanitizedHeap(?heap) <-
2    SanitizationMethod(?index, ?tomethod),
3    CallGraphEdge(?ctx, ?invocation, _, ?tomethod),
4    ActualParam[?index, ?invocation] = ?var,
5    VarPointsTo(_, ?heap, ?ctx, ?var),
6    TaintedHeap(_, _, ?heap).
7
8  SanitizedHeapFromSourceFlowsToSink(?source, ?invocation) <-
9    SinkVariable(?invocation, ?ctx, ?var),
10   VarPointsTo(_, ?heap, ?ctx, ?var),
11   TaintedHeap(_, ?source, ?heap),
12   SanitizedHeap(?heap).
```

Figure 3.9: Datalog code for computing sanitized objects flowing to sinks

This treatment of sanitization is optimistic as our analysis is flow-insensitive, assuming that statements can execute in any order. Consequently, a heap that reaches a sanitization method for the first time only after flowing to a sink, is regarded as sanitized for this sink.

3.4 Reflection

Having defined as sink the **Class.forName** method, we extend our analysis to handle the two most common features of the Java Reflection API: creating a reflective object representing a class given a string (**Class.forName**) and creating a new object given a

class object (**Class.newInstance**).

```
+ForNameHeap(?heap, ?method),
+Instruction:Value(?heap:?heapstr),
+HeapAllocation(?heap),
+HeapAllocation:Type[?heap] = "java.lang.Class" <-
  MethodSignature:Value(?method:?m),
  ?heapstr = "for-name-heap-" + ?m.

+NewInstanceHeap(?heap, ?method),
+Instruction:Value(?heap:?heapstr),
+HeapAllocation(?heap),
+HeapAllocation:Type[?heap] = "java.lang.Object" <-
  MethodSignature:Value(?method:?m),
  ?heapstr = "new-instance-heap" + ?m.
```

Figure 3.10: Datalog code for creating reflective objects

Extending our analysis means adding two new input relations to the EDB, `ForNameHeap` and `NewInstanceHeap`, presented in Figure 3.10. These two relations store one abstract object of type `java.lang.Class` and one of type `java.lang.Object` for every method of the JCL, respectively.

The `java.lang.Class` and `java.lang.Object` objects created by the two EDB rules above, represent the returned objects of calls to `Class.forName` and `Class.newInstance`, respectively. We present the two inference rules that model reflection in Figure 3.11.

```
RecordMacro(?ctx, ?heap, ?hctx),
ClassObjectFromSink(?source, ?sink, ?heap),
VarPointsTo(?hctx, ?heap, ?ctx, ?to) <-
  TaintedHeapFromSourceFlowsToSink(?ctx, ?source, ?sink),
  ForNameHeap(?heap, ?source),
  AssignReturnValue[?sink] = ?to.

RecordMacro(?ctx, ?heap, ?hctx),
HeapFromSink(?source, ?sink, ?heap),
VarPointsTo(?hctx, ?heap, ?ctx, ?to) <-
  NewInstanceInvocation(_, ?to, ?from),
  VarPointsTo(_, ?classobject, ?ctx, ?from),
  ClassObjectFromSink(?source, ?sink, ?classobject),
  NewInstanceHeap(?heap, ?source).
```

Figure 3.11: Datalog code for assigning reflective objects

The first rule says that if a tainted heap from source method `?source` flows to **forName** sink `?sink`, whose result is assigned to local variable `?to`, and the invented class object corresponding to method `?source` is `?heap`, then `?to` should point to `?heap`. The second rule reads: if the receiver object, `?heap`, of a **newInstance** call is a class object returned

from sink `?sink` (to which flowed a tainted heap from source method `?source`), and the result of the call is assigned to variable `?to`, then `?to` should point to object `?instance` (which corresponds to method `?source`). The two rules use the `RecordMacro` to create a new heap context for the allocated object. Note the relations `ClassObjectFromSink` and `HeapFromSink`, which associate the object `?heap`, with the source method `?source` and the sink `?sink`, which it came from.

3.5 Leaks

In its final step our analysis computes which reflective objects, coming from a sink, can leak through public functions of the API. In order to achieve this, we only need to define a handful of rules (Figure 3.12), as Doop's points-to analysis takes care of how objects flow intra- and inter- procedurally throught the program.

The first rule computes the objects that may be returned by a public method. The helper relation `PublicMethodReturnsHeap` is used by the second and the third rule to jointly compute the relation `SourceToSinkToLeak`. A `SourceToSinkToLeak(?source, ?sink, ?leak)` fact is inferred when a reflective object `?heap`, created by a sink `?sink` to which a *non sanitized* tainted heap flowed from source method `?source`, is returned by a public method `?leak`. Note the additionaly computed relations `LeakClassObject` (2nd rule) and `LeakHeap` (3rd rule), which contain the leaked objects coming from **forName** and **newInstance** calls, respectively.

```
1  PublicMethodReturnsHeap(?method, ?heap) <-
2    PublicMethod(?method),
3    Instruction:Method[?x] = ?method,
4    ReturnNonvoid:Var[?x] = ?var,
5    VarPointsTo(_, ?heap, _, ?var).
6
7  LeakClassObject(?heap),
8  SourceToSinkToLeak(?source, ?sink, ?leak) <-
9    PublicMethodReturnsHeap(?leak, ?heap),
10   ClassObjectFromSink(?source, ?sink, ?heap),
11   !SanitizedHeapFromSourceFlowsToSink(?source, ?sink).
12
13 LeakHeap(?heap),
14 SourceToSinkToLeak(?source, ?sink, ?leak) <-
15   PublicMethodReturnsHeap(?leak, ?heap),
16   HeapFromSink(?source, ?sink, ?heap),
17   !SanitizedHeapFromSourceFlowsToSink(?source, ?sink).
```

Figure 3.12: Datalog code for computing leaked objects

3.6 String Operations

An important way of enhancing the empirical soundness of our analysis is via richer string flow. String concatenation in Java is typically done through **StringBuffer** or **StringBuilder** objects. The common concatenation operator, **+**, reduces to calls over such factory objects.

Figure 3.13 presents two helper relations needed to implement the analysis logic for string factories. The relation `StringFactoryVar` captures which variables are of a string factory type. The relation `StringFactoryVarPointsTo` is a subset of the `VarPointsTo` relation, containing only the variables that are of a string factory type.

```
1  StringFactoryVar(?var) <-
2    Var:Type[?var] = ?type,
3    StringFactoryType(?type).
4
5  StringFactoryVarPointsTo(?factoryHctx, ?factoryHeap, ?ctx, ?var) <-
6    VarPointsTo(?factoryHctx, ?factoryHeap, ?ctx, ?var),
7    StringFactoryVar(?var).
```

Figure 3.13: Datalog code for computing variables pointing to string factory objects

To evaluate whether tainted objects may flow into factory objects, we leverage the points-to analysis itself, pretending that an object flow into an **append** method and out of a **toString** method is equivalent to an assignment. In order to keep the analysis scalable, while treating the most common case, we require that the base variables of an **append** and a **toString** method call over the same factory object have the same *calling context* (i.e. are in the same method). The main logic for string operations is captured in three rules illustrated in Figure 3.14.

```
1  VarIsTaintedFromVar(?base, ?ctx, ?param) <-
2    VirtualMethodInvocation:SimpleName[?invocation] = "append",
3    VirtualMethodInvocation:Base[?invocation] = ?base,
4    StringFactoryVarPointsTo(_, _, ?ctx, ?base),
5    ActualParam[0, ?invocation] = ?param.
6
7  VarIsTaintedFromVar(?ret, ?ctx, ?base) <-
8    VirtualMethodInvocation:SimpleName[?invocation] = "toString",
9    VirtualMethodInvocation:Base[?invocation] = ?base,
10   StringFactoryVarPointsTo(_, _, ?ctx, ?base),
11   AssignReturnValue[?invocation] = ?ret.
12
13 VarPointsTo(?hctx, ?heap, ?ctx, ?to) <-
14   VarIsTaintedFromVar(?to, ?ctx, ?from),
15   TaintedHeap(_, _, ?heap),
16   VarPointsTo(?hctx, ?heap, ?ctx, ?from).
```

Figure 3.14: Datalog code for computing string flow through string factory objects

4. EXPERIMENTAL RESULTS

In this chapter, we present the evaluation of the analysis presented in Chapter 3 and comment on the experimental results.

One of the main goals of this work is to research how mock objects affect the percantage of the codebase that is analyzed. An empirical metric to quantify this is the number of tainted heaps that flow to a sink method. We define four mock object techniques:

1. *Tainted*. The analysis uses only mock objects of tainted type as arguments for the source methods.
2. *+This*. Extends the first technique by assigning mock objects to source methods' *this* variables.
3. *+Param*. Extends the second technique by using mock objects of non tainted type as arguments for the source methods.
4. *+Field*. Extends the third technique by assigning mock objects to fields of mock objects.

Figure 4.1 plots the results of our experiments, combining both the empirical metric (i.e. tainted heaps reaching sink methods) and the analysis time for each mock object technique. We run a context insensitive and a 2-type sensitive+heap analysis for each technique, so we use seperate bars (tainted heaps reaching sink methods) and lines (analysis time) for each one.

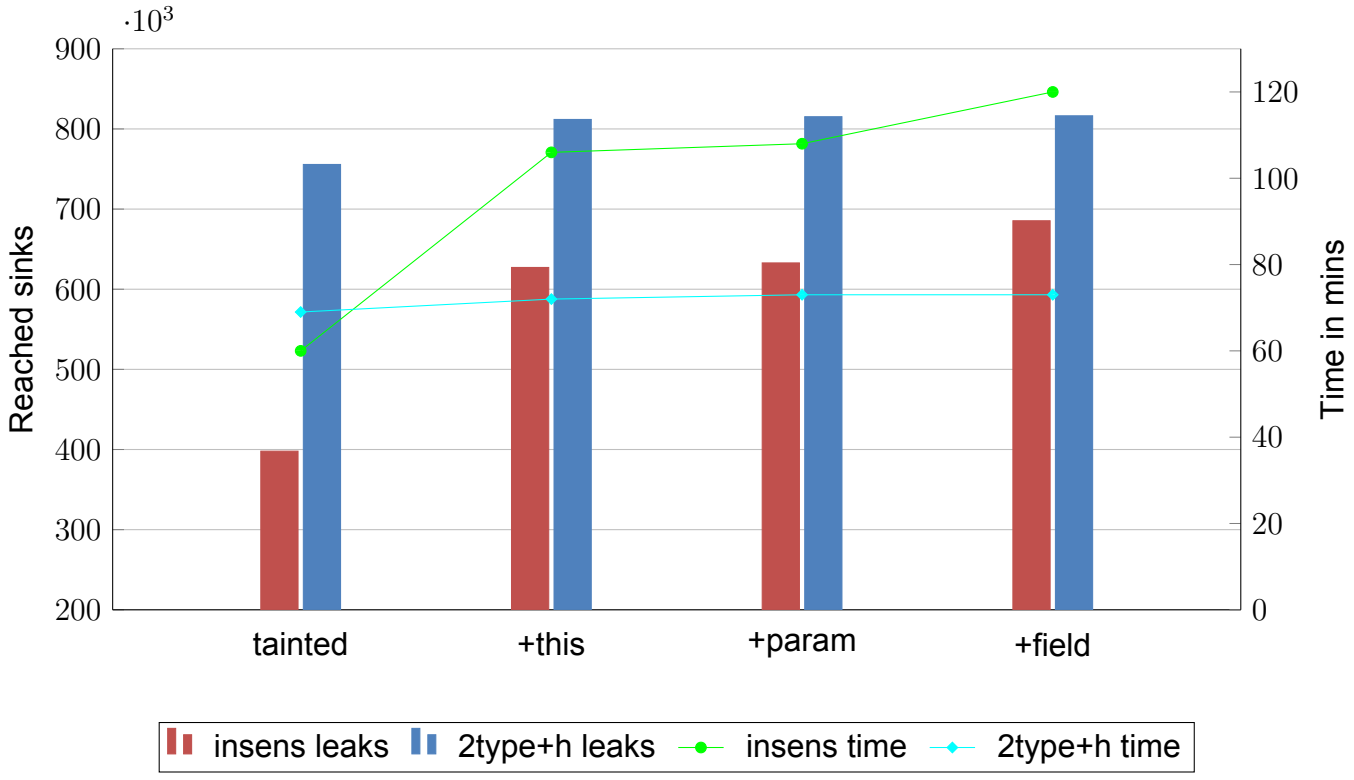

Figure 4.1: Reached Sinks and analysis time for JRE 7u45

The increase in analysis time (due to extra computations) and reached sinks that accompanies the addition of mock objects, shows the positive contribution of mock objects towards a more complete analysis. As can be seen, the *+this* technique is the most effective one, while the other two techniques are less impactful.

Table 4.1 presents more metrics that back up our claims. We are interested in the following metrics: `VarPointsTo` entries (vars), `TaintedHeapFromSourceFlowsToSink` (sinks), `LeakClassObject` (leak-co), `LeakHeap` (leak-h) and `SourceToSinkToLeak` (leak-path).

Table 4.1: Metrics concerning the effect of mock objects for JRE 7u45

		metrics				
analysis	technique	vars	sinks	leak-co	leak-h	leak-path
insens	tainted	372,278,126	397,945	10	10	5,492
	+this	667,646,329	627,159	15	15	11,895
	+param	689,780,680	632,819	15	15	12,122
	+field	783,466,088	685,423	15	15	13,322
2type+h	tainted	361,050,700	755,673	5	0	241
	+this	382,224,371	811,833	5	0	241
	+param	383,406,524	815,253	5	0	241
	+field	383,761,261	816,393	5	0	241

Regarding the leakage of sensitive information, the *insens +field* analysis reports 15 distinct class objects leaking from 13,322 public methods. Obviously, an insensitive analysis is not a good fit for our problem, as due to its lack in precision reports a huge number of false positives. As expected, the *2type+h +field* analysis proves more precise reporting five distinct class objects leaking from 241 public methods. In both cases, after inspecting the results it is almost certain that the reported vulnerabilities are false warnings. However, many of these are to be expected as our analysis is optimistic in some cases, and we do not implement any logic for the more advanced security mechanisms of the JCL.

The two exploits described in the Common Vulnerabilities and Exposures Directory under identifiers 2012-4681 and 2013-0422 are patched in JRE 7u45 and our analyses succeed in not reporting any false positives about them. In JRE 7u6 both exploits appear to be unpatched. Our analyses are able to report the leakage of restricted class objects in both cases.

Table 4.2: Metrics concerning the effect of mock objects for JRE 7u6

		metrics				
analysis	technique	vars	sinks	leak-co	leak-h	leak-path
insens	tainted	374,332,919	390,030	14	12	15,508
	+field	784,891,220	672,740	747	746	1,360,586
2type+h	tainted	485,497,672	693,223	737	725	12,635,463
	+field	513,125,141	749,299	782	768	14,374,448

Table 4.2 presents some metrics for JRE 7u6. It appears that the 2type+h analysis is less precise for this JRE version, as it reports more leaks. However, this happens mainly due to the imprecision of the insensitive analysis, which reports many flows to sinks as sanitized when it should not. Both our analysis yield imprecise results, reporting a huge number of false positives. Apparently, the sanitization method that we defined is not widely used in this JRE version, which employs other techniques to ensure authorized access to classes of restricted package. Our analysis could possible be extended to model more countermeasures.

5. CONCLUSIONS

Using Datalog and the Doop framework, we were able to express a compact and succinct security analysis aiming to find instances of the Confused Deputy Problem in the Java Class Library. A key feature of our analysis is the creation and use of mock objects that mimic attacker-created objects. In this way, we were able to analyse a sufficiently large part of the JCL codebase in about 1 to 2 hours. The scalability of the analysis results mainly from Doop's explicit representation of relations instead of the traditionally used BDDs. The Doop framework also made it trivial to decouple the implementation of the analysis from the context chosen for the points-to analysis.

Although our techniques yield positive results, further work is necessary to achieve good empirical soundness and scalability. The JCL employs many complicated security mechanisms that our analysis does not take into account, thus reporting false warnings in some cases. Furthermore, our research in mock objects for points-to analysis cannot be considered complete in any way. We hope that this work could lay the groundwork for future security analyses that make use of mock objects to achieve better results.

ACRONYMS AND ABBREVIATIONS

JCL	Java Class Library
JRE	Java Runtime Environment
EDB	Extensional Database
IDB	Intensional Database
API	Application Programming Interface
LB	LogicBlox Inc.
insens	context insensitive analysis
2type+h	2-type sensitive+heap analysis

ANNEX I: ANALYSIS INPUT RELATIONS

The domains of the analysis include: invocation sites, *I*; variables, *V*; heap object abstractions (i.e., allocation sites), *H*; calling contexts, *C*; heap contexts, *HC*; method signatures, *MS*; field signatures, *FS*; types, *T*; methods, *M*; natural numbers, *N*, and strings. Our analysis takes as input the relations presented in Figure A.1, which are part of the points-to analysis of Doop.

The reader might note that some of our analysis rules contain predicates of the special form `PredicateName[arg1, arg2] = value`. These are functional predicates that map the values found between the square brackets to the value right of the equals sign. Note that if we forget about the functional property, the predicate `PredicateName[arg1, arg2] = value` is equivalent to `PredicateName(value, arg1, arg2)`.

Instruction:Value(*i: I, s: string*): instruction *i* is uniquely identified by the string *s*.
Instruction:Method(*m: M, i: I*): instruction *i* is in method *m*.
ReturnNonVoid:Var(*v: V, i: I*): instruction *i* returns local var *v*.
MethodSignature:Value(*m: M, s: MS*): *m* represents the method for method signature *s*.
MethodModifier(*mod: string, m: M*): method's *m* definition has a *mod* modifier.
ThisVar(*v: V, m: M*): var *v* is the this var of method *m*.
FormalParam(*v: V, n: N, m: M*): the *n*-th formal parameter of method *m* is var *v*.
ActualParam(*v: V, n: N, i: I*): at invocation *i*, the *n*-th parameter is local var *v*.
AssignReturnValue(*v: V, i: I*): at invocation *i*, the value returned is assigned to local var *v*.
Type:fqn(*t: T, s: string*): type *t* is uniquely identified by the string *s*.
Var:Type(*t: T, v: V*): var *v* has type *t*.
HeapAllocation:Type(*t: T, h: H*): object *h* has type *t*.
FieldSignature:DeclaringClass(*s: FS, t: T*): type *t* has a field with signature *s*.
FieldSignature:Type(*s: FS, t: T*): field with signature *s* has type *t*.
VirtualMethodInvocation:SimpleName(*s: string, i: I*): instruction *i* is an invocation to a method with name *s*.
VirtualMethodInvocation:Base(*v: V, i: I*): instruction *i* is an invocation whose receiver object is pointed to by local var *v*.

Figure A.1: Analysis input relations

ANNEX II: SECURITY ANALYSIS CODE

```
#include "../2-type-sensitive+heap/analysis.logic"

GlobalContext[] = ?ctx,
ContextFromRealContext[?immType, ?immType] = ?ctx,
Context(?ctx) <-
  ImmutableTypeValue[] = ?immType.

GlobalHContext[] = ?hctx,
RecordImmutableMacro(_, _, ?hctx) <- .

FieldIsStatic(?sig) <-
  FieldModifier("static", ?sig).

PublicMethod(?method) <-
  MethodModifier("public", ?method).

AbstractMethod(?method) <-
  MethodModifier("abstract", ?method).

InterestingMethod(?method) <-
  PublicMethod(?method),
  !AbstractMethod(?method).

SourceMethod(?method) <-
  InterestingMethod(?method),
  FormalParam[_, ?method] = ?formal,
  Var:Type[?formal] = ?taintedtype,
  TaintedType(?taintedtype).

ReachableContext(?ctx, ?method) <-
  SourceMethod(?method),
  GlobalContext[] = ?ctx.

/* 1 tainted heap per method */
OptTaintedHeap(?heap, ?type, ?method) <-
  TaintedHeap(?type, ?method, ?heap).

VarPointsTo(?hctx, ?heap, ?ctx, ?formal) <-
  SourceMethod(?method),
  FormalParam[_, ?method] = ?formal,
  Var:Type[?formal] = ?type,
```

```
42    OptTaintedHeap(?heap, ?type, ?method),
43    GlobalContext[] = ?ctx,
44    GlobalHContext[] = ?hctx.
45
46  /* 1 mock heap per type */
47  VarPointsTo(?hctx, ?heap, ?ctx, ?this) <-
48    SourceMethod(?method),
49    ThisVar[?method] = ?this,
50    MethodSignature:DeclaringType[?method] = ?type,
51    MockHeap(?heap, ?type),
52    GlobalContext[] = ?ctx,
53    GlobalHContext[] = ?hctx.
54
55  VarPointsTo(?hctx, ?heap, ?ctx, ?formal) <-
56    SourceMethod(?method),
57    FormalParam[_, ?method] = ?formal,
58    Var:Type[?formal] = ?type,
59    MockHeap(?heap, ?type),
60    GlobalContext[] = ?ctx,
61    GlobalHContext[] = ?hctx.
62
63  InstanceFieldPointsTo(?hctx, ?heap, ?signature, ?hctx, ?baseheap) <-
64    MockHeap(?baseheap, ?basetype),
65    ReferenceType(?basetype),
66    FieldSignature:DeclaringClass[?signature] = ?basetype,
67    FieldSignature:Type[?signature] = ?type,
68    ReferenceType(?type),
69    !FieldIsStatic(?signature),
70    MockHeap(?heap, ?type),
71    GlobalHContext[] = ?hctx.
72
73  SinkVariable(?invocation, ?ctx, ?var) <-
74    SinkMethod(?index, ?tomethod),
75    CallGraphEdge(?ctx, ?invocation, _, ?tomethod),
76    ActualParam[?index, ?invocation] = ?var.
77
78  TaintedHeapFromSourceFlowsToSink(?ctx, ?source, ?invocation) <-
79    SinkVariable(?invocation, ?ctx, ?var),
80    VarPointsTo(_, ?heap, ?ctx, ?var),
81    TaintedHeap(_, ?source, ?heap).
82
83  SanitizedHeap(?heap) <-
84    SanitizationMethod(?index, ?tomethod),
```

```
85    CallGraphEdge(?ctx, ?invocation, _, ?tomethod),
86    ActualParam[?index, ?invocation] = ?var,
87    VarPointsTo(_, ?heap, ?ctx, ?var),
88    TaintedHeap(_, _, ?heap).
89
90  SanitizedHeapFromSourceFlowsToSink(?source, ?invocation) <-
91    SinkVariable(?invocation, ?ctx, ?var),
92    VarPointsTo(_, ?heap, ?ctx, ?var),
93    TaintedHeap(_, ?source, ?heap),
94    SanitizedHeap(?heap).
95
96  RecordMacro(?ctx, ?heap, ?hctx),
97  ClassObjectFromSink(?source, ?sink, ?heap),
98  VarPointsTo(?hctx, ?heap, ?ctx, ?to) <-
99    TaintedHeapFromSourceFlowsToSink(?ctx, ?source, ?sink),
100   ForNameHeap(?heap, ?source),
101   AssignReturnValue[?sink] = ?to.
102
103 RecordMacro(?ctx, ?heap, ?hctx),
104 HeapFromSink(?source, ?sink, ?heap),
105 VarPointsTo(?hctx, ?heap, ?ctx, ?to) <-
106   NewInstanceInvocation(_, ?to, ?from),
107   VarPointsTo(_, ?classobject, ?ctx, ?from),
108   ClassObjectFromSink(?source, ?sink, ?classobject),
109   NewInstanceHeap(?heap, ?source).
110
111 PublicMethodReturnsHeap(?method, ?heap) <-
112   PublicMethod(?method),
113   Instruction:Method[?x] = ?method,
114   ReturnNonvoid:Var[?x] = ?var,
115   VarPointsTo(_, ?heap, _, ?var).
116
117 LeakClassObject(?source, ?sink),
118 SourceToSinkToLeak(?source, ?sink, ?leak) <-
119   PublicMethodReturnsHeap(?leak, ?heap),
120   ClassObjectFromSink(?source, ?sink, ?heap),
121   !SanitizedHeapFromSourceFlowsToSink(?source, ?sink).
122
123 LeakHeap(?source, ?sink),
124 SourceToSinkToLeak(?source, ?sink, ?leak) <-
125   PublicMethodReturnsHeap(?leak, ?heap),
126   HeapFromSink(?source, ?sink, ?heap),
127   !SanitizedHeapFromSourceFlowsToSink(?source, ?sink).
```

```
128
129 /* String operations */
130 StringFactoryVar(?var) <-
131   Var:Type[?var] = ?type,
132   StringFactoryType(?type).
133
134 StringFactoryVarPointsTo(?factoryHctx, ?factoryHeap, ?ctx, ?var) <-
135   VarPointsTo(?factoryHctx, ?factoryHeap, ?ctx, ?var),
136   StringFactoryVar(?var).
137
138 VarIsTaintedFromVar(?base, ?ctx, ?param) <-
139   VirtualMethodInvocation:SimpleName[?invocation] = "append",
140   VirtualMethodInvocation:Base[?invocation] = ?base,
141   StringFactoryVarPointsTo(_, _, ?ctx, ?base),
142   ActualParam[0, ?invocation] = ?param.
143
144 VarIsTaintedFromVar(?ret, ?ctx, ?base) <-
145   VirtualMethodInvocation:SimpleName[?invocation] = "toString",
146   VirtualMethodInvocation:Base[?invocation] = ?base,
147   StringFactoryVarPointsTo(_, _, ?ctx, ?base),
148   AssignReturnValue[?invocation] = ?ret.
149
150 VarPointsTo(?hctx, ?heap, ?ctx, ?to) <-
151   VarIsTaintedFromVar(?to, ?ctx, ?from),
152   TaintedHeap(_, _, ?heap),
153   VarPointsTo(?hctx, ?heap, ?ctx, ?from).
```

```
#include "../2-type-sensitive+heap/delta.logic"

/* Methods and invocations */
+SinkMethod(0, ?sig) <-
  MethodSignature:Value(?sig:"<java.lang.Class: java.lang.Class
      forName(java.lang.String)>") ;
  MethodSignature:Value(?sig:"<java.lang.Class: java.lang.Class
      forName(java.lang.String,boolean,java.lang.ClassLoader)>").

+SanitizationMethod(0, ?sig) <-
  MethodSignature:Value(?sig:"<sun.reflect.misc.ReflectUtil: void
      checkPackageAccess(java.lang.String)>").

+NewInstanceInvocation(?invocation, ?to, ?from) <-
  MethodSignature:Value(?sig:"<java.lang.Class: java.lang.Object
      newInstance()>"),
  MethodInvocation:Signature[?invocation] = ?sig,
  AssignReturnValue[?invocation] = ?to,
  VirtualMethodInvocation:Base[?invocation] = ?from.


/* Tainted types */
+TaintedType("java.lang.Object").
+TaintedType("java.lang.Object[]").
+TaintedType("java.lang.String").
+TaintedType("java.lang.String[]").


/* Mock objects */

/* 1 mock heap per type */
+MockHeap(?heap, ?type),
+Instruction:Value(?heap:?heapstr),
+HeapAllocation(?heap),
+HeapAllocation:Type[?heap] = ?type <-
  ClassType(?type),
  !TaintedType(?type),
  Type:fqn(?type:?typestr),
  ?heapstr = "mock-" + ?typestr.

/* 1 tainted mock heap per method's parameter type
 *
```

```
39  * The input facts have already been added in a previous transaction, so we use
40  * the @previous suffix to avoid delta recursion.
41  */
42 +TaintedHeap(?type, ?method, ?heap),
43 +Instruction:Value(?heap:?heapstr),
44 +HeapAllocation(?heap),
45 +HeapAllocation:Type[?heap] = ?type <-
46   MethodSignature:Value(?method:?m),
47   FormalParam@previous[_, ?method] = ?formal,
48   Var:Type@previous[?formal] = ?type,
49   Type:fqn@previous(?type:?typestr),
50   TaintedType(?type),
51   ?heapstr = "tainted-heap-" + ?m + "-" + ?typestr.
52
53 /* We create 2 heap abstractions, one of type java.lang.Class and one of type
54  * java.lang.Object, for every available method.
55  */
56 +ForNameHeap(?heap, ?method),
57 +Instruction:Value(?heap:?heapstr),
58 +HeapAllocation(?heap),
59 +HeapAllocation:Type[?heap] = "java.lang.Class" <-
60   MethodSignature:Value(?method:?m),
61   ?heapstr = "for-name-sink-heap-" + ?m.
62
63 +NewInstanceHeap(?heap, ?method),
64 +Instruction:Value(?heap:?heapstr),
65 +HeapAllocation(?heap),
66 +HeapAllocation:Type[?heap] = "java.lang.Object" <-
67   MethodSignature:Value(?method:?m),
68   ?heapstr = "new-instance-sink-heap" + ?m.
69
70
71 /* String factory types */
72 +StringFactoryType("java.lang.StringBuffer").
73 +StringFactoryType("java.lang.StringBuilder").
```

REFERENCES

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

[2] Yannis Smaragdakis and George Balatsouras (2015), ”Pointer Analysis”, Foundations and Trends® in Programming Languages: Vol. 2: No. 1, pp 1-69.

[3] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, Program flow analysis: theory and applications, chapter 7, pages 189–233. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981. ISBN 0137296819.

[4] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In Proc. of the 2002 International Symp. on Software Testing and Analysis, ISSTA ’02, pages 1–11, New York, NY, USA, 2002. ACM. ISBN 1-58113-562-9.

[5] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. ACM Trans. Softw. Eng. Methodol., 14(1):1–41, 2005. ISSN 1049-331X.

[6] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. Pick your contexts well: Understanding object-sensitivity. In Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL ’11, pages 17–30, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0.

[7] 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.

[8] Onrej Lhotak. Program Analysis using Binary Decision Diagrams. PhD thesis, McGill University, January 2006.

[9] 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.

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

[11] 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.

[12] 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.

[13] P. Lam, E. Bodden, O. Lhotak, and L. Hendren, "The soot framework for java program analysis: a retrospective", In Cetus Users and Compiler Infastructure Workshop (CETUS 2011), 2011.

[14] S. Ceri, G. Gottlob and L. Tanca, "What you always wanted to know about Datalog (and never dared to ask)", IEEE Transactions on Knowledge and Data Engineering, vol. 1, no. 1, pp. 146-166, 1989.

[15] Security Explorations. Security Vulnerabilities in Java SE, Technical Report, Ver 1.0.2, 2012.

[16] H. B. Enderton. A Mathematical Introduction to Logic. Academic Press, 1st edition, 1972.