INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

Static Security Analysis for Java Programs

Author:
Akash GARG

Supervisor:
Prof. A. SANYAL
Prof. A. KARKARE

November 26, 2016

Declaration

I declare that this written submission represents my ideas in my own words and where other's ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Date: 26th November, 2016 — Akash Garg

Place: IIT Bombay, Mumbai — Roll No: 130070060

Acknowledgement

I would like to express my gratitude to Prof. Amitabh Sanyal and Prof. Amey Karkare for providing me nice ideas to work upon. Not only did they guided me during the course of the project but also encouraged me to discuss ideas with them, corrected my mistakes and helped me learn a lot of cool stuff. Without those discussion it would have been impossible to complete the project in such a manner.

I would also like to thank Prof. Supratik , Prof. Mythlli and Mr. Rishikesh for allowing me to use their computing resources.

Simply put it would have been impossible to complete the project without all the helps I have received from so many people.

Akash Garg

Contents

Chapter 1

Introduction [1]

Java has a security architecture which intends to protects the client and server system from malicious code. The source of such malicious code can vary widely. It could be installed dynamically(executing online downloaded applets) or could be statically configured(added to the system by a user). Java applet can be downloaded from internet via a web browser and can be uploaded onto a server using Remote Method Invocation. This dynamic code can do lot of damage to the system and therefore it is important to check the code before actually executing it. In this report we focus on the automated checking of Java code for certain types of potential security threats.

Java Security Model

Java uses sandboxing for protecting system, from external applications. When Java allows a program to be hosted on a system, it provides an environment where the program can play. This environment has certain bound which limits the tasks the applet can do. A typical machine has access to lot of resources. This sandbox prevents the applet from accessing all these resources and allows access to limited resources.

The security model used for an applet solely depends on the application. When an applet runs inside a $HotJava^{TM}$ browser, the browser is the application that has determined the security policy for that applet. The anatomy of a typical Java application is as follows:

The bytecode verifier The main task of bytecode verifier is to ensure that Java class files follow the rules of the language. In resource terms, it helps enforce memory protections for Java programs.

The class loader The main task of class loader is to loads the classes that are not found on the CLASSPATH.

The access controller The access controller is responsible for allowing (or preventing) accesses from the core API to the operating system.

The security manager The security manager acts as the primary interface between core API and the operating system. It holds the responsibility for allowing(or preventing) access to all system resources. The security manager uses the access controller for this purpose.

The security package This is the basis for authenticating signed Java classes. The security package classes are present in the java.security package.

The key database To verify the digital signature that accompanies a signed class file, the security manager and access controller uses set of keys which are present in key database. In the Java architecture, it is part of the security package, though it may be manifested as an external file or database.

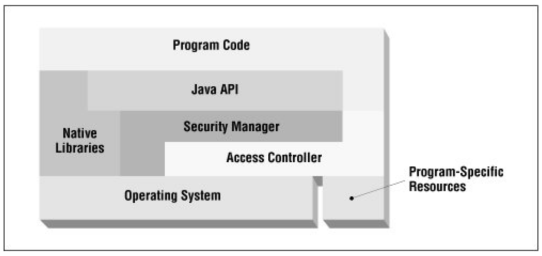

Figure 1.1: Java Security Architecture.[1]

Java Class Loader

The class loader, the security manager and the access controller works together to provide most of the protections associated with the Java sandbox. The class loader is important because there are certain information about the classes which have been loaded which only the class loader knows. These information include the origin location of a class, whether the class was signed or not etc.

As we know that Java applet itself cannot read a file when applet is being run in a browser such HotJava. The HotJava itself can read files even when it is running applets. Both browset and applet are using the same class to read the file. This differentiation is caused by class loader. The class loader allows security manager to find out particular information about the class, which allows the security manager to apply correct security policy to class depending on the context of the request. The specific machanics by which this is achieved has been discussed later.

Security Manager

The security manager is responsible for determining most of the parameters of the sandbox. It is ultimately upto the security manager to determine whether a particular operation should be permitted or rejected. e.g. if Java program tries to open a file it is upto the security manager to decide whether or not that operation should be permitted or not. Whenever an operation is requested which is not permitted, the security manager throws an exception to the Java API which asked whether operation was allowed or not. In general Java applications do not have security manager–unless the author has provided one. While Java applets have(by default) a very strict security manager.

[1]This figure has been taken from Java Security by O'reily. See references.

Access Controller

The access controller is the mechanism which the security manager actually uses to enforce its protections. The access controller is mainly build on four concepts:

Code sources A code source is a simple object which is merely the URL from which a class was loaded and the keys that were used to sign that class. The SecureClass-Loader class (and its subclasses) are responsible for creating and manipulating these code source objects.

Permissions The access controller operates on permission object which is an instance of the Permission class(java.security.Permission). The permission class is an abstract class which represents a particular operation. An instance of the Permission class represents one specific permission. A set of permissions–e.g., all the permissions that are given to classes signed by a particular individual–is represented by an instance of the Permissions class (java.security.Permissions).

Policy The security policy specifies which permissions should be applied to which code cources. It is encapsulated by the Policy class(java.security.Policy).

Protection Domains A protection domain is a grouping of a code source and permissions–that is, a protection domain represents all the permissions that are granted to a particular code source. In the default implementation of the Policy class, a protection domain is one grant entry in the file.

There are two ways in which a permission for a particular resource can be granted by access controller. When a class request access to a particular resource then the access controller checks whether the class has the required permissions or not. The it checks the class above it which used the current class object has the permission or not. Similarly it checks all the classes in the hierarchy and takes intersection of the permissions to determine the permission for current operation.

Another way is by using privileged action. A class can be temporarily given the ability to perform a specific function. This can be done by using the `doPrivileged()` function. This method specifies a stopping point in the access controller checking of permissions for classes on the stack. A protection domain can grant privileges to code that has called it, but it cannot grant privileges to code that it calls i.e. when a non-privileged class calls the Java API, the API can grant access the class the privileges to do some action. However any class cannot grant permission to a class it calls.

This kind of behaviour by `doPrivileged()` lead to many potential security threats to the system. Methods could read restricted files when invoked with doPrivilege and then can leak the sensitive data.

The structure of this report is as follows: the section Problem Statement formally describes the problem we intend to solve; the section Points-to Analysis describes the state of the art in abstractions suitable for OOP languages. It also describes the state of the art shape analysis; the section Security Analysis describes the analysis that intend to perform. It also describes the difference between our analysis and state of the art; the section Datalog gives a general structure and execution model of datalog. It also describes a specific interpreter of datalog, bddbddb which we have used in our analysis; we conclude in the next section with suggestions for future works.

Problem Statement

Oracle has issued secure coding guidelines but they may not always be followed while developing libraries. Library function can get external class object and can execute their method using the doPrivilege. This call to doPrivilege can potentially return a classified value. Therefore it must be made sure that such classified are not returned from the library function. Let us look at an example program where a classfied value is being returned cleverly.

```java
1  public class TaintEscape {
2
3    private Object answer;
4
5    public void entry(final Object tainted) {
6    Object taint = new Object();
7    taint = AccessController.doPrivileged(
8                new PrivilegedAction<Object>() {
9                        public Object run() {
10                           return ((Either)tainted).
11                                   doSomething();
12                       }
13                  });
14
15   answer.next = taint;
16   }
17
18   public Object getValue() {
19     return answer;
20   }
21
22 }
```

In the example above a tainted value is introduced in the function and is assigned to taint at line 7. The address to taint is being stored in the publicly accessible member object answer at line 15. After the function is executed, malicious user user can get the classified value by calling the `getValue()` function. Thus this class can be compromised.

All such instances have to be detected by doing static analysis of the library code only without making any assumptions about user code. In Java all data is always stored in the heap and only a pointer to it stored on the stack. So if we had an analysis which could detect which data members can have paths(direct or indirect) to tainted value, then we can check whether any such data member will be accessible to external code. This will enable us to security vulnerabilities in the code.

Chapter 2

Points-to Analysis [2–4]

One of the techniques for detecting security violations is static analysis. Points-to analysis is a type of static analysis technique which serves as a basis for many other techniques for object-oriented programming languages like Java. It computes for all data, pointers and fields in object which can point to it while executing the program. For a static points-to anlysis to terminate, the data (or their abstraction) that a variable can point to, which are objects in Java, need to be finite. Thus we need to use abstractions to approximate concrete runtime objects(also called heap abstractions) which are finite. The precision and scalability of the points-to analysis depends on the chosen abstractions.

Let is look at an example.

```
1  head = NULL; count = 0;
2  while (count < limit){
3    x = new node();
4       count = count + 1;
5       x.next = head;
6       head = x;
7  }
```

The statement at line number 3 will be executed multiple times each time creating new node. In the concrete behaviour there will be multiple objects being created and *x* and *head* will point to last object created. While abstract behaviour keeps track of only the object creation site. The variable *x* always points to location at line number 3 while head may point to either line 3 or null.

The most basic abstraction that is used is to approximate all objects which are created by single new statement by a single object. This is referred to as object creation site abstraction. So the location of the statement represents all objects created by a statement at runtime. Another aspect of the abstraction is that two fields of an object can point to different object. We need to represent abstract objects pointed to by fields of abstract object.

A useful way to understand this is to consider the objects created on the heap and pointers to them are nodes of a graph. If an object/pointer points to another object the corresponding nodes will have a directed edge.

Using this abstraction various analysis are possible. Escape analysis is traditionally used by compilers to determine which objects are local to a method and hence can be allocated on the stack. Taint analysis in a security context determines whether the values from untrusted sources reach a secure location. This can be viewed as source to sink dataflow.

Notations: Let $\mathcal{H}$ denote the set of all the heap directed pointers at any given time and $\mathcal{F}$ denote the set of all pointers at that time. For $p,q \in \mathcal{H}$, a path from p to q is sequence of pointer fields which needs to be traversed for reaching q from p. Since there is no upper limit on length of path we only keep track of first field of the path. TO distinguish between path of length 1(direct path) and path of length more than 1(indirect path) we use superscript D and I respectively. Since it is possible to have multiple indirect paths starting at a field f, we use no. of paths as superscript along with I. If this no. of paths is more than a certain k then we use inf to represent it.

We also use field sensitive path matrix P_F which stores all the information about paths between objects(direct and indirect paths).

Given $p,q \in \mathcal{H}, f \in \mathcal{F}$:

ε	$\in P_F[p,p]$	where ε denotes the empty path.
f^D	$\in P_F[p,q]$	if there is a direct path f from p to q.
f^{Im}	$\in P_F[p,q]$	if there are m indirect paths starting with field f from p to q and $m \leq k$.
$f^{I\infty}$	$\in P_F[p,q]$	if there are m indirect paths starting with field f from p to q and $m > k$.

Two pointers $p,q \in \mathcal{H}$ are said to interfere at a program point if there exists $s \in \mathcal{H}$ such that both p and q have paths reaching s at that point.

We also use following operations in our analysis. Let S denote the set of approximate paths between two nodes, P denote a set of pair of paths, and $k \in \mathcal{N}$ denote the limit on maximum indirect paths stored for a given field. Then,

- Projection: For$f \in \mathcal{F}, S \triangleright f$ extracts the paths starting at field f.

$$S \triangleright f \equiv S \cap \{f^D, f^{I1}, ..., f^{Ik}, f^{I\infty}\}$$

- Counting: The count on the number of paths is defined as:

$$|\varepsilon| = 1, |f^D| = 1, |f^{I\infty}| = \text{inf}$$
$$|f^{Ij}| = j \quad for\, j \in \mathcal{N}$$
$$|S| = \sum_{\alpha \in S} |\alpha|$$

2.1 Shape Analysis [3,4]

One of the most popular analysis based on this abstraction is the Shape Analysis. This has traditionally been used for speeding up compilers and for dataflow analysis. Programs in all modern languages use heap intensively. If we want to have any non-trivial analysis, then we need to reason about the heap. The reasoning is complex because heap structures are not static but are manipulated dynamically during execution of the program.

The goal of shape analysis is to estimate the shape of data structure accessible from a given heap directed pointer: is it tree-like, dag-like or a general graph containing cycles. More specifically it identifies tree-like and dag-like structures built by program and provide conservative estimates otherwise.

State of the art shape analysis [3] captures field sensitivity using two components: (a) Field based boolean values to remember direct connections between two pointer variables, and (b) Path matrices that store the approximate paths between pointer variables. The shape of a pointer variable at any given program point can be inferred from these two components.

Let us look at the following code snippets.

```
object o1 = new object(), o2 = new object(), o3 = new object();
o1.next1=o2;
o2.next1=o3;
o3.next1=o1;
```

```
object o1 = new object(), o2 = new object(), o3 = new object();
o1.next1=o2;
o2.next1=o3;
o1.next2=o3;
```

If o2 is a tainted object then in the first code snippet nothing can be returned because all the variables have paths to o2, while o3 can be returned in second code snippet.

For $\{p,q\} \subseteq \mathcal{H}, f \in \mathcal{F}, n \in \mathcal{N}$ and op $\in$ {+,-}, we have basic eight statements that can access or modify the heap structures. The superscript *in* denotes those values before the analysis of current statement. The superscript *kill* denotes those values which will be removed from *in* values to get final values. The superscript *gen* denotes those values which will be generated from this statement.

We give analysis of each of the basic statements and the changes introduced by them.

p = NULL This statement kills all the existing values of p. The heap node pointed by p is no longer reachable by p.

$$\begin{aligned} p^{\text{kill}}_{\texttt{Cycle}} &= p^{\text{in}}_{\texttt{Cycle}} & p^{\text{kill}}_{\texttt{Dag}} &= p^{\text{in}}_{\texttt{Dag}} \\ p^{\text{gen}}_{\texttt{Cycle}} &= \textbf{False} & p^{\text{gen}}_{\texttt{Dag}} &= \textbf{False} \end{aligned}$$

$\forall s \in \mathcal{H},$

$$\begin{aligned} f_{ps} &= \textbf{False} & f_{sp} &= \textbf{False} \\ P_F^{\text{kill}}[p,s] &= P_F^{\text{in}}[p,s] & P_F^{\text{gen}}[p,s] &= \emptyset \\ P_F^{\text{kill}}[s,p] &= P_F^{\text{in}}[s,p] & P_F^{\text{gen}}[s,p] &= \emptyset \end{aligned}$$

p = malloc() After this statement all the existing dataflow values of p get killed and p starts to a newly allocated object. The kill effect is exactly same as $p = NULL$. After the statement p has an empty path to itself only.

$$\begin{aligned} p^{\text{kill}}_{\texttt{Cycle}} &= p^{\text{in}}_{\texttt{Cycle}} & p^{\text{kill}}_{\texttt{Dag}} &= p^{\text{in}}_{\texttt{Dag}} \\ p^{\text{gen}}_{\texttt{Cycle}} &= \textbf{False} & p^{\text{gen}}_{\texttt{Dag}} &= \textbf{False} \end{aligned}$$

$\forall s \in \mathcal{H},\ s \neq p,$

$$\begin{aligned} f_{ps} &= \textbf{False} & f_{sp} &= \textbf{False} \\ P_F^{\text{kill}}[p,s] &= P_F^{\text{in}}[p,s] & P_F^{\text{gen}}[p,s] &= \emptyset \\ P_F^{\text{kill}}[s,p] &= P_F^{\text{in}}[s,p] & P_F^{\text{gen}}[s,p] &= \emptyset \\ P_F^{\text{kill}}[p,p] &= P_F^{\text{in}}[p,p] & P_F^{\text{gen}}[p,p] &= \{\varepsilon\} \end{aligned}$$

p = q, p = &(q → f), p = q op n We consider these three types if statements as equivalent. These statements kill all the existing relations of p and it will point to heap object which was earlier pointed by q or NULL. The kill effect is same as previous cases. The generated paths from p will be same as those of q.

$$
\begin{array}{lll}
 & p^{\text{kill}}_{\texttt{Cycle}} = p^{\text{in}}_{\texttt{Cycle}} & p^{\text{kill}}_{\texttt{Dag}} = p^{\text{in}}_{\texttt{Dag}} \\
 & p^{\text{gen}}_{\texttt{Cycle}} = q^{\text{in}}_{\texttt{Cycle}}[q/p] & p^{\text{gen}}_{\texttt{Dag}} = q^{\text{in}}_{\texttt{Dag}}[q/p] \\
\forall s \in \mathcal{H}, s \neq p, \forall f \in \mathcal{F} & f_{ps} = f_{qs} & f_{sp} = f_{sq} \\
 & P^{\text{kill}}_F[p,s] = P^{\text{in}}_F[p,s] & P^{\text{gen}}_F[p,s] = P^{\text{in}}_F[q,s] \\
 & P^{\text{kill}}_F[s,p] = P^{\text{in}}_F[s,p] & P^{\text{gen}}_F[s,p] = P^{\text{in}}_F[s,q] \\
 & P^{\text{kill}}_F[p,p] = P^{\text{in}}_F[p,p] & P^{\text{gen}}_F[p,p] = P^{\text{in}}_F[q,q]
\end{array}
$$

p → f = NULL This statement breaks the existing link emanting from p. Thus all paths starting from p using f are broken.

$$
\begin{array}{llll}
 & p^{\text{kill}}_{\texttt{Cycle}} = \textbf{False}, & & p^{\text{kill}}_{\texttt{Dag}} = \textbf{False} \\
 & p^{\text{gen}}_{\texttt{Cycle}} = \textbf{False}, & & p^{\text{gen}}_{\texttt{Dag}} = \textbf{False} \\
\forall q, s \in \mathcal{H}, s \neq p, & f_{pq} = & \textbf{False} & \\
 & P^{\text{kill}}_F[p,q] = & P^{\text{in}}_F[p,q] \triangleright f & P^{\text{kill}}_F[s,q] = \emptyset
\end{array}
$$

p → f = q This statement first breaks all the existing links of p which starts with f and then links the object pointed by p to object pointed by q. It is possible that the shape of p can become a dag or a cycle after this statement.

$$
\begin{array}{lcl}
p^{\text{gen}}_{\texttt{Cycle}} & = & (f_{pq} \wedge q^{\text{in}}_{\texttt{Cycle}}) \vee (f_{pq} \wedge (|P_F[q,p]| \geq 1)) \\
p^{\text{gen}}_{\texttt{Dag}} & = & (f_{pq} \wedge q^{\text{in}}_{\texttt{Dag}}) \vee (f_{pq} \wedge (|I_F[p,q]| > 1) \\
q^{\text{gen}}_{\texttt{Cycle}} & = & f_{pq} \wedge (|P_F[q,p]| \geq 1) \\
q^{\text{gen}}_{\texttt{Dag}} & = & \textbf{False} \\
f_{pq} & = & \textbf{True}
\end{array}
$$

For nodes $s \in \mathcal{H}$ other than p or q, the function s^{gen}_{cycle} captures the fact that cycle on s consists of some path from s to p(or q) and the fact that cycles on p(or q) have just been created by this statement. Similarly the function s^{gen}_{dag} simply say that variable s reaches a DAG because there are more than one way of interference between s and q.

$$
\begin{array}{lcl}
s^{\text{gen}}_{\texttt{Cycle}} & = & ((|P_F[s,p]| \geq 1) \wedge f_{pq} \wedge q^{\text{in}}_{\texttt{Cycle}}) \\
 & & \vee\ ((|P_F[s,p]| \geq 1) \wedge f_{pq} \wedge (|P_F[q,p]| \geq 1)) \\
 & & \vee\ ((|P_F[s,q]| \geq 1) \wedge f_{pq} \wedge (|P_F[q,p]| \geq 1)), \\
 & & \qquad \forall s \in \mathcal{H}, s \neq p, s \neq q \\
s^{\text{gen}}_{\texttt{Dag}} & = & (|P_F[s,p]| \geq 1) \wedge f_{pq} \wedge (|I_F[s,q]| > 1), \\
 & & \qquad \forall s \in \mathcal{H}, s \neq p, s \neq q
\end{array}
$$

After this statement all the nodes which have paths towards p will have paths towards q.

For $r,s \in \mathcal{H}$:

$$\begin{aligned}
P_F^{\text{gen}}[r,s] &= |P_F^{\text{in}}[q,s]| \star P_F^{\text{in}}[r,p],\ s \neq p,\ r \notin \{p,q\} \\
P_F^{\text{gen}}[r,p] &= |P_F^{\text{in}}[q,p]| \star P_F^{\text{in}}[r,p],\ r \neq p \\
P_F^{\text{gen}}[p,r] &= |P_F^{\text{in}}[q,r]| \star (P_F^{\text{in}}[p,p] \ominus \{\varepsilon\} \cup \{f^{I1}\}), \\
&\qquad\qquad r \neq q \\
P_F^{\text{gen}}[p,q] &= \{f^D\} \cup (|P_F^{\text{in}}[q,q] - \{\varepsilon\}| \star \{f^{I1}\}) \cup \\
&\quad (|P_F^{\text{in}}[q,q]| \star (P_F^{\text{in}}[p,p] \ominus \{P_F^{\text{in}}[p,p] \triangleright f \cup \{\varepsilon\}\})) \\
P_F^{\text{gen}}[q,q] &= 1 \star P_F^{\text{in}}[q,p] \\
P_F^{\text{gen}}[q,r] &= |P_F^{\text{in}}[q,r]| \star P_F^{\text{in}}[q,p],\ r \notin \{p,q\}
\end{aligned}$$

p = q $\rightarrow$ **f** The values killed by this statement are same as those of $p = NULL$. The values created are heavily approximated. After this statement p points to heap object which is accessible from pointer q through link f. This only shows that p is reachable from any r which can reach $q \rightarrow f$ before assignment. Shape of no other pointer variable gets affected.

We record that q reaches p through f. Also, any object reachable from q using f is marked as reachable from p using all fields.

$$\begin{aligned}
f_{qp} &= \textbf{True} \\
h_{pr} &= |P_F^{\text{in}}[q,r] \triangleright f| \geq 1 \quad \forall h \in \mathcal{F}, \forall r \in \mathcal{H}
\end{aligned}$$

As a side effect of this statement, any node s that is reachable from q through f becomes reachable from p. This, however, cannot determine path from p to s. So a conservative estimate is that any path starting starting from can reach s.

$$\forall s \in \mathcal{H}, s \neq p \wedge P_F^{\text{in}}[q,s] \triangleright f \neq \emptyset$$

$$P_1[p,s] = \begin{cases} \{\varepsilon\} & (P_F^{\text{in}}[q,s] \triangleright f = f^D) \\ & \wedge\ q.\text{shape evaluates to } \texttt{Tree} \text{ or } \texttt{Dag} \\ \mathcal{U} & \text{Otherwise} \end{cases}$$

$$P_1[p,p] = \begin{cases} \mathcal{U} & q.\text{shape evaluates to } \texttt{Cycle} \\ \{\varepsilon\} & \text{Otherwise} \end{cases}$$

In case a cycle is reachable from q we can safely say that there is self loop on p.

$$P_2[s,p] = \infty \star P_F^{\text{in}}[s,q] \quad \forall s \in \mathcal{H},\ s \notin \{p,q\}$$

If $p \neq q$, then we record the path from q to p as:

$$P_2[q,p] = \begin{cases} \{f^D\} & q.\text{shape evaluates to } \texttt{Tree} \\ \mathcal{U} & \text{Otherwise} \end{cases}$$

Any node s(excluding p and q), that has path to q before statement, will have path to p after statement. However we cannot exact number of such paths and therefore we use upperlimit ∞ as a approximation.

$$\forall r,s \in \mathcal{H},\ r \notin \{p,q\}$$

$$P_3[s,p] \quad = \quad \bigcup_r \{\alpha \mid f^D \in P_F^{\text{in}}[q,r] \wedge \alpha \in P_F^{\text{in}}[s,r] \ominus P_F^{\text{in}}[s,q]\}$$

Those nodes that interfere with the node corresponding to $q \to f$, without going through q will have paths to p after the statement.

$$P_F^{\text{gen}}[r,s] \quad = \quad P_1[r,s] \cup P_2[r,s] \cup P_3[r,s] \quad \forall r,s \in \mathcal{H}$$

Let us look at small example[1] program and modifications to its path matrix after the statements.

```
       void mirror(tree t) {
S11:     tl = t->left;
S12:     tr = t->right;
S13:     mirror(tl);
S14:     mirror(tr);
S15:     t->left = tr;
S16:     t->right = tl;
       }
```

Stmt	Path matrix P_F after the stmt			
		t	tl	tr
S11	t	$\emptyset$	$\{\texttt{left}^{\texttt{D}}\}$	$\emptyset$
	tl	$\emptyset$	$\emptyset$	$\emptyset$
	tr	$\emptyset$	$\emptyset$	$\emptyset$
		t	tl	tr
S12	t	$\emptyset$	$\{\texttt{left}^{\texttt{D}}\}$	$\{\texttt{right}^{\texttt{D}}\}$
	tl	$\emptyset$	$\emptyset$	$\emptyset$
	tr	$\emptyset$	$\emptyset$	$\emptyset$
		t	tl	tr
S15	t	$\emptyset$	$\emptyset$	$\{\texttt{right}^{\texttt{D}},\texttt{left}^{\texttt{D}}\}$
	tl	$\emptyset$	$\emptyset$	$\emptyset$
	tr	$\emptyset$	$\emptyset$	$\emptyset$
		t	tl	tr
S16	t	$\emptyset$	$\{\texttt{right}^{\texttt{D}}\}$	$\{\texttt{left}^{\texttt{D}}\}$
	tl	$\emptyset$	$\emptyset$	$\emptyset$
	tr	$\emptyset$	$\emptyset$	$\emptyset$

[1]This example has been taken from Prof. Amey's paper on Shape Analysis. See references

Chapter 3

Security Analysis

We propose a security based on points-to framework which detect unauthorized escape of classified values from the program. Our analysis determines which pointer are safe to be returned from a library function and which pointers can be stored as public data members in the class object.

Tainted values can be introduced inside the function using statements like `value = doPrivileged(...);` Other data members can then point to these values directly or indirectly via multiple paths. All such object which have path to these tainted values have to identified so that they do not escape the function. Once we have determined which objects are safe to be returned, we can identify the functions which are returning such values and thus might be used to perform malicious activities.

Notations: The notation used is similar to that of shape analysis. We have an extra matrix for determining paths to bad object.

We also use field sensitive taint matrix T_F which stores all the information about paths to tainted objects(direct and indirect paths). The rows are pointer names and columns are field names. Also we have a special column called D for pointer which directly points to tainted object.

Given $p \in \mathcal{H}, f \in \mathcal{F}$:

1	$\in T_F[p,D]$	when p directly points to tainted object
f^D	$\in T_F[p,f]$	if there is a direct path f from p to tainted object.
f^{Im}	$\in T_F[p,f]$	if there are m indirect paths starting with field f from p to taint and $m \leq k$.
$f^{I\infty}$	$\in T_F[p,f]$	if there are m indirect paths starting with field f from p to taint and $m > k$.

We store whether an object is bad or not using boolean values. We analyze each kind of basic statement that can modify heap or introduce tainted value in the function. Since most of effects of these statements are similar to those of shape analysis, we only give differences here. Also we only emphasize those which might change after a particular statement.

p = "doPrivilege()" This rule specifies the introduction of tainted value into the system. First of all this rule kills are existing relations of p. The nodes which X

could earlier reach are no longer reachable. Also those nodes which could earlier reach p can no longer reach. If X had any path to something earlier then that gets killed. Now p points to a tainted value directly.
$\forall f \in \mathcal{F}$

$$T_F\text{kill}[p,f] = T_F\text{in}[p,f] \qquad T_F^{\text{gen}}[p,D] = 1$$

p = NULL All the existing relations of p are killed. All paths from p to q or from q to p gets killed. Furthermore if p had a path to tainted value(direct or indirect via any field), that also gets killed. p is now completely $Null$. $\forall f \in \mathcal{F}$

$$T_F^{\text{kill}}[p,D] = T_F^{\text{in}}[p,D] \qquad T_F^{\text{gen}}[p,D] = \emptyset$$
$$T_F^{\text{kill}}[p,f] = T_F^{\text{in}}[s,p] \qquad T_F^{\text{gen}}[p,f] = \emptyset$$

p = malloc() The relations killed by this rule are exactly same as that of the previous rule. The only change is that now p is newly allocated object rather than $Null$. There is no change in tainted paths of other objects.
$\forall f \in \mathcal{F}$

$$T_F^{\text{kill}}[p,D] = T_F^{\text{in}}[p,D] \qquad T_F^{\text{gen}}[p,D] = \emptyset$$
$$T_F^{\text{kill}}[p,f] = T_F^{\text{in}}[p,f] \qquad T_F^{\text{gen}}[p,f] = \emptyset$$

p = q, p = &(q $\to$ f), p = q op n We consider all these three statements as equivalent. All the existing relations of p are killed and now p will point to same object which was pointed to by q or $Null$. All the previous tainted paths also gets killed and the new tainted relations of p will be same as those of q. Note we have made this approximation conservatively. The generated paths from p will be same as those of q
$\forall f \in \mathcal{F}$

$$T_F^{\text{kill}}[p,D] = T_F^{\text{in}}[p,D] \qquad T_F^{\text{gen}}[p,D] = T_F^{\text{in}}[q,D]$$
$$T_F^{\text{kill}}[p,f] = T_F^{\text{in}}[q,f] \qquad T_F^{\text{gen}}[p,f] = T_F^{\text{in}}[q,f]$$

p $\to$ f = NULL This statement kills all existing relations of p which starts with f. Also all paths to tainted objects from p via field f gets killed as well. $\forall s \in \mathcal{H}, s \neq p, g \in \mathcal{F}$

$$T_F^{\text{kill}}[p,f] \quad = \quad T_F^{\text{in}}[p,f] \qquad T_F^{\text{kill}}[s,D] = \emptyset$$
$$T_F^{\text{kill}}[s,g] = \emptyset$$

p $\to$ f = q This rule can viewed as a combination of two rules. First one nullify the field f of p ans second builds paths from p via f to all s to which q has path. So we first break all the relations of $p \to f$ as per above. Then create paths to all s to which q had a path. Also any r which had path to p will have path to q and all s reachable from q.

Initially all paths to tainted objects starting from p via f gets killed. The if q had a path to tainted object, then p will also have a path to tainted object via f. Also all r having paths to p will have paths to tainted objects if q has such a path.

For $r, s \in \mathcal{H}, g \in \mathcal{F}$:

$$T_F^{\text{gen}}[p,f] = T_F^{\text{in}}[q,D] \vee T_F\text{in}[q,g]$$
$$T_F^{\text{gen}}[r,g] = P_F^{\text{in}}[r,p] \triangleright g! = \emptyset \wedge$$
$$(T_F^{\text{in}}[q,D] \vee T_F\text{in}[q,g])$$

p = q $\rightarrow$ f All the existing paths(including tainted paths) of p gets killed. The paths generated are same as those of shape analysis with augmentation of the few tainted paths. If q had a direct path to tainted object via f then p now directly points to tainted object. Else if q had an indirect path to tainted object then p will have indirect path to tainted object via all its fields.

$\forall g \in \mathcal{F}$

$$T_F^{\text{gen}}[p,g] = T_F^{\text{in}}[q,f]$$

Let us look at small example program and modifications to its taint matrix after the statements.

```
      void mirror(tree t) {
S11:    tn = doPrivileged();
S12:    t->left = tn;
S13:    tn = Null;
S14:    t->right = tr;
      }
```

Stmt	Taint matrix T_F after the stmt			
		D	left	right
S11	tn	1	$\emptyset$	$\emptyset$
	tr	$\emptyset$	$\emptyset$	$\emptyset$
	t	$\emptyset$	$\emptyset$	$\emptyset$
		D	left	right
S12	tn	1	$\emptyset$	$\emptyset$
	tr	$\emptyset$	$\emptyset$	$\emptyset$
	t	$\emptyset$	$\{\text{left}^{\text{D}}\}$	$\emptyset$
		D	left	right
S13	tn	$\emptyset$	$\emptyset$	$\emptyset$
	tr	$\emptyset$	$\emptyset$	$\emptyset$
	t	$\emptyset$	$\{\text{left}^{\text{D}}\}$	$\emptyset$
		D	left	right
S14	tn	$\emptyset$	$\emptyset$	$\emptyset$
	tr	$\emptyset$	$\emptyset$	$\emptyset$
	t	$\emptyset$	$\{\text{left}^{\text{D}}\}$	$\emptyset$

Chapter 4

Datalog [5,6]

Datalog is a declarative logic programming language that syntactically is a subset of prolog. It is often used as a query language for deductive databases. Some of the recent applications of datalog include data integration, information extraction, networking, program analysis, security and cloud computing.

Unlike prolog, the order of statements does not affects the program in datalog. Furthermore, Datalog queries on finite sets are guarranteed to terminate, so Datalog does not have Prolog's cut operator. This makes Datalog a truely declarative language.

The main features of difference between datalog and prolog are

- Datalog disallows complex terms as arguments of of predicates e.g., p(1,2) is admissible but not p(f(1),2),
- It imposes restrictions on the use of negation and recursion
- It requires that every variable that appears in the head of a clause also appears in a nonarithmetic positive literal in the body of the clause,
- It requires that every variable appearing in a negative literal in the body of a clause also appears in some positive literal in the body of the clause

Query evaluation with Datalog is based on first order logic, and thus is sound and complete.

4.1 Advantages of Datalog

Using Datalog over traditional programming languages has many advantages like

- Analysis implementation is greatly simplified. It can be expressed in few lines of Datalog as compared to thousands of lines of traditional languages.
- By automatically deriving implementation from Datalog specification, we introduce fewer errors.
- Since all analysis informational is expressed in a uniform manner, it is easy to use analysis results or to combine analyses.
- Optimization of Datalog can be applied to all analyses expressed in the language.

Although Datalog has many advantages, one of its biggest disadvantage comes from the fact that implementation using logic programming systems are often slower than traditional implementations and can have difficulty scaling to large programs.

We give a very brief introduction to datalog syntax. A datalog program consists of facts and rules. Facts are stored in tables. If john is the parent of douglas then `parent(john,douglas).` stores this fact in table named parent. Each item in the parenthesized list following the name of the table is called a term.

A query can be used to see if a particular fact is the table. e.g. `parent(john,douglas)?` It can also be used to find out value of specific attribute from certain row. e.g. `parent(john,X)?` returns all childs of john. A term that begin with a capital letter is a variable.

A deductive database can use rules of inference to derive new facts. e.g `ancestor(A,B):-parent(A,B).`

The main difference from prolog is that the order in which clauses are asserted is irrelevant. All queries terminate and every possible answer is derived.

Let us look at a sample datalog program. This is interpretable with bddbddb.

```
Z 1024

assign(X:Z,Y:Z) outputtuples
bad(X:Z) outputtuples
return(X:Z) outputtuples
shouldnotreturn(X:Z) outputtuples

bad(3).

assign(0,1).
assign(1,2).
assign(10,5).
assign(5,4).
assign(4,2).
assign(7,3).
assign(4,7).
assign(X,Y):- assign(X,Z),assign(Z,Y).

shouldnotreturn(X):- bad(X).

shouldnotreturn(X)?
```

We first define domain of `Z` as integers. Then we give declarations of rules. This program is based only on assignment. Then it contains all the program facts. After it contains the definitions of the rules and then a query asking which object should not be returned i.e which object are bad.

Now we discuss below one of the interpreters of Datalog which tried to scale analysis for larger code bases.

4.2 bddbddb [7]

Bddbddb, which stands for BDD-Based Deductive DataBase, is a solver for Datalog with stratified negation, totally-ordered finite domains and comparison operators. bddbddb represents relations using binary decision diagrams or BDDs. bddbddb translates each Datalog rule into a series of BDD operations, the find fix-point by applying the operations for each rule until the program converges on a final set of relations.

4.3 Expressing a Program Analysis in Datalog

Program analyses like type inference and points-to analyses, are often described formally in the compiler architecture as inference rules, which naturally map to Datalog programs. A datalog program analysis works similar to any other program. It accepts set of relations as input and generated new relations as output.

4.4 From Datalog to BDD Operations

The solution to a datalog query with finite domains and stratified negations can be obtained by applying sequences of relational algebra operations corresponding to the Datalog rules iteratively, until a fixedpoint solution is reached. This is exactly similar query evaluation for relational databases.

Now the relations can be encoded using boolean functions over tuples of binary values. The first step is assigning elements in the domain, consecutive numeric values, starting from 0. Suppose each of the attributes of an n-ary relation is associated with numeric domains $D_1 \times ... \times D_n \rightarrow \{0,1\}$ such that $(d_1,...,d_n) \in R$ iff $f(d1,...,d_n) = 1$ and vice-versa.

Large boolean functions can be represented efficiently using BDDs, which were originally invented for hardware verification to efficiently store a large number of states that share many commonalities.

The boolean functions needed to evaluate queries are a standard feature of BDD libraries. The $\wedge$(and), $\vee$(or), and $-$(difference) boolean function operations can be applied to two BDDs, producing a BDD of the resulting function. Other operations like existential quantification etc. can also be easily used to produce new BDD.

4.5 Translating and Optimizing Datalog Programs

The bddbddb system applies large number of optimizations to transfoem Datalog programs into efficient BDD operations:

- Before compilation the input is normalized using many transformations. These include substituting comparisons with precomputations, inlining temporary relations, changing some variables to underscore etc.
- Datalog rules are also optimized for fast computation. The main optimization applied include removing rules which don't contribute to final output, stratification, finding cycles, and determining rule application order.
- Rules are translated to intermediate representation based relational algebra.
- Incrementalization is also applied to prevent repeated computation. So if all inputs are same are previous application of this rule then the result can be directly obtained and need not be computed
- Additional optimzations like global value numbering, copy propogation and liveness which reduces memory footprint.
- Decision variables are also ordered in such a way that keeps the size of tree smaller and thus reduces computation time.

Chapter 5

Conclusion

Although Java itself has tried to address the problem of security, there are some loopholes in the current security architecture. The architecture consisting of multiple layers including security manager, access controller, class loader is far from complete. Some of the layers like security manager are carried on just for backward compatibility since all the work can be done by access controller.

Due to the possibility of a class being able to give privileges to other class for performing operations, certain loopholes have been introduced. These loopholes needs to be addressed at the analysis level due to huge importance of the privileged action for legitimate accesses.

Points-to analysis is a type of static analysis which can be used to obtain information about the program pointers. Its accuracy depends on type of abstraction used. It is the basis of many analysis like field-sensitive shape analysis.

We have developed a security analysis based on points-to analysis to detect malicious behaviour. We detect all the tainted objects and all pointers which have paths to these tainted objects. This allows to determine which objects can be returned.

We plan to code our analysis in datalog which is a declarative programming language. It is subset of prolog and is purely declarative. Even the order of statements does not matter. But datalog has not been standardized like C++, Java etc. So there many different interpreters each different from others in many aspects.

Datalog cannot operate directly on Java programs and requires intermediate simplified representation of the program in form of facts. This task does not have a well defined tool as of now. Many tools exists likt soot and soufle for extracting facts from program but they all have lot of problems in working. There are lots of version conflicts and most of these tools have been designed for some earlier versions of Java. Even the fact extractor for bddbddb has been designed with some of the starting versions of java which are no longer supported. So the process of fact extraction is not in good form as of now.

In future we would like to standardize the tools used in the analysis. We also would like to scale the analysis for larger programs as most of programs where this kind will be useful are libraries which have large code bases.

Bibliography

[1] S. Oaks, Java Security. O'reily Publication, 1998.

[2] Y. L. Padmanabhan Krishnan and K. R. Raghavendra, "Detecting unauthorised information flows using points-to analysis," in Engineering and Technology Reference, 2015.

[3] S. Dasgupta, A. Karkare, and V. K. Reddy, "Precise shape analysis using field sensitivity," Innov. Syst. Softw. Eng., vol. 9, pp. 79–93, June 2013.

[4] R. Ghiya and L. J. Hendren, "Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c," in POPL '96, pp. 1–15, January 1996.

[5] "Datalog." `https://docs.racket-lang.org/datalog/Tutorial.html`. Accessed: 2016-11-24.

[6] "Datalog wiki." `https://en.wikipedia.org/wiki/Datalog`. Accessed: 2016-11-24.

[7] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using datalog with binary decision diagrams for program analysis," in Proceedings of the Third Asian Conference on Programming Languages and Systems, APLAS'05, (Berlin, Heidelberg), pp. 97–118, Springer-Verlag, 2005.