Who Dropped My Tables: Taint Analysis in Simple Java Programs

ZHIMING MENG, University of San Francisco, USA
JIAHE TIAN, University of San Francisco, USA

Nowadays, with modern frameworks and libraries. SQL Injections are the least of one's worries when dealing with queries. However, they serve as an intuitive example on showing how taint analysis is useful and even significant. In this paper, we write a very simple taint analysis in Soot that looks for areas of user input and traces them to query executions.

ACM Reference Format:
Zhiming Meng and Jiahe Tian. 2024. Who Dropped My Tables: Taint Analysis in Simple Java Programs. 1, 1 (May 2024), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Importance of Taint Analysis

In an era where data breaches and cyber threats are increasingly common, securing web applications has become paramount. Taint analysis serves as a critical technique in identifying vulnerabilities by tracking how information flows through software, particularly how data from untrusted sources interacts with sensitive areas of an application. For example, we might want to prove that an area in our application that stores or has access to sensitive user information never interacts with areas that have network access. Taint analysis will let us find these connections and break them, ensuring security.

1.2 Objective and Scope

The objective of this project is to implement a robust taint analysis tool using the Soot framework, focusing on Java web applications to detect and prevent SQL injections. This analysis was confined to applications built with Java 8, using simple, open-source projects for initial testing. The analysis itself is excuted on Java 17.

2 APPROACH

2.1 Intra-procedural Analysis

2.1.1 ForwardFlowAnalysis. Soot provides an abstract class `ForwardFlowAnalysis` that we subclass to implement our own taint analysis. We define a concrete implementation of the abstract method `flowThrough` which utilizes set implementing the `FlowSet` interface to compute a fixed-point in the dataflow through a worklist algorithm. The `flowThrough` method traverses the method, with every program point having its own `FlowSet`. Since we are working with Jimple, program points are of type `Stmt`, and they implement the `Unit` interface, needed to denote a unit of execution within the intermediate representation.

Authors' Contact Information: Zhiming Meng, University of San Francisco, San Francisco, California, USA; Jiahe Tian, University of San Francisco, San Francisco, California, USA.

2024. ACM XXXX-XXXX/2024/5-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.1.2 TaintStore. To be able to map variables to their respective taint sources for each program point, we create a `TaintStore` class that implements the `FlowSet` interface. By setting the generic parameter of the `FlowSet` interface to `Map.Entry<K, Set<V>>`, we are able to have an underlying $store : var \mapsto \{s \mid s \text{ is a taint source}\}$ mapping structure. A `LinkedTreeMap` is used to preserve the order that the individual statements are traversed. The key and value types are left generic for extensibility should we need to use different types to represent variables and taint sources.

Table 1. Methods for interacting with taint store.

method	params	operation
addTaint	k, v	$store[k] = store[k] \cup \{v\}$
addTaints	$k, \{v_1, v_2, ...\}$	$store[k] = store[k] \cup \{v_1, v_2, ...\}$
propagateTaints	k_1, k_2	$store[k_2] = store[k_1] \cup store[k_2]$
setTaint	k, v	$store[k] = \{v\}$
setTaints	$k, \{v_1, v_2, ...\}$	$store[k] = \{v_1, v_2, ...\}$
setTaints	k_1, k_2	$store[k_1] = store[k_2]$
clearTaints	k	$store[k] =$
isTainted	k	**return** *true* **if** $\lvert store[k] \rvert > 0$
getTaints	k	**return** $store[k]$

2.1.3 flowThrough. We override `flowThrough` to check if the *unit* parameter is an instance of certain interface and classes to determine the rules that will be used to propagate taints. In Algorithm 1, *in* is the incoming taint store from the analysis of the previous statement, *out* is the statement that will be modified in the analysis of the current function, and then passed to the analysis of the next statement. Methods used to manipulate the taint store are described in Table 1.

Algorithm 1 Intra-procedural analysis with flowThrough

```
Map sinkToSourceMap : sink ↦ {src | src taints sink}
procedure FLOWTHROUGH(in, unit, out)
    in.COPY(out)                                              ▷ Sets out = in as a baseline
    if unit instanceof JAssignmentStmt then                   ▷ Handles assignment statements
        rightOp ← jAssignmentStmt.GETRIGHTOP()
        leftOp ← jAssignmentStmt.GETLEFTOP()
        if ISSOURCE(rightOp) then
            out.SETTAINT(leftOp, unit)                        ▷ Unit is used as the key for sources
        end if
        if rightOp instanceof StaticFieldRef then             ▷ Handles static fields
            out.SETTAINT(rightOp, leftOp)
        end if
        if rightOp instanceof InvokeExpr then                 ▷ Handles method invokes
            if invokeExpr instanceof InstanceInvokeExpr then
                out.SETTAINT(leftOp, instanceInvokeExpr.GETBASE())
            end if
            if invokeExpr instanceof StaticInvokeExpr then
                out.SETTAINT(leftOp, staticInvokeExpr)
            end if
            if invokeExpr instanceof DynamicInvokeExpr then
                out.SETTAINT(leftOp, dynamicInvokeExpr)
            end if
            for arg in invokeExpr.GETARGS() do                ▷ Weak update arguments
                out.PROPAGATETAINTS(arg, leftOp)
            end for
        end if
        if rightOp instanceof Local then
            out.SETTAINTS(leftOp, rightOp)
        end if
        if rightOp instanceof BinopExpr then
            out.CLEARTAINTS(leftOp)
            out.SETTAINT(leftOp, binopExpr.GETOP1())
            out.SETTAINT(leftOp, binopExpr.GETOP2())
        end if
    end if
    UPDATESINK()                      ▷ Add currently tainted sources to solution if sink
end procedure
```

3 RESULTS

This is a taint analysis that works for a very limited instruction set. As an example of programs with possible taints, we created simple test programs that reads for a user input, then executes queries to an SQL server. In our test program, we identify `nextLine` as a source invoke, and `executeQuery`, `executeUpdate`, `execute` as sink invokes.

3.1 Test Code

3.1.1 Sources. In our test program, we provided two lines that scanned for *username* and *password* variables through a user's input.

Listing 1. Taint sources in Java.

```
System.out.println("Enter username:");
String username = scanner.nextLine();
System.out.println("Enter password:");
String password = scanner.nextLine();
```

In the Jimple generated from this test program, two lines were identified as lines that introduce taint sources, which correspond to the two scanner lines in the Java.

Listing 2. Taint sources in test Jimple.

```
$stack9 = virtualinvoke $stack8.<java.util.Scanner: java.lang.String nextLine()>()
$stack12 = virtualinvoke $stack11.<java.util.Scanner: java.lang.String nextLine()>()
```

3.1.2 Sinks. The below line is set as the sink statement for the program.

Listing 3. Sinks in Java.

```
ResultSet rs = stmt.executeQuery(sql);
```

Following is the sink statement converted to Jimple.

Listing 4. Sinks in Jimple.

```
$stack16 = interfaceinvoke $stack6.<java.sql.Statement: java.sql.ResultSet
     executeQuery(java.lang.String)>($stack13)
```

3.1.3 Output. After running the analysis on the test program, the below was output to terminal. The arrows on the left show a source to sink relationship. The line above shows where the taint source is introduced. The line below shows where the taint is sunk.

Listing 5. Source to sink paths.

```
    $stack9 = virtualinvoke $stack8.<java.util.Scanner: java.lang.String nextLine()>()
--> $stack16 = interfaceinvoke $stack6.<java.sql.Statement: java.sql.ResultSet
     executeQuery(java.lang.String)>($stack13)

    $stack12 = virtualinvoke $stack11.<java.util.Scanner: java.lang.String
         nextLine()>()
--> $stack16 = interfaceinvoke $stack6.<java.sql.Statement: java.sql.ResultSet
     executeQuery(java.lang.String)>($stack13)
```

When we create an additional line in the test program to set the value of *sql* to a `String` literal before being sunk, the analysis returned nothing for the taint information.

4 DISCUSSION

Through the initial results, it can be seen that the analysis is able to detect a very simple path through assignments in our test program. Furthermore, when the tainted variables are overwritten before being sunk, the analysis is able to determine that there is no active source to sink path, and thus print no taint information. This showcases the taint store being functional at a very basic level.

4.1 Benchmarks

4.1.1 Detection Accuracy. The tool was benchmarked for its ability to accurately detect SQL Injection vulnerabilities. It successfully identified the taint propagation from the source `Scanner.nextLine()` to the sink `Statement.executeQuery()`.

4.1.2 Limitations. The analysis currently coarsely makes big assumptions for assignment statements. It is also currently only intra-procedural. An attempt was made to build an abtract context for an inter-procedural analysis, but the complexity of Soot's call graph proved it a challenge for this time frame.

5 FUTURE WORK

We chose to use Soot for this project as there are many existing codebases and documentation. This taint analysis could be ported to SootUp [1], the successor to Soot. As SootUp was designed to be Soot but better, future implementations of inter-procedural analysis would possibly be easier to implement in SootUp. Also, the analysis can be expanded further to fit the full range of Jimple's syntax.

REFERENCES

[1] [n. d.]. https://soot-oss.github.io/SootUp/speedup_typehierarchy_preview/
[2] Cloudanger. [n. d.]. Inserting code in this LaTeX document with indentation. https://stackoverflow.com/a/3175141