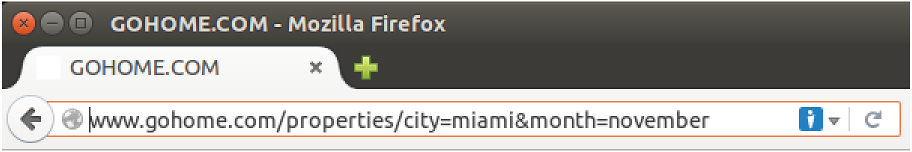
Boolean Formulas for the Static Identification of Injection Attacks in Java

Michael D. Ernst Alberto Lovato Damiano Macedonio
Ciprian Spiridon Fausto Spoto

University of Washington, USA & University of Verona, Italy & Julia Srl, Italy

Suva, November 25, 2015, LPAR

Servlets and Their Parameters



Servlet Code

```
public class MyServlet extends HttpServlet {
  void doGet(HttpServletRequest request, HttpServletResponse response) {
    String city = request.getParameter("city");
    String month = request.getParameter("month");
    .....
    PrintWriter out = response.getWriter();
    out.println("<p>this goes to the browser</p>");
    .....
  }
}
```

The Risk of Injections

Servlets allow user input to flow through the code

- input should flow to as fewer places as possible
- input should be checked for validity (*sanitized*)

Unconstrained flow of input into sensitive program statements poses a security risk

Here we deal with the flow issue (taintedness analysis)

Top SW Errors according to CWE/SANS 2011

`http://cwe.mitre.org/top25/#Listing`

Rank	Score	Id	Name
1	93.8	CWE-89	SQL Injection
2	83.3	CWE-78	OS Command Injection
3	79.0	CWE-120	Buffer Overflow
4	77.7	CWE-79	Cross-site Scripting
...			
10	73.8	CWE-807	Untrusted Inputs in Security Decision
...			
16	66.0	CWE-829	Inclusion of Untrusted Functionality
...			
22	61.1	CWE-601	Open Redirect

Example 1/2

```
public class MyServlet extends HttpServlet {
  void doGet(HttpServletRequest request, HttpServletResponse response) {
    String user = request.getParameter("user"); (A)
    String url = "jdbc:mysql://192.168.2.128:3306/anvayaV2";
    Class.forName("com.mysql.jdbc.Driver").newInstance(); (B)
    try (Connection conn = DriverManager.getConnection(url, "root", "");
         PrintWriter out = response.getWriter()) { (C)
      Statement st = conn.createStatement();
      String query = wrapQuery(user); (D)
      out.println("Query : " + query); (E)
      ResultSet res = st.executeQuery(query); (F)
      out.println("Results:");
      while (res.next())
        out.println("\t\t" + res.getString("address")); (G)
      st.executeQuery(wrapQuery("dummy")); (H)
    }
  }
  private String wrapQuery(String s) {
    return "SELECT * FROM User WHERE userId='" + s + "'";
  }
}
```

Example 2/2

Actual vulnerabilities:

- SQL injection at (F)

```
ResultSet res = st.executeQuery(query);
```

- Cross-site scripting injections at (E) and (G)

```
out.println("Query : " + query);
out.println("\t\t" + res.getString("address"));
```

	SQL	XSS
actual	(F)	(E) (G)
FindBugs	(F)	
Google CodePro Analytix	(F) (H)	(E) (G)
HP Fortify SCA	(F)	(E)
Julia	(F)	(E) (G)

Our Goal

1. formalize taintedness for variables of reference type
2. define taintedness analysis for Java bytecode, through abstract interpretation
3. implement that analysis through binary decision diagrams
4. experiment and compare the results (soundness/precision)

Taintedness for Variables of Reference Type

The result of `wrapQuery()` is as tainted as the parameter:

```
private String wrapQuery(String s) {
  return "SELECT * FROM User WHERE userId='" + s + "'
}
```

What does "Tainted" Mean for a String?

- the pointer itself is not tainted information
- the field `char[] String.value` can contain tainted data
  - there is no fixed partition of the fields into tainted or untainted
  - a string can be tainted and, at the same time, other strings can be untainted

Object-sensitive Taintedness based on Reachability

- a primitive value is tainted if it is computed from tainted information
- a reference value is tainted if it is possible to reach a tainted value from it (in memory, by following its fields)

As all notions based on reachability, ours is sensitive to side-effects and hence more difficult to analyze statically than a property based on the value immediately bound to each variable only

- encapsulation and immutable types such as strings simplify the job

Formalization of Our Notion of Taintedness

We use a concrete semantics that explicitly tags data injected as user input. We represent such tainted data as boxed values

Tainted Value

Let $v \in \mathbb{Z} \cup \boxed{\mathbb{Z}} \cup \mathbb{L} \cup \{\texttt{null}\}$ be a value.
Let μ be a memory.
The property of being *tainted* for v in μ is defined as:

1. $v \in \boxed{\mathbb{Z}}$, or
2. v is a location, $o = \mu(v)$ is the object at that location and there is a field f such that its value $o(f)$ is tainted in μ

Selection of Tainted Variables in a State

JVM states σ contain i local variables and j stack elements. Exceptional states are underlined and have a single ($j = 1$) stack element: the reference to the exception object

Tainted Variables

$$
tainted(\sigma) = \begin{cases}
\{ l_k \mid l[k] \text{ is tainted in } \mu,\ 0 \le k < i \} \\
\quad \cup \{ s_k \mid v_k \text{ is tainted in } \mu,\ 0 \le k < j \} \\
\qquad \text{if } \sigma = \langle l \parallel v_{j-1} :: \cdots :: v_0 \parallel \mu \rangle \\
\{ l_k \mid l[k] \text{ is tainted in } \mu,\ 0 \le k < i \} \cup \{ e, s_0 \} \\
\qquad \text{if } \sigma = \underline{\langle l \parallel v_0 \parallel \mu \rangle} \text{ and } v_0 \text{ is tainted in } \mu \\
\{ l_k \mid l[k] \text{ is tainted in } \mu,\ 0 \le k < i \} \cup \{ e \} \\
\qquad \text{if } \sigma = \underline{\langle l \parallel v_0 \parallel \mu \rangle} \text{ and } v_0 \text{ is not tainted in } \mu
\end{cases}
$$

Abstract Domain of Boolean Formulas

A Boolean variable l_k or s_k is true iff the corresponding local variable or stack element holds a tainted value

The taintedness abstract domain is the set of Boolean formulas over

input state

output state

$$\{\check{e}, \hat{e}\} \cup \{\check{l}_k \mid 0 \leq k\} \cup \{\check{s}_k \mid 0 \leq k\} \cup \{\hat{l}_k \mid 0 \leq k\} \cup \{\hat{s}_k \mid 0 \leq k\}$$

Concretization Map

$$\gamma(\phi) = \left\{ \text{denotation } \delta \;\middle|\; \begin{array}{l} \text{for all states } \sigma \text{ s.t. } \delta(\sigma) \text{ is defined} \\ \check{tainted}(\sigma) \cup \hat{tainted}(\delta(\sigma)) \models \phi \end{array} \right\}$$

Abstraction of each Bytecode Instruction 1/3

Each bytecode instruction is abstracted into a Boolean formula whose model is consistent with the propagation of taintedness

`const v`

$U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge \neg\hat{s}_j$

`load k`

$U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{l}_k \leftrightarrow \hat{s}_j)$

`store k`

$U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{l}_k)$

with a frame condition

$$U = \wedge_{v \in L}(\check{v} \leftrightarrow \hat{v}) \wedge (\neg\hat{e} \rightarrow \wedge_{v \in S}(\check{v} \leftrightarrow \hat{v}))$$

Abstraction of each Bytecode Instruction 2/3

add

$U \wedge \neg \check{e} \wedge \neg \hat{e} \wedge (\hat{s}_{j-2} \leftrightarrow (\check{s}_{j-2} \vee \check{s}_{j-1}))$

new k

$U \wedge \neg \check{e} \wedge (\neg \hat{e} \to \neg \hat{s}_j) \wedge (\hat{e} \to \neg \hat{s}_0)$

throw

$U \wedge \neg \check{e} \wedge \hat{e} \wedge (\hat{s}_0 \to \check{s}_{j-1})$

catch

$U \wedge \check{e} \wedge \neg \hat{e}$

Abstraction of each Bytecode Instruction 3/3

For reading a field, we exploit our notion of taintedness based on reachability to get an object-sensitive approximation

getfield f

$$U \wedge \neg\check{e} \wedge (\neg\hat{e} \to (\hat{s}_{j-1} \to \check{s}_{j-1})) \wedge (\hat{e} \to \neg\hat{s}_0)$$

For writing into a field, we must conservatively foresee all possible side-effects on data reachable from the variables

putfield f

$$\wedge_{v \in L} R_j(v) \wedge (\neg\hat{e} \to \wedge_{v \in S} R_j(v)) \wedge (\hat{e} \to \neg\hat{s}_0) \wedge \neg\check{e}$$

where we use a preliminary reachability analysis in

$$R_j(v) = \begin{cases} \check{v} \leftrightarrow \hat{v} & \text{if } \neg \mathit{reach}(v, s_{j-2}) \\ (\check{v} \vee \check{s}_{j-1}) \leftarrow \hat{v} & \text{if } \mathit{reach}(v, s_{j-2}) \end{cases}$$

The Approximation of Method Calls

A Denotational Approach

- we start from the denotation ϕ of the callee(s)
- we plug ϕ at the calling point
  - by renaming callee's formal arguments into caller's actual arguments
  - by renaming the returned value into the result of the call
  - caller's variables that share with at least an argument that might be side-effected get involved in a worst-case assumption

Abstract Compositional Semantics

Sequential Composition

$\phi_1 ;^{\mathbb{T}} \phi_2 = \exists_{\overline{V}}(\phi_1[\overline{V}/\hat{V}] \wedge \phi_2[\overline{V}/\check{V}])$

Disjunctive Composition

$\phi_1 ;^{\mathbb{T}} \phi_2 = \phi_1 \vee \phi_2$

Fixpoint

- A fixpoint is needed to build the abstract semantics by saturating all execution paths of loops and recursion
- The fixpoint is reached in a finite number of iterations since there is a finite number of (equivalence classes of) Boolean formulas over a finite number of variables (those in scope at each given program point)

A Sound Framework of Analysis

Sources Program variables corresponding to sources of tainted data (user input) are forced to true in the Boolean formulas

Sinks Specific variables where tainted data must not flow are observed to see if the Boolean formulas entail them to be true

Soundness

We have a formal statement of soundness for the abstraction of each single bytecode instruction and for the operators for sequential and disjunctive composition

Sources and Sinks

Sources of tainted data

- servlet requests
- console read methods
- database operations
- manually annotated as `@Untrusted`

Methods that must never receive tainted data

- SQL query methods
- servlet output methods
- library loading methods
- reflective operations
- manually annotated as `@Trusted`

Field Sensitivity

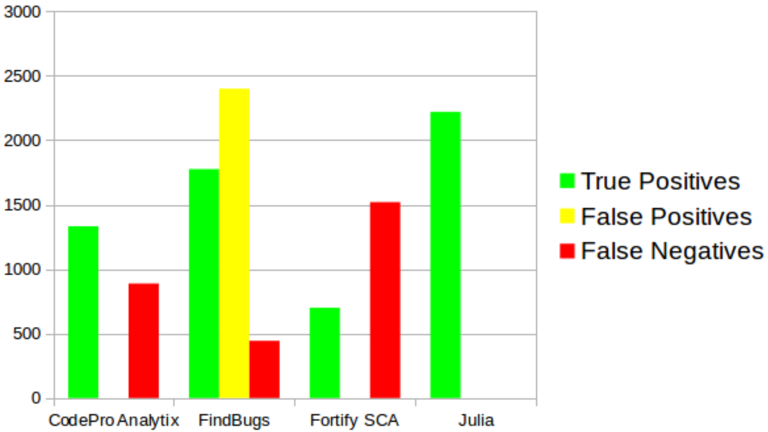
According to our Boolean approximation for `getfield`, if an object is assumed to be tainted, then all its fields are conservatively assumed to be tainted.

This is object-sensitive but field-insensitive.

It is possible to build a field-sensitive analysis through a greatest fixpoint computation of an oracle of fields assumed to be always untainted, for all objects.

Experiments have shown that field-sensitivity does not actually increase the precision of the analysis.

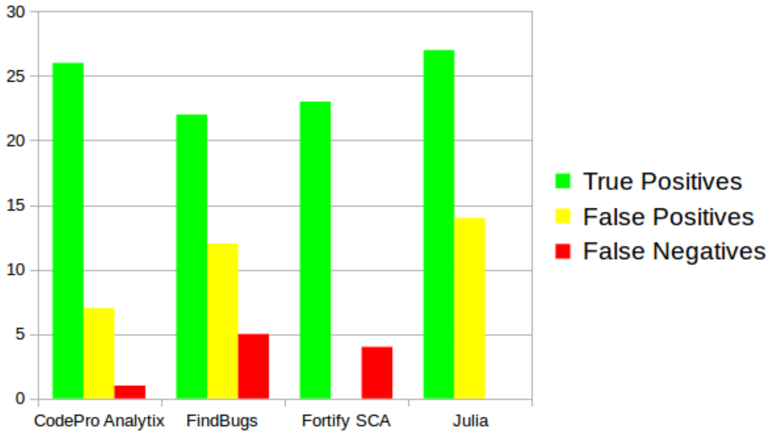
Identification of SQL-Injections: CWE89



Times in minutes

CodePro A.: 20 FindBugs: 2 Fortify SCA: 3600 Julia: 79

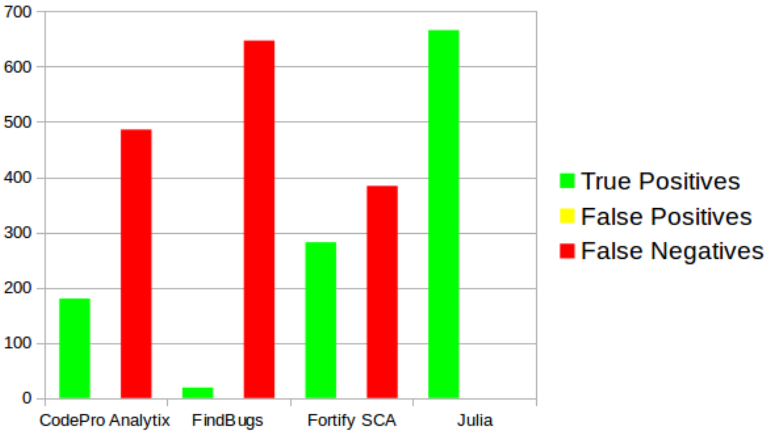
Identification of SQL-Injections: WebGoat



Times in minutes

CodePro A.: 1 FindBugs: 20 Fortify SCA: 164 Julia: 3

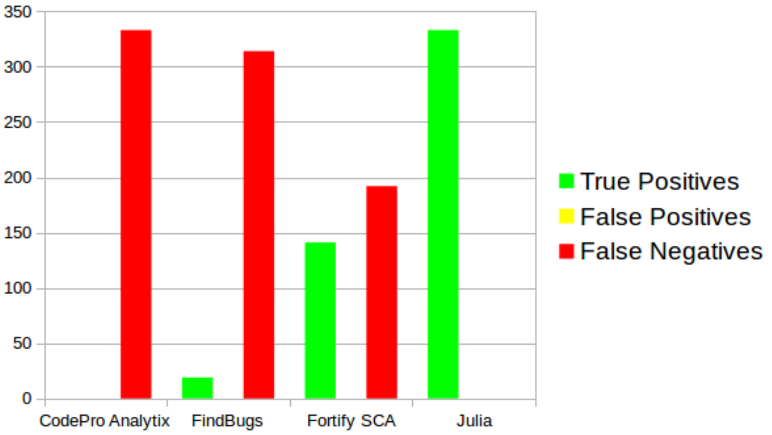
Identification of XSS-Injections: CWE80



Times in minutes

CodePro A.: 9 FindBugs: < 1 Fortify SCA: 590 Julia: 5

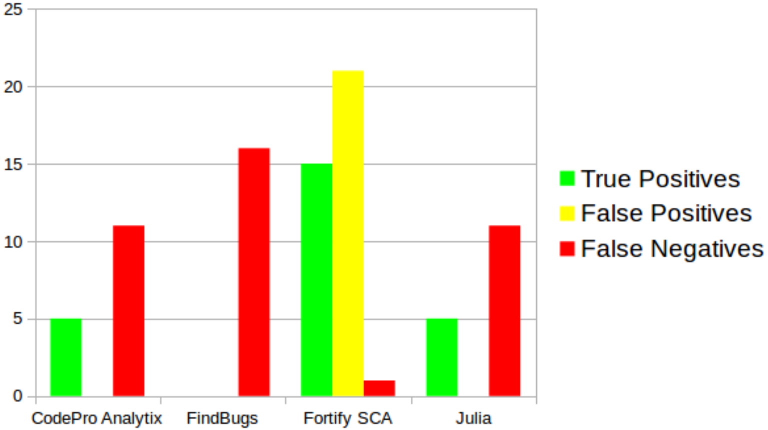
Identification of XSS-Injections: CWE81



Times in minutes

CodePro A.: < 1 FindBugs: < 1 Fortify SCA: 303 Julia: 3

Identification of XSS-Injections: WebGoat 1/2



Times in minutes

CodePro A.: 1 FindBugs: < 1 Fortify SCA: 164 Julia: 3

False Negatives for a Sound Analysis?

A sound static analysis should never have false negatives (real bugs that are not found by the analysis)

Java Server Pages (JSP)

- browser pages made up of a mixture of HTML and Java code, processed by a servlet container such as Tomcat
- Tomcat uses Jasper to compile JSP on-the-fly into Java source that gets compiled into Java bytecode and run
- JSP compiled code is not available to Julia and its entry points of tainted data are unkown to Julia

We have manually run Jasper/javac to get the Java bytecode of the JSP. With that, Julia's analysis finds all bugs, with no false negatives anymore

Identification of XSS-Injections: WebGoat 2/2

Here all tools have received the classes compiled with Jasper

Times in minutes

CodePro A.: 1 FindBugs: < 1 Fortify SCA: 164 Julia: 3

Conclusion

Contributions

- a new notion of taintedness for reference types
- taintedness analysis in Boolean form
- efficient implementation with BDDs
- runs on real software with good results

Next steps

- automatic identification of entry points of tainted data for Java frameworks
- extension to Android