Boolean Formulas for the Static Identification of Injection Attacks in Java

Michael D. Ernst[1], Alberto Lovato[2], Damiano Macedonio[3], Ciprian Spiridon[3], and Fausto Spoto[2,3]

[1] University of Washington, Seattle, USA
[2] Dipartimento di Informatica, Università di Verona, Italy
[3] Julia Srl, Verona, Italy

Abstract. The most dangerous security-related software errors, according to CWE 2011, are those leading to injection attacks — user-provided data that result in undesired database access and updates (*SQL-injections*), dynamic generation of web pages (*cross-site scripting-injections*), redirection to user-specified web pages (*redirect-injections*), execution of OS commands (*command-injections*), class loading of user-specified classes (*reflection-injections*), and many others. This paper describes a flow- and context-sensitive static analysis that automatically identifies if and where injections of tainted data can occur in a program. The analysis models explicit flows of tainted data. Its notion of taintedness applies also to reference (non-primitive) types dynamically allocated in the heap, and is object-sensitive and field-sensitive. The analysis works by translating the program into Boolean formulas that model all possible flows. We implemented it within the Julia analyzer for Java and Android. Julia found injection security vulnerabilities in the Internet banking service and in the customer relationship management of a large Italian bank.

1 Introduction

Dynamic web pages and web services react to user input coming from the network, and this introduces the possibility of an attacker *injecting* special text that induces unsafe, unexpected behaviors of the program. Injection attacks are considered the most dangerous software error [19] and can cause free database access and corruption, forging of web pages, loading of classes, denial-of-service, and arbitrary execution of commands. Most analyses to spot such attacks are dynamic and unsound (see Sec. 3).

This article defines a sound static analysis that identifies if and where a Java bytecode program lets data flow from *tainted* user input (including servlet requests) into critical operations that might give rise to injections. Data flow is a prerequisite to injections, but the user of the analysis must later gage the actual risk of the flow. Namely, analysis approximations might lead to false alarms and proper input validation might make actual flows harmless.

Our analysis works by translating Java bytecode into Boolean formulas that express all possible explicit flows of tainted data. The choice of Java bytecode simplifies the semantics and its abstraction (many high-level constructs must not be explicitly considered) and lets us analyze programs whose source code is not available, as is typically the case in industrial contexts that use software developed by third parties, such as banks.

Our contributions are the following:

– an object-sensitive formalization of taintedness for reference types, based on reachability of tainted information in memory;
– a flow-, context- and field-sensitive static analysis for explicit flows of tainted information based on that notion of taintedness, which is able to deal with data dynamically allocated in the heap (not just primitive values);
– its implementation inside the Julia analyzer, through binary decision diagrams, and its experimental evaluation.

Sec. 6 shows that our analysis can analyze large real Java software. Compared to other tools available on the market, ours is the only one that is sound, yet precise and efficient. Our analysis is limited to explicit flows [25]; as is common in the literature, it does not yet consider implicit flows (arising from conditional tests) nor hidden flows (such as timing channels). In particular, considering implicit flows is relatively simple future work (we could apply our previous work [10], unchanged) but would likely degrade the precision of the analysis of real software.

This article is organized as follows. Sec. 2 gives an example of injection and clarifies the importance of a new notion of taintedness for values of reference type. Sec. 3 discusses related work. Sec. 4 defines a concrete semantics for Java bytecode. Sec. 5 defines our new object-sensitive notion of taintedness for values of reference type and its use to induce an object- and field-sensitive abstract interpretation of the concrete semantics. Sec. 6 presents experiments with the implementation of the analysis. Extended definitions and proofs are in a technical report [8].

2 Example

Fig. 1 is a Java servlet that suffers from SQL-injection and cross-site scripting-injection attacks. (For brevity, the figure omits exception-handling code.)

A *servlet* (lines 1 and 2) is code that listens to HTTP network connection requests, retrieves its parameters, and runs some code in response to each request. The response (line 2) may be presented as a web page, XML, or JSON. This is a standard way of implementing dynamic web pages and web services. The user of a servlet connects to the web site and provides the parameters through the URL, as in `http://my.site.com/myServlet?user=spoto`. Code retrieves these through the `getParameter` method (line 5). Lines 9 and 10 establish a connection to the database of the application, which is assumed to define a table `User` (line 27) of the users of the service. Line 27 builds an SQL query from the user name provided as parameter. This query is reported to the response (line 15) and executed (line 17). The result is a relational table of all users matching the

```
 1  public class MyServlet extends HttpServlet {
 2    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
 3      response.setContentType("text/html;charset=UTF-8");
 4
 5      String user = request.getParameter("user"), url = "jdbc:mysql://192.168.2.128:3306/";
 6      String dbName = "anvayaV2", driver = "com.mysql.jdbc.Driver";
 7      String userName = "root", password = "";
 8
 9      Class.forName(driver).newInstance();
10      try (Connection conn = DriverManager.getConnection(url + dbName, userName, password);
11           PrintWriter out = response.getWriter()) {
12
13        Statement st = conn.createStatement();
14        String query = wrapQuery(user);
15        out.println("Query : " + query);
16
17        ResultSet res = st.executeQuery(query);
18        out.println("Results:");
19        while (res.next())
20          out.println("\t\t" + res.getString("address"));
21
22        st.executeQuery(wrapQuery("dummy"));
23      }
24    }
25
26    private String wrapQuery(String s) {
27      return "SELECT * FROM User WHERE userId=’" + s + "’";
28    }
29  }
```

Fig. 1. A Java servlet that suffers from SQL and cross-site scripting-injections.

given criterion (the `user` parameter might be a specific name or a wildcard that matches more users). This table is then printed to the response (lines 17–20).

The interesting point here is that the user of this servlet is completely free to specify the value of the `user` parameter. In particular, she can provide a string that actually lets line 17 run *any* possible database command, including malicious commands that erase its content or insert new rows. For instance, if the user supplies the string "`'; DROP TABLE User; --`" as `user`, the resulting concatenation is an SQL command that erases the `User` table from the database. In literature, this is known as an SQL-injection attack and follows from the fact that user (*tainted*) input flows from the `request` *source* into the `executeQuery` *sink* method. There is no SQL-injection at line 22, although it looks very much like line 17, since the query there is not computed from user-provided input.

Another risk exists at lines 15 and 20. There, data is printed to the response object, and is typically interpreted by the client as HTML contents. A malicious user might have provided a `user` parameter that contains arbitrary HTML tags, including tags that will let the client execute scripts (such as Javascript). This might result in evil. For instance, if the user injects a crafted URL such as "`http://my.site.com/myServlet?user=<script>malicious</script>`", the parameter `user` holds "`<script>malicious</script>`". At line 15 this code is sent to the user's browser and interpreted as Javascript, running any `malicious` Javascript. In literature, this is known as cross-site scripting-injection and follows from the fact that user (*tainted*) input from the `request` *source* flows into the *sink* output writer of the response object. The same might happen at line 20, where the flow is more complex: in other parts of the application, the user might save her address to the database and store malicious code instead; line 20 will fetch this malicious code and send it to the browser of the client to run it.

Many kinds of injections exist. They arise from information flows from what the user can specify (the parameter of the request, input from console, data on a database) to specific methods, such as `executeQuery` (SQL-injection), `print` (cross-site scripting-injection), reflection methods (that allow one to load any class or execute any method and lead to a reflection-injection), `execute` (that allows one to run any operating system command and leads to a command-injection), etc. This article focuses on the identification of flows of tainted information, not on the exact enumeration of sources and sinks. Our approach can be instantiated from well-known lists of sources and sinks in the literature.

3 Related Work

The identification of possible injections and the inference of information flows are well-studied topics. Nevertheless, no previous sound techniques work on real Java code, even only for explicit flows. Most injection identification techniques are dynamic and/or unsound. Existing static information-flow analyses are not satisfactory for languages with reference types.

Identification of Injections. Data injections are security risks, so there is high industrial and academic interest in their automatic identification. Here, we have space to mention only the most recent works regarding SQL-injection. Almost all techniques aim at the dynamic identification of the injection when it occurs [14, 12, 18, 35, 21, 7, 30, 28] or at the generation of test cases of attacks [1, 17] or at the specification of good coding practices [29].

By contrast, static analysis has the advantage of finding the vulnerabilities before running the code, and a sound static analysis *proves* that injections *only* occur where it issues a warning. A static analysis is *sound* if it finds all places where an injection might occur (for instance, it must spot line 17 in Fig. 1); it is *precise* if it minimizes the number of false alarms (for instance, it should not issue a warning at line 22 in Fig. 1). Beyond Julia, static analyzers that identify injections in Java are FindBugs (`http://findbugs.sourceforge.net`), Google's CodePro Analytix (`https://developers.google.com/java-dev-tools/codepro`), and HP Fortify SCA (on-demand web interface at `https://trial.hpfod.com/Login`). These tools do not formalize the notion of *taintedness* (as we do in Def. 4). For the example in Fig. 1, Julia is correct and precise: it warns at lines 15, 17, and 20 but not at 22; FindBugs incorrectly warns at line 17 only; Fortify SCA incorrectly warns at lines 15 and 17 only; CodePro Analytix warns at lines 15, 17, 20, and also, imprecisely, at the harmless line 22. Sec. 6 compares those tools with Julia in more detail. We also cite FlowDroid [2], that however works for Android packages, not on Java bytecode, and TAJ [33], that is part of a commercial product. Neither comes with a soundness proof nor a definition of taintedness for variables of reference type.

Modelling of Information Flow. Many static analyses model explicit and often also implicit information flows [25] in Java-like or Java bytecode programs.

There are data/control-flow analyses [5, 15, 26, 20]; type-based analyses [31, 34, 3, 4, 13, 9] and analyses based on abstract interpretation [10]. They are satisfactory for variables of primitive type but impractical for heap-allocated data of reference type, such as strings. Most analyses [4, 5, 13, 9, 15, 20, 26, 34] assume that the language has only primitive types; others [3, 10] are object-insensitive, *i.e.*, for each field f, assume that $a.f$ and $b.f$ are both tainted or both untainted, regardless of the container objects a and b. Even if a user specifies, by hand, which f is tainted (unrealistic for thousands of fields, including those used in the libraries), object-insensitivity leads to a very coarse abstraction that is industrially useless. Consider the `String` class, which holds its contents inside a `private final char[] value` field. If any string's `value` field is tainted, then every string's `value` field must be tainted, and this leads to an alarm at every use of strings in a sensitive context in the program, many of which may be false alarms. The problem applies to any data structure that can carry tainted data, not just strings. Our analysis uses an object-sensitive and *deep* notion of taintedness, that fits for heap-allocated data of reference type. It can be considered as data-flow, formalized through abstract interpretation. This has the advantage of providing its correctness proof in a formal and standard way.

4 Denotational Semantics of Java Bytecode

This section presents a denotational semantics for Java bytecode, which we will use to define an abstraction for taintedness analysis (Sec. 5). The same semantics has been used for nullness analysis [32] and has been proved equivalent [23] to an operational semantics. The only difference is that, in this article, primitive values are decorated with their taintedness.

We assume a Java bytecode program P given as a collection of graphs of *basic blocks* of code, one for each method. Bytecodes that might throw exceptions are linked to a handler starting with a `catch`, possibly followed by bytecodes selecting the right kind of exception. For simplicity, we assume that the only primitive type is `int` and the only reference types are *classes*; we only allow *instance* fields and methods; and method parameters cannot be reassigned inside their body. Our implementation handles full Java bytecode.

Definition 1 (Classes). *The set of* classes $\mathbb{K}$ *is partially ordered w.r.t. the* subclass relation $\leq$. *A* type *is an element of* $\mathbb{K} \cup \{$`int`$\}$. *A class* $\kappa \in \mathbb{K}$ *defines* instance fields $\kappa.f : t$ *(field f of type t defined in κ) and* instance methods $\kappa.m(t_1, \ldots, t_n) : t$ *(method m with arguments of type $t_1, \ldots, t_n$, returning a value of type t, possibly* `void`*). We consider constructors as methods returning* `void`*. If it does not introduce confusion, we write f and m for fields and methods.*

A *state* provides *values* to program variables. *Tainted* values are computed from servlet/user input; others are *untainted.* Taintedness for reference types (such as string `request` in Fig. 1) will be defined later as a reachability property from the reference (Def. 4); primitive tainted values are explicitly marked in the state.

Definition 2 (State). *A* value *is an element of* $\mathbb{Z}\cup\boxed{\mathbb{Z}}\cup\mathbb{L}\cup\{\mathbf{\textit{null}}\}$*, where* $\mathbb{Z}$ *are untainted integers,* $\boxed{\mathbb{Z}}$ *are tainted integers, and* $\mathbb{L}$ *is a set of* locations. *A* state *is a triple* $\langle l \,\|\, s \,\|\, \mu\rangle$ *where* l *are the values of the* local variables*,* s *the values of the* operand stack*, which grows leftwards, and* μ *a* memory *that binds locations to* objects*. The empty stack is written* ε*. Stack concatenation is* $::$ *with* $s::\varepsilon$ *written as just* s*. An object* o *belongs to class* $o.\kappa \in \mathbb{K}$ *(is an* instance *of* $o.\kappa$*) and maps identifiers (the fields* f *of* $o.\kappa$ *and of its superclasses) into values* $o.f$*. The set of states is* Ξ*. We write* $\Xi_{i,j}$ *when we want to fix the number* i *of local variables and* j *of stack elements. A value* v has type t *in a state* $\langle l \,\|\, s \,\|\, \mu\rangle$ *if* $v \in \mathbb{Z}\cup\boxed{\mathbb{Z}}$ *and* $t = \mathbf{\textit{int}}$*, or* $v = \mathbf{\textit{null}}$ *and* $t \in \mathbb{K}$*, or* $v \in \mathbb{L}$*,* $t \in \mathbb{K}$ *and* $\mu(v).\kappa \le t$*.*

Example 1. Let state $\sigma = \langle[3, \mathtt{null}, \boxed{4}, \ell] \,\|\, \boxed{3} :: \ell'' :: \ell'' \,\|\, \mu\rangle \in \Xi_{4,3}$, with $\mu = [\ell \mapsto o, \ell' \mapsto o', \ell'' \mapsto o'']$, $o.f = \ell'$, $o.g = 13$, $o'.g = \boxed{17}$ and $o''.g = 10$. Local 0 holds the integer 3 and local 2 holds the integer 4, marked as computed from servlet/user input. The top of the stack holds 3, marked as computed from servlet/user input. The next two stack elements are aliased to ℓ''. Location ℓ is bound to object o, whose field f holds ℓ' and whose field g holds the untainted integer 13. Location ℓ' is bound to o' whose field g holds a tainted integer $\boxed{17}$. Location ℓ'' is bound to o'' whose field g holds the untainted value 10.

The Java Virtual Machine (JVM) allows exceptions. Hence we distinguish *normal* states $\sigma \in \Xi$, arising during the normal execution of a piece of code, from *exceptional* states $\underline{\sigma} \in \underline{\Xi}$, arising *just after* a bytecode that throws an exception. The latter have only one stack element, *i.e.*, the location of the thrown exception object, also in the presence of nested exception handlers [16]. The semantics of a bytecode is then a *denotation* from an *initial* to a *final* state.

Definition 3 (JVM State and Denotation). *The set of* JVM states *(from now just* states*) with* i *local variables and* j *stack elements is* $\Sigma_{i,j} = \Xi_{i,j} \cup \underline{\Xi}_{i,1}$*. A* denotation *is a partial map from an* input *or* initial *state to an* output *or* final *state; the set of denotations is* Δ *or* $\Delta_{i_1,j_1\to i_2,j_2} = \Sigma_{i_1,j_1} \to \Sigma_{i_2,j_2}$ *to fix the number of local variables and stack elements. The* sequential composition *of* $\delta_1, \delta_2 \in \Delta$ *is* $\delta_1;\delta_2 = \lambda\sigma.\delta_2(\delta_1(\sigma))$*, which is undefined when* $\delta_1(\sigma)$ *or* $\delta_2(\delta_1(\sigma))$ *is undefined.*

In $\delta_1;\delta_2$, the idea is that δ_1 describes the behaviour of an instruction ins_1, δ_2 that of an instruction ins_2 and $\delta_1;\delta_2$ that of the execution of ins_1 and then ins_2.

At each program point, the number i of local variables and j of stack elements and their types are statically known [16], hence we can assume the semantics of the bytecodes undefined for input states of wrong sizes or types. Readers can find the denotations of bytecode instructions in a technical report [8], together with the construction of the concrete fixpoint collecting semantics of Java bytecode, explicitly targeted at abstract interpretation, since it only requires to abstract three concrete operators ;, $\cup$, and *extend* on $\wp(\Delta)$, *i.e.*, on the subsets of Δ and the denotation of each single bytecode distinct from `call`. The operator *extend* plugs a method's denotation at its calling point and implements `call`. The concrete fixpoint computation is in general infinite, but its abstractions converge in a finite number of steps if, as in Sec. 5, the abstract domain has no infinite ascending chain.

5 Taintedness Analysis

This section defines an abstract interpretation [6] of the concrete semantics of Sec. 4, whose abstract domain is made of Boolean formulas whose models are consistent with all possible ways of propagating taintedness in the concrete semantics. The concrete semantics works over $\wp(\Delta)$ and is built from singletons (sets made of a single $\delta \in \Delta$), one for each bytecode, with three operators ;, $\cup$, and *extend*. Hence we define here correct abstractions of those sets and operators.

Our analysis assumes that three other analyses have been performed in advance. (1) $reach(v, v')$ is true if (the location held in) v' is reachable from (the location held in) v. (2) $share(v, v')$ is true if from v and v' one can reach a common location. (3) $updated_M(l_k)$ is true if some call in the program to method M might ever modify an object reachable from local variable l_k. All three analyses are conservative overapproximations of the actual (undecidable) relations. Our implementation computes these predicates as in [22], [27], and [11], respectively.

Primitive values are explicitly marked as tainted (Def. 2), while taintedness for references is indirectly defined in terms of reachability of tainted values. Hence, this notion allows $a.f$ and $b.f$ to have distinct taintedness, depending of the taintedness of variables a and b (object-sensitivity).

Definition 4 (Taintedness). *Let $v \in \mathbb{Z} \cup \boxed{\mathbb{Z}} \cup \mathbb{L} \cup \{\texttt{null}\}$ be a value and μ a memory. The property of being* tainted *for v in μ is defined recursively as: $v \in \boxed{\mathbb{Z}}$ or ($v \in \mathbb{L}$ and $o = \mu(v)$ and there is a field f such that $o(f)$ is tainted in μ).*

A first abstraction step selects the variables that, in a state, hold tainted data. It yields a logical model where a variable is true if it holds tainted data.

Definition 5 (Tainted Variables). *Let $\sigma \in \Sigma_{i,j}$. Its* tainted variables *are*

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

Example 2. Consider σ from Ex. 1. We have $tainted(\sigma) = \{l_2, l_3, s_2\}$, since tainted data is reachable from both locations ℓ and ℓ', but not from ℓ''.

To make the analysis flow-sensitive, distinct variables abstract the input (marked with $\check{}$) and output (marked with $\hat{}$) of a denotation. If S is a set of identifiers, then $\check{S} = \{\check{v} \mid v \in S\}$ and $\hat{S} = \{\hat{v} \mid v \in S\}$. The abstract domain contains Boolean formulas that constraint the relative taintedness of local variables and stack elements. For instance, $\check{l}_1 \to \hat{s}_2$ states that if local variable l_1 is tainted in the input of a denotation, then the stack element s_2 is tainted in its output.

$$(const\ \ v)^{\mathbb{T}} = U \land \neg\check{e} \land \neg\hat{e} \land \neg\hat{s}_j \qquad (load\ \ k\ \ t)^{\mathbb{T}} = U \land \neg\check{e} \land \neg\hat{e} \land (\check{l}_k \leftrightarrow \hat{s}_j)$$

$$(store\ \ k\ \ t)^{\mathbb{T}} = U \land \neg\check{e} \land \neg\hat{e} \land (\check{s}_{j-1} \leftrightarrow \hat{l}_k) \qquad (add)^{\mathbb{T}} = U \land \neg\check{e} \land \neg\hat{e} \land (\hat{s}_{j-2} \leftrightarrow (\check{s}_{j-2} \lor \check{s}_{j-1}))$$

$$(throw\ \ \kappa)^{\mathbb{T}} = U \land \neg\check{e} \land \hat{e} \land (\hat{s}_0 \to \check{s}_{j-1}) \qquad (new\ \ \kappa)^{\mathbb{T}} = U \land \neg\check{e} \land (\neg\hat{e} \to \neg\hat{s}_j) \land (\hat{e} \to \neg\hat{s}_0)$$

$$(catch)^{\mathbb{T}} = U \land \check{e} \land \neg\hat{e} \qquad (getfield\ \ \kappa.f\,{:}\,t)^{\mathbb{T}} = U \land \neg\check{e} \land (\neg\hat{e} \to (\hat{s}_{j-1} \to \check{s}_{j-1})) \land (\hat{e} \to \neg\hat{s}_0)$$

$$(putfield\ \ \kappa.f\,{:}\,t)^{\mathbb{T}} = \land_{v \in L} R_j(v) \land (\neg\hat{e} \to \land_{v \in S} R_j(v)) \land (\hat{e} \to \neg\hat{s}_0) \land \neg\check{e}.$$

Fig. 2. Bytecode abstraction for taintedness, in a program point with j stack elements. Bytecodes not reported in this figure are abstracted into the default $U \land \neg\check{e} \land \neg\hat{e}$.

Definition 6 (Taintedness Abstract Domain $\mathbb{T}$). *Let $i_1, j_1, i_2, j_2 \in \mathbb{N}$. The taintedness abstract domain $\mathbb{T}_{i_1,j_1 \to i_2,j_2}$ is the set of Boolean formulas over $\{\check{e}, \hat{e}\} \cup \{\check{l}_k \mid 0 \le k < i_1\} \cup \{\check{s}_k \mid 0 \le k < j_1\} \cup \{\hat{l}_k \mid 0 \le k < i_2\} \cup \{\hat{s}_k \mid 0 \le k < j_2\}$ (modulo logical equivalence).*

Example 3. $\phi = (\check{l}_1 \leftrightarrow \hat{l}_1) \land (\check{l}_2 \leftrightarrow \hat{l}_2) \land (\check{l}_3 \leftrightarrow \hat{l}_3) \land \neg\check{e} \land \neg\hat{e} \land (\check{s}_0 \leftrightarrow \hat{l}_0) \in \mathbb{T}_{4,1 \to 4,0}$.

The concretization map γ states that a $\phi \in \mathbb{T}$ abstracts those denotations whose behavior, *w.r.t.* the propagation of taintedness, is a model of ϕ.

Proposition 1 (Abstract Interpretation). *$\mathbb{T}_{i_1,j_1 \to i_2,j_2}$ is an abstract interpretation of $\wp(\Delta_{i_1,j_1 \to i_2,j_2})$ with $\gamma : \mathbb{T}_{i_1,j_1 \to i_2,j_2} \to \wp(\Delta_{i_1,j_1 \to i_2,j_2})$ given by*

$$\gamma(\phi) = \left\{ \delta \in \Delta_{i_1,j_1 \to i_2,j_2} \,\middle|\, \begin{array}{l} \textit{for all } \sigma \in \Sigma_{i_1,j_1} \textit{ s.t. } \delta(\sigma) \textit{ is defined} \\ \check{\textit{tainted}}(\sigma) \cup \hat{\textit{tainted}}(\delta(\sigma)) \models \phi \end{array} \right\}.$$

Example 4. Consider ϕ from Ex. 3 and bytecode `store 0` at a program point with $i = 4$ locals and $j = 1$ stack elements. Its denotation *store 0* $\in \gamma(\phi)$ since that bytecode does not modify locals 1, 2 and 3, hence their taintedness is unchanged $((\check{l}_1 \leftrightarrow \hat{l}_1) \land (\check{l}_2 \leftrightarrow \hat{l}_2) \land (\check{l}_3 \leftrightarrow \hat{l}_3))$; it only runs if no exception is thrown just before it $(\neg\check{e})$; it does not throw any exception $(\neg\hat{e})$; and the output local 0 is an alias of the topmost and only element of the input stack $(\check{s}_0 \leftrightarrow \hat{l}_0)$.

Fig. 2 defines correct abstractions for the bytecodes from Sec. 4, but `call`. A formula U (for *unchanged*) is a frame condition for input local variables and stack elements, that are also in the output and with unchanged value: their taintedness is unchanged. For the stack, this is only required when no exception is thrown, since otherwise the only output stack element is the exception.

Definition 7. *Let sets S (of stack elements) and L (of local variables) be the input variables that after all executions of a given bytecode in a given program point (only after the normal executions for S) survive with unchanged value. Then $U = \land_{v \in L}(\check{v} \leftrightarrow \hat{v}) \land (\neg\hat{e} \to \land_{v \in S}(\check{v} \leftrightarrow \hat{v}))$.*

Consider Fig. 2. Bytecodes run only if the preceding one does not throw any exception $(\neg\check{e})$ but `catch` requires an exception to be thrown $(\check{e})$. Bytecode `const` v pushes an untainted value on the stack: its abstraction says that no variable changes its taintedness (U), the new stack top is untainted ($\neg\hat{s}_j$) and `const` v never throws an exception ($\neg\hat{e}$). Most abstractions in Fig. 2 can be explained similarly. The result of `add` is tainted if and only if at least one operand is tainted ($\hat{s}_{j-2} \leftrightarrow (\check{s}_{j-2} \vee \check{s}_{j-1})$). For `new` κ, no variable changes its taintedness (U), if its execution does not throw any exception then the new top of the stack is an untainted new object ($\neg\hat{e} \rightarrow \neg\hat{s}_j$); otherwise the only stack element is an untainted exception ($\hat{e} \rightarrow \neg\hat{s}_0$). Bytecode `throw` κ always throws an exception ($\hat{e}$); if this is tainted, then the top of the initial stack was tainted as well ($\hat{s}_0 \rightarrow \check{s}_{j-1}$). The abstraction of `getfield` says that if it throws no exception and the value of the field is tainted, then the container of the field was tainted as well ($\neg\hat{e} \rightarrow (\hat{s}_{j-1} \rightarrow \check{s}_{j-1})$). This follows from the object-sensitivity of our notion of taintedness (Def. 4). Otherwise, the exception is untainted ($\hat{e} \rightarrow \neg\hat{s}_0$). For `putfield`, we cannot use U and must consider each variable v to see if it might reach the object whose field is modified ($\check{s}_{j-2}$). If that is not the case, v's taintedness is not affected ($\check{v} \leftrightarrow \hat{v}$); otherwise, if its value is tainted then either it was already tainted before the bytecode or the value written in the field was tainted ($(\check{v} \vee \check{s}_{j-1}) \leftarrow \hat{v}$). In this last case, we must use $\leftarrow$ instead of $\leftrightarrow$ since our reachability analysis is a *possible* approximation of actual (undecidable) reachability. This is expressed by formula $R_j(v)$, used in Fig. 2, where $R_j(v) = \check{v} \leftrightarrow \hat{v}$ if $\neg \mathit{reach}(v, s_{j-2})$, and $R_j(v) = (\check{v} \vee \check{s}_{j-1}) \leftarrow \hat{v}$, if $\mathit{reach}(v, s_{j-2})$.

Example 5. According to Fig. 2, the abstraction of `store 0` at a program point with $i = 4$ local variables and $j = 1$ stack elements is the formula ϕ of Ex. 3.

Example 6. Consider a `putfield f` at a program point p where there are $i = 4$ local variables, $j = 3$ stack elements and the only variable that reaches the receiver s_1 is the underlying stack element s_0. A possible state at p in Ex. 1. According to Fig. 2, the abstraction of that bytecode at p is $\phi' = (\check{l}_0 \leftrightarrow \hat{l}_0) \wedge (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\check{l}_2 \leftrightarrow \hat{l}_2) \wedge (\check{l}_3 \leftrightarrow \hat{l}_3) \wedge (\neg\hat{e} \rightarrow ((\check{s}_0 \vee \check{s}_2) \leftarrow \hat{s}_0)) \wedge (\hat{e} \rightarrow \neg\hat{s}_0) \wedge \neg\check{e} \in \mathbb{T}_{4,3\rightarrow 4,1}$.

Proposition 2. *The approximations in Fig. 2 are correct w.r.t. the denotations of Sec. 4, i.e., for all bytecode* `ins` *distinct from* `call` *we have* $\mathit{ins} \in \gamma(\mathit{ins}^{\mathbb{T}})$.

Denotations are composed by ; and their abstractions by $;^{\mathbb{T}}$. The definition of $\phi_1;^{\mathbb{T}}\phi_2$ matches the output variables of ϕ_1 with the corresponding input variables of ϕ_2. To avoid name clashes, they are renamed apart and then projected away.

Definition 8. *Let* $\phi_1, \phi_2 \in \mathbb{T}$. *Their* abstract sequential composition $\phi_1;^{\mathbb{T}}\phi_2$ *is* $\exists_{\overline{V}}(\phi_1[\overline{V}/\hat{V}] \wedge \phi_2[\overline{V}/\check{V}])$, *where* $\overline{V}$ *are fresh overlined variables.*

Example 7. Consider the execution of `putfield f` at program point p and then `store 0`, as in Ex. 6. The former is abstracted by ϕ' from Ex. 6; the latter by ϕ from Ex. 5. Their sequential composition is $\phi';^{\mathbb{T}}\phi = \exists_{\overline{V}}(\phi'[\overline{V}/\hat{V}] \wedge \phi[\overline{V}/\check{V}]) = \exists_{\overline{V}}([(\check{l}_0 \leftrightarrow \bar{l}_0) \wedge (\check{l}_1 \leftrightarrow \bar{l}_1) \wedge (\check{l}_2 \leftrightarrow \bar{l}_2) \wedge (\check{l}_3 \leftrightarrow \bar{l}_3) \wedge (\neg\bar{e} \rightarrow ((\check{s}_0 \vee \check{s}_2) \leftarrow \bar{s}_0)) \wedge (\bar{e} \rightarrow \neg\bar{s}_0) \wedge \neg\check{e}] \wedge [(\bar{l}_1 \leftrightarrow \hat{l}_1) \wedge (\bar{l}_2 \leftrightarrow \hat{l}_2) \wedge (\bar{l}_3 \leftrightarrow \hat{l}_3) \wedge \neg\bar{e} \wedge \neg\hat{e} \wedge (\bar{s}_0 \leftrightarrow \hat{l}_0)])$ which simplifies into $(\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\check{l}_2 \leftrightarrow \hat{l}_2) \wedge (\check{l}_3 \leftrightarrow \hat{l}_3) \wedge ((\check{s}_0 \vee \check{s}_2) \leftarrow \hat{l}_0) \wedge \neg\check{e} \wedge \neg\hat{e}$.

The second semantical operator is $\cup$ of two sets, approximated as $\cup^{\mathbb{T}} = \vee$. The third is *extend*, that makes the analysis context-sensitive by plugging the behavior of a method at each distinct calling context. Let ϕ approximate the taintedness behaviour of method $M = \kappa.m(t_1, \ldots, t_n) : t$; ϕ's variables are among $\check{l}_0, \ldots, \check{l}_n$ (the actual arguments including `this`), $\hat{s}_0$ (if M does not return `void`), $\hat{l}_0, \hat{l}_1 \ldots$ (the final values of M's local variables), $\check{e}$ and $\hat{e}$. Consider a call M at a program point where the $n+1$ actual arguments are stacked over other b stack elements. The operator plugs ϕ at the calling context: the return value $\hat{s}_0$ (if any) is renamed into $\hat{s}_b$; each formal argument $\check{l}_k$ of the callee is renamed into the actual argument $\check{s}_{k+b}$ of the caller; local variable $\hat{l}_k$ at the end of the callee is temporarily renamed into $\bar{l}_k$. Then a frame condition is built: the set $SA_{b,M,v}$ contains the formal arguments of the caller that might share with variable v of the callee at call-time and might be updated during the call. If this set is empty, then nothing reachable from v is modified during the call and v keeps its taintedness unchanged. This is expressed by the first case of formula $A_{b,M}(v)$. Otherwise, if v is tainted at the end of the call then either it was already tainted at the beginning or at least one of the variables in $SA_{b,M,v}$ has become tainted during the call. The second case of formula $A_{b,M}(v)$ uses the temporary variables to express that condition, to avoid name clashes with the output local variables of the caller. The frame condition for the b lowest stack elements of the caller is valid only if no exception is thrown, since otherwise the stack contains the exception object only. At the end, all temporary variables $\{\bar{l}_0, \ldots, \bar{l}_{i'}\}$ are projected away.

Definition 9. *Let $i, j \in \mathbb{N}$ and $M = \kappa.m(t_1, \ldots, t_n) : t$ with $j = b + n + 1$ and $b \geq 0$. We define $(\mathit{extend}_M^{i,j})^{\mathbb{T}}{:}\mathbb{T}_{n+1,0 \to i',r} \to \mathbb{T}_{i,j \to i,b+r}$ with $r = 0$ if $t =$ `void` and $r = 1$ otherwise, as $(\mathit{extend}_M^{i,j})^{\mathbb{T}}(\phi) = \neg\check{e} \wedge \exists_{\{\bar{l}_0,\ldots,\bar{l}_{i'}\}}\big(\phi[\hat{s}_b/\hat{s}_0][\bar{l}_k/\hat{l}_k \mid 0 \leq k < i'][\check{s}_{k+b}/\check{l}_k \mid 0 \leq k \leq n] \wedge \bigwedge_{0 \leq k < i} A_{b,M}(l_k) \wedge \big(\neg\hat{e} \to \bigwedge_{0 \leq k < b} A_{b,M}(s_k)\big)\big)$, with $SA_{b,M,v} = \{l_k \mid 0 \leq k \leq n,\ \neg\overline{share}(v, s_{b+k})$ or $\neg\overline{updated}_M(l_k)\}$, $A_{b,M}(v) = \check{v} \leftrightarrow \hat{v}$ if $SA_{b,M,v} = \emptyset$ and $A_{b,M}(v) = ((\check{v} \vee (\bigvee_{w \in SA_{b,M,v}} \overline{w})) \leftarrow \hat{v})$ otherwise.*

Proposition 3. *The operators $;^{\mathbb{T}}$, $\mathit{extend}^{\mathbb{T}}$ and $\cup^{\mathbb{T}}$ are correct.*

Since the number of Boolean formulas over a given finite set of variables is finite (modulo equivalence), the abstract fixpoint is reached in a finite number of iterations. Hence this abstract semantics is a static analysis tool if one specifies the sources of tainted information and the sinks where it should not flow.

Sources. Some formal parameters or return values must be considered as sources of tainted data, that can be freely provided by the external world. Our implementation uses a database of library methods for that, such as the `request` argument of `doGet` and `doPost` methods of servlets and the return value of console and database methods. Moreover, it lets users specify their own sources through annotations. The abstract denotation in Fig. 2 is modified at *receiver_is* (a special bytecode at the beginning of each method) and *return* to force to true those formal arguments and return values that are injected tainted data, respectively.

Sinks. Our implementation has a database of library methods that need untainted parameters (users can add their own through annotations). Hence it knows which `call`s in P need an untainted parameter v (such as `executeQuery` in Fig. 1). But a denotational semantics is an input/output description of the behavior of P's methods and does not say what is passed *at* a `call`. For that, a *magic-sets transformation* [23] of P adds new blocks of code whose denotation gives information at internal program points, as traditional in denotational static analysis. It computes a formula ψ that holds at the `call`. If ψ entails $\neg\hat{v}$ then the `call` receives untainted data for v. Otherwise, the analysis issues a warning.

5.1 Making the Analysis Field-Sensitive

The approximation of getfield f in Fig. 2 specifies that if the value of field f (pushed on the stack) is tainted then the container of f must be tainted as well ($\hat{s}_{j-1} \rightarrow \check{s}_{j-1}$). Read the other way round, if the container is untainted then f's value is untainted, otherwise it is conservatively assumed as tainted. This choice is sound and object-sensitive, but field-insensitive: when $\check{s}_{j-1}$ is tainted, both its fields f and g are conservatively assumed as tainted. But if the program never assigns tainted data to f, then f's value can only be untainted, regardless of the taintedness of $\check{s}_{j-1}$. If the analyzer could spot such situations, the resulting analysis would be field-sensitive and hence more precise (fewer false positives).

We apply here a technique pioneered in [32]: it uses a set of fields O (the *oracle*) that might contain tainted data. For getfield f, it uses a better approximation than in Fig. 2: it assumes that f's value is tainted if its container is tainted *and* $f \in O$. The problem is now the computation of O. As in [32], this is done iteratively. The analyzer starts with $O = \emptyset$ and runs the analysis in Sec. 5, but with the new abstraction for getfield f seen in this paragraph. Then it adds to O those fields g such that there is at least one putfield g that stores tainted data. The analysis is repeated with this larger O. At its end, O is further enlarged with other fields g such that there is at least one putfield g that stores tainted data. The process is iterated until no more fields are added to O. As proved in [32], this process converges to a sound overapproximation of O and the last analysis of the iteration is sound. In practice, repeated analyses with larger and larger O are made efficient by caching abstract computations. On average, this process converges in around 5 iterations, also for large programs. By using caching, this only doubles the time of the analysis. Since preliminary analyses are more expensive than information flow analysis, this technique increases the total time by around 25% on average. (Sec. 6 shows effects on cost and precision.) This technique is not identical to statically, manually classifying fields as tainted and untainted, as [3, 10] do. The classification of the fields is here dynamic, depending on the program under analysis, and completely automatic. Moreover, a field might be in O (and hence be potentially tainted) but the analyzer might still consider its value untainted, because its container is untainted.

6 Experiments

We have implemented our analysis inside Julia (`http://www.juliasoft.com/julia`). Julia represents Boolean formulas via BDDs (binary decision diagrams).

Test	Tool	True Positives	False Positives	False Negatives	Analysis Time
CWE89	CodePro Analytix	1332	0	888	20 minutes
	FindBugs	1776	2400	444	2 minutes
	Fortify SCA	700	0	1520	2.5 days
	Julia fs/fi	2220/2220	0/0	0/0	79/65 minutes
WebGoat	CodePro Analytix	26	7	1	1 minute
	FindBugs	22	12	5	20 seconds
	Fortify SCA	23	0	4	164 minutes
	Julia fs/fi	27/27	14/15	0/0	3/2 minutes

Fig. 3. Experiments with the identification of SQL injections.

We have compared Julia with other tools that identify injections (Sec. 3). For Julia we have compared a field-sensitive analysis with an oracle (Sec. 5.1, *Julia fs*) with a field-insensitive analysis without oracle (*Julia fi*).

Our experiments analyze third-party tests developed to assess the power of a static analyzer to identify injection attacks: WebGoat 6.0.1 (`https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project`) and 4 tests from the Samate suite (`http://samate.nist.gov/SARD/testsuite.php`). The table on the right reports their number of non-blank, non-comment lines of application source code (LoC), without supporting libraries.

Test	LoC
WebGoat	25070
CWE80	68967
CWE81	34317
CWE83	34317
CWE89	748962

Fig. 3 reports the evaluation for SQL injections using CWE89 and WebGoat. It shows that only Julia is sound (no false negatives: if there is an injection, Julia finds it). Julia issued no false positives to CWE89: possibly these tests just propagate information, without side-effects that degrade the precision of Julia (Def. 9; we do not know if and how other tools deal with side-effects). Julia issued 14 false alarms for WebGoat, often where actual information flows from source to sink exist, but constrained in such a way to be unusable to build an SQL-injection attack. Only here the field-insensitive version of Julia is slightly less precise (one false positive more). In general, its cost is around 25% higher than the field-sensitive version. The conclusion is that field sensitivity is not relevant when object sensitivity is used to distinguish different objects. Analysis time indicates the efficiency, roughly: CodePro Analytix and FindBugs work on the client machine in Eclipse, Fortify SCA on its cloud like Julia, that is controlled from an Eclipse client. Times include all supporting analyses.

We evaluated the same tools for the identification of cross-site scripting injections in CWE80/81/83, and WebGoat. As shown in Fig. 4, Julia is perfectly precise. It missed 11 cross-site scripting attacks in JSP (not in the main Java code of the application), found only by Fortify SCA. If we translate JSP's into Java through Jasper (as a servlet container would do, automatically) and include its bytecode in the analysis, Julia finds the missing 11 attacks. Nevertheless, this process is currently manual and we think fairer to count 11 false negatives.

We have run Julia on real code from our customers. Julia found 6 real SQL-injections in the Internet banking services (575995 LoC) of a large Italian bank,

Test	Tool	True Positives	False Positives	False Negatives	Analysis Time
CWE80	CodePro Analytix	180	0	486	9 minutes
	FindBugs	19	0	647	18 seconds
	Fortify SCA	282	0	384	590 minutes
	Julia fs/fi	666/666	0/0	0/0	5/4 minutes
CWE81	CodePro Analytix	0	0	333	10 seconds
	FindBugs	19	0	314	4 seconds
	Fortify SCA	141	0	192	303 minutes
	Julia fs/fi	333/333	0/0	0/0	3/2 minutes
CWE83	CodePro Analytix	90	0	243	5 minutes
	FindBugs	19	0	314	4 seconds
	Fortify SCA	141	0	192	296 minutes
	Julia fs/fi	333/333	0/0	0/0	3/2 minutes
WebGoat	CodePro Analytix	5	0	11	1 minute
	FindBugs	0	0	16	20 seconds
	Fortify SCA	15	21	1	164 minutes
	Julia fs/fi	5/5	0/0	11/11	3/2 minutes

Fig. 4. Experiments with the identification of XSS injections.

and found 5 more in its customer relation management system (346170 LoC). The analysis never took more than one hour. This shows that Julia is already able to scale to real software and automatically find evidence of security attacks.

7 Conclusion

We have formalized an object-sensitive notion of taintedness that can be applied to reference types. We have built a new, flow-, context- and field-sensitive static taintedness analysis based on this notion, proved it sound, implemented it, and evaluated it. It scales to real code and gives useful results. As far as we know, this is the first object-sensitive taintedness analysis. As usual in static analysis, soundness is jeopardized by the use of reflection or non-standard class loaders. However, soundness is still relevant since it increases the confidence on the results, up to those features. Julia deals instead with the full bytecode generated by Java 8, including the new `invokedynamic`.

The novelty of the approach stems from Def. 4 of a property of reference types as a reachability property, whose relevance goes beyond the case of taintedness analysis. Here, we mean reachability of data from a memory reference, which is not reachability of abstract states through execution paths as in [24]. Def. 4 results in an object-sensitive analysis: the taintedness of an object determines that of its fields; a drawback is that a sound analysis must consider side-effects at `putfield` and `call`. The analysis becomes then field sensitive through an oracle-based approach (Sec. 5.1), already used for nullness analysis [32]. Hence the oracle is a general technique for building sound field-sensitive static analyses.

The extension of this work to implicit and hidden flows would provide a stronger guarantee against injections of tainted information into a set of sinks.

The problem is complex: implicit flows in Java are not just due to conditionals but also to exception branches and dynamic resolution of method calls. The risk is that a sound analysis *w.r.t.* implicit flows would end up being very conservative and imprecise. Declassification might be helpful here, but its meaning for reference types (not just primitive values) must be studied. The extension of this work to the analysis of JSP, that are non-Java code mixed and interacting with Java code, currently not analyzed by Julia (only partially by concurrent tools), would avoid missed alarms, as Sec. 6 shows. It is also important to explain the warnings to the users, with an execution trace where data flows from sources into sinks. Fortify SCA already provides some support in that direction.

Acknowledgments This material is based upon work supported by the United States Air Force under Contract No. FA8750-12-C-0174.

References

1. D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan. Automated Testing for SQL Injection Vulnerabilities: An Input Mutation Approach. In *ISSTA*, pages 259–269, San Jose, CA, USA, 2014.
2. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*, page 29, Edinburgh, UK, June 2014.
3. G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight non-Interference Java Bytecode Verifier. *Mathematical Structures in Computer Science*, 23(5):1032–1081, 2013.
4. G. Barthe, T. Rezk, and A. Basu. Security Types Preserving Compilation. *Computer Languages, Systems & Structures*, 33(2):35–59, 2007.
5. D. Clark, C. Hankin, and S. Hunt. Information Flow for ALGOL-like Languages. *Computer Languages*, 28(1):3–28, April 2002.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.
7. J. C. Doshi, M. Christian, and B. H. Trivedi. SQL FILTER - SQL Injection Prevention and Logging using Dynamic Network Filter. In *SSCC*, pages 400–406, Delhi, India, 2014.
8. Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Ciprian Spiridon, and Fausto Spoto. Boolean Formulas for the Static Identification of Injection Attacks in Java. Technical Report UW-CSE-15-09-03, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, September 2015.
9. S. Genaim, R. Giacobazzi, and I. Mastroeni. Modeling Secure Information Flow with Boolean Functions. In Peter Ryan, editor, *WITS'04*, April 2004.
10. S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In R. Cousot, editor, *VMCAI*, pages 346–362, Paris, France, 2005. Springer-Verlag.
11. S. Genaim and F. Spoto. Constancy Analysis. In M. Huisman, editor, *FTfJP*, Paphos, Cyprus, July 2008. Radboud University.
12. Y.-S. Jang and J.-Y. Choi. Detecting SQL Injection Attacks using Query Result Size. *Computers & Security*, 44:104–118, 2014.

13. N. Kobayashi and K. Shirane. Type-based Information Flow Analysis for Low-Level Languages. In *APLAS*, 2002.
14. D. G. Kumar and M. Chatterjee. MAC based Solution for SQL Injection. *Journal of Computer Virology and Hacking Techniques*, 11(1):1–7, 2015.
15. P. Laud. Semantics and Program Analysis of Computationally Secure Information Flow. In *ESOP*, pages 77–91. Springer-Verlag, 2001.
16. T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.
17. L. Liu, J. Xu, M. Li, and J. Yang. A Dynamic SQL Injection Vulnerability Test Case Generation Model Based on the Multiple Phases Detection Approach. In *COMPSAC*, pages 256–261, Kyoto, Japan, 2013.
18. A. Makiou, Y. Begriche, and A. Serhrouchni. Improving Web Application Firewalls to Detect Advanced SQL Injection Attacks. In *IAS*, pages 35–40, Okinawa, Japan, 2014.
19. MITRE/SANS. Top 25 Most Dangerous Software Errors. `http://cwe.mitre.org/top25`, September 2011.
20. M. Mizuno. A Least Fixed Point Approach to Inter-Procedural Information Flow Control. In *NCSC*, pages 558–570, 1989.
21. N. M. Naghmeh Moradpoor Sheykhkanloo. Employing Neural Networks for the Detection of SQL Injection Attack. In *SIN*, page 318, Glasgow, Scotland, UK, 2014.
22. Đ. Nikolić and F. Spoto. Reachability Analysis of Program Variables. *ACM Transactions on Programming Languages and Systems*, 35(4):14, 2013.
23. É. Payet and F. Spoto. Magic-Sets Transformation for the Analysis of Java Bytecode. In *SAS*, pages 452–467. Springer, 2007.
24. T. W. Resp, S. Horwitz, and S. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL'95*, pages 49–61, San Francisco, California, USA, January 1995.
25. A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
26. A. Sabelfeld and D. Sands. A PER Model of Secure Information Flow in Sequential Programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.
27. S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *SAS*, pages 320–335. Springer, 2005.
28. H. Shahriar and M. Zulkernine. Information-Theoretic Detection of SQL Injection Attacks. In *HASE*, pages 40–47, Omaha, NE, USA, 2012.
29. L. K. Shar and K. Tan, H. B. Defeating SQL Injection. *IEEE Computer*, 46(3):69–77, 2013.
30. B. Simic and J. Walden. Eliminating SQL Injection and Cross Site Scripting using Aspect Oriented Programming. In *ESSoS*, pages 213–228, Paris, France, 2013.
31. C. Skalka and S. Smith. Static Enforcement of Security with Types. In *ICFP*, pages 254–267. ACM press, 2000.
32. F. Spoto. Nullness Analysis in Boolean Form. In *SEFM*, pages 21–30, Washington, DC, USA, 2008. IEEE.
33. O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. *SIGPLAN Notices*, 44(6):87–97, June 2009.
34. D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.
35. T.-Y. Wu, J.-S. Pan, C.-M. Chen, and C.-W. Lin. Towards SQL Injection Attacks Detection Mechanism using Parse Tree. In *ICGEC*, pages 371–380, Nanchang, China, 2014.