Static Identification of Injection Attacks in Java

FAUSTO SPOTO, Università di Verona, Italy and JuliaSoft Srl, Italy
ELISA BURATO, JuliaSoft Srl, Italy
MICHAEL D. ERNST, University of Washington, USA
PIETRO FERRARA, JuliaSoft Srl, Italy
ALBERTO LOVATO, Università di Verona, Italy
DAMIANO MACEDONIO, JuliaSoft Srl, Italy
CIPRIAN SPIRIDON, JuliaSoft Srl, Italy

The most dangerous security-related software errors, according to the OWASP Top Ten 2017 list, affect web applications. They are potential injection attacks, which exploit user-provided data in order to execute undesired operations: database access and updates (*SQL injection*); generation of malicious web pages (*cross-site scripting injection*); redirection to user-specified web pages (*redirect injection*); execution of OS commands and arbitrary scripts (*command injection*); loading of user-specified, possibly heavy or dangerous classes at run time (*reflection injection*); access to arbitrary files on the file system (*path-traversal*); storing user-provided data into heap regions normally assumed to be shielded from the outside world (*trust boundary violation*). All these attacks exploit the same weakness: unconstrained propagation of data from *sources* that the user of a web application controls into *sinks* whose activation might trigger dangerous operations. Although web applications are written in a variety of languages, Java remains a frequent choice, in particular for banking applications, where security has tangible relevance.

This article defines a unified, sound protection mechanism against such attacks, based on the identification of all possible explicit flows of *tainted* data in Java code. Such flows can be arbitrarily complex, passing through dynamically allocated data structures in the heap. The analysis is based on abstract interpretation and is interprocedural, flow-sensitive, and context-sensitive. Its notion of taint applies 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 data flows. Its implementation, within the Julia analyzer for Java and Android, found injection security vulnerabilities in the Internet banking service and in the customer relationship management of large Italian banks, as well as in a set of open-source third-party applications. It found the command injection which is at the origin of the 2017 Equifax data breach, one of the worst data breaches ever. For objective, repeatable results, this article also evaluates the implementation on two open-source security benchmarks: the Juliet Suite and the OWASP Benchmark for the automatic comparison of static analyzers for cybersecurity. We compared this technique against more than ten other static analyzers, both free and commercial. The result of these experiments is that ours is the only analysis for injection that is sound (up to well-stated limitations such as multithreading and native code) and works on industrial code, and it is also much more precise than other tools.

Authors' addresses: Fausto Spoto, Dipartimento di Informatica, Università di Verona, Verona, Italy, JuliaSoft Srl, Verona, Italy, fausto.spoto@univr.it; Elisa Burato, JuliaSoft Srl, Verona, Italy, elisa.burato@juliasoft.com; Michael D. Ernst, University of Washington, Seattle, USA, mernst@cs.washington.edu; Pietro Ferrara, JuliaSoft Srl, Verona, Italy, pietro.ferrara@juliasoft.com; Alberto Lovato, Dipartimento di Informatica, Università di Verona, Verona, Italy, alberto.lovato@univr.it; Damiano Macedonio, JuliaSoft Srl, Verona, Italy; Ciprian Spiridon, JuliaSoft Srl, Verona, Italy.

Manuscript submitted to ACM

CCS Concepts: • **Security and privacy → Logic and verification**; **Web application security**; *Software security engineering*; • **Theory of computation → Program analysis**; *Denotational semantics.*

Additional Key Words and Phrases: Static Analysis, Web Application Security, SQL Injection Attack, XSS, Taint Analysis, Abstract Interpretation

ACM Reference Format:
Fausto Spoto, Elisa Burato, Michael D. Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, and Ciprian Spiridon. 2019. Static Identification of Injection Attacks in Java. *ACM Trans. Program. Lang. Syst.* 1, 1 (November 2019), 59 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACKNOWLEDGMENTS

This work has been partially supported by the United States Air Force under Contract No.: FA8750-12-C-0174.

1 INTRODUCTION

Many programs, such as web applications, web services, and network applications, react to user input or data from the network The user or the environment controls the input and can pass any value for it. The execution of the program depends on the input, but the program should use the input in a controlled and validated way. Otherwise, an attacker can *inject* special data that induces unsafe, unexpected behaviors of the program. Injection attacks are considered the most dangerous software error [34]. They can cause free database access and corruption, forging of web pages, run-time loading of dangerous or large classes, logging of sensitive data, denial-of-service, free access to the file system, and arbitrary execution of commands or operating system scripts.

Many software analyzers implement techniques for finding injection attacks before the program gets executed (static analyzers) or when an attack is in progress (dynamic analyzers). Sec. 3 overviews these tools. Static analyzers can in principle guarantee that an application or service does not suffer from potential injection attacks, for every possible execution. If current tools fail to do so, that is because static analysis has not yet been applied at its full power to the identification of injection attacks.

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. The user of the analysis can then gauge the risk of the flow. As always with static analysis, approximations might lead to false alarms, and proper input validation might make actual flows harmless. Managing the number of false alarms is crucial for success in an industrial context. This article reports extensive experimentation to assess precision. Our soundness statement has limitations, in particular related to the fact that native methods, reflection and multithreading can jeopardize soundness. This will be discussed in Sec. 11.

Our techniques are general. Our implementation works on Java bytecode. Handling bytecodes is essential for analyzing programs that use libraries whose source code is not available, which is usually the case in industrial contexts. The choice of Java bytecode also simplifies the semantics and its abstraction (many high-level constructs need not be explicitly considered). For clarity, we show our examples as Java source code.

The contributions of this article are the following:

- the formalization of an object-sensitive taint analysis for reference types, which is based on reachability of tainted information in memory;

- a sound 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 with primitive values);
- the proof of correctness of the static analysis, by using abstract interpretation;
- its implementation inside the Julia analyzer, through binary decision diagrams;
- extensive experimentation on large open-source third-party programs and large, standard, third-party benchmarks for the identification of injection attacks, including comparison against other static analyzers.

We performed two types of experiments. First, we showed that the analysis scales to real-world industrial Java software: both open-source applications (Sec. 7) and closed-source code (Sec. 8) from customers of the JuliaSoft Srl company. Second, to compare to other tools, we ran the analysis on two standard benchmarks for injection detection in Java code: the NIST Juliet Suite [37] (Sec. 9) and the OWASP Benchmark [41] (Sec. 10). Julia, which implements the technique described in this article, scores best among all static analyzers, both free and commercial. In particular, it is the only sound tool available at the moment and is more precise than competing tools.

As is common in the literature, the taint analysis in this article is limited to explicit flows [49]; namely, it does not consider implicit flows (arising from conditional tests) nor hidden flows (such as timing channels). Implicit flows could be handled by adding a Boolean variable at each branch whose outcome depends on tainted data [18]. That variable would be used in the scope of the branch: when it holds true, the analysis marks as tainted all variables updated in the scope of the branch. However, in an object-oriented language there are many such branches, including those that arise because of dynamic method dispatch or exception handling. The result would be to mark as tainted almost every variable, thus degrading the precision of the analysis. Such a static analyzer would issue warnings almost everywhere, hence sound *w.r.t.* implicit flows but never accepted by its users. Probably for this reason, current benchmarks and test cases for injection attacks consider explicit flows only. Julia suffers no false negatives on the Juliet and OWASP tests.

This article is organized as follows. Sec. 2 gives an example of injection and clarifies the need for 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 a 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 describes the implementation of the analysis inside the Julia static analyzer. Sec. 7 reports experiments with the analysis of a set of open-source third-party applications. Sec. 8 reports examples of vulnerabilities found on actual code from customers of JuliaSoft Srl. Sec. 9 presents experiments over the Juliet Suite. Sec. 10 presents experiments over the OWASP Benchmark and compares against other tools. Sec. 11 enumerates the limitations of the analysis. Sec. 12 concludes.

A preliminary report about this static analysis appeared in Ernst et al. [14]. Some experiments with the OWASP Benchmark appeared at a national workshop [10]. This article gives a uniform and more detailed presentation of the material, includes proofs of correctness of the analysis, reports examples of flaws from open-source applications and from real code of customers of JuliaSoft Srl, and includes more detailed discussion about the experiments with the Juliet Suite and the OWASP Benchmark.

2 EXAMPLES

Sec. 2.1 shows examples of Java code that contains security vulnerabilities related to injection of data. Sec. 2.2 introduces, informally, a reachability-based notion of taintedness.

```
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 injection and XSS attacks. For brevity, this figure omits exception-handling code.

2.1 Examples of Injections

Fig. 1 shows a Java servlet that suffers from potential SQL-injection and cross-site scripting (XSS) attacks. A *servlet* (lines 1 and 2) listens for HTTP network connection requests, retrieves its parameters, and runs some code in response to each request. The response (formal parameter `response` on 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. Programmers can write Java servlets directly, or generate them at compile time (say, by using Java Enterprise Edition [21] or Spring [23]), or generate them dynamically at run time (say, by using Java Server Pages, *i.e.*, JSP [22]). In any case, the user of a servlet provides the parameters through the URL, as in `http://my.site.com/myServlet?user=john`. Servlet code retrieves these (`getParameter` method, line 5). Lines 9 and 10 establish a connection to a database, which is assumed to define a table `User` (line 27) of all users of the service. Line 14 builds an SQL query from the user name provided as parameter. This query is reported to the response (line 15) and also executed (line 17). The result is a relational table of all users matching the given criterion. This table is then printed to the response (lines 18–20).

This code is subject to an SQL injection attack, because the client is free to specify the value of the `user` parameter. In particular, the client can provide a string that makes line 17 run *any* possible database command, including malicious commands that erase the database, access unexpected data, or insert new rows. For instance, if the user supplies the string "`'; DROP TABLE User --`" as `user`, the resulting concatenation is the "`SELECT * FROM User WHERE userId='; DROP TABLE User --'`" SQL command, which performs a selection (likely yielding no result) and then erases the `User` table from the database. The attacker here is exploiting the SQL comment operator `--` in order to neutralize the final `'` character. As another example, the user might supply the string "`' OR '1'='1' --`" as `user`, inducing the application to run the SQL command "`SELECT * FROM User WHERE userId=' OR '1'='1' --'`", which would select all users registered to the system, not just the single user that was meant to be specified by variable `user`. In the literature, this is known as an SQL-injection attack and follows from the fact that *tainted* user input flows from the `request` *source* into the `executeQuery` *sink* method. By contrast, there is no SQL-injection at line 22, although it looks very much like line 17, since the query at line 22 is not computed from user-provided input.

A different risk exists at lines 15 and 20. Data printed to the response object is typically interpreted by the client browser as HTML content. A malicious user can provide a `user` parameter that contains arbitrary HTML tags, including tags that will make the client execute malicious scripts. For instance, the malicious user might post the following URL to a public forum or send it by email: "`http://my.site.com/myServlet?user=<script>malicious-javascript</script>`". The reader of the forum or the receiver of the email might trust the link based on its host (here `my.site.com`). If he clicks on the link, the parameter `user` holds "`<script>malicious-javascript</script>`". At line 15 this code would be sent to the user's browser, which runs `malicious-javascript` as a JavaScript command. This script might open a pop-up, redirecting a naive user into a dangerous site, or might ask for credentials and send them to the attacker. Moreover, that script runs with all permissions already granted to the original web page, which increases the risk of harm. The user trusts the web site, but the user is seeing content that has been injected into the original web page. In the literature, this is known as a cross-site scripting attack (XSS) 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, however, the flow of information is more complex and indirect: in other parts of the application, the user might have already saved her address to the database and used malicious JavaScript code instead of the actual address; line 20 will then fetch this malicious code and send it to the browser of the client to run it.

Many other kinds of injections exist. They all arise from an information flow from a source the user can specify (the parameter of the request, input from console, text widgets, data in a database) to a sensitive sink, such as `executeQuery` (SQL-injection), `print` (XSS), reflection methods (reflection-injection via loading any class, inspecting any object, or executing any method), `execute` (command-injection via running any operating system command), etc. This article focuses on the identification of flows of tainted information, not on the exact enumeration of sources and sinks, which is long and gets longer with each new framework for web application development that hits the market. The Julia analyzer is instantiated with a list of sources and sinks from the literature (see https://static.juliasoft.com/docs/latest/injection_sources_sinks.html for the complete list).

A flow of information from source to sink must exist in order to build an injection attack. However, the converse does not hold, in general. Namely, the flow might be constrained in such a way that no harmful attack can be triggered. Or information might be *sanitized* before reaching the sink. For instance, many web programming frameworks provide sanitizing methods against XSS attacks, that escape all characters that could be used to run a JavaScript command, making them useless for an attack. Nevertheless, the number of web applications that do not use such sanitizing

methods is surprisingly high, even in critical contexts. Sec. 5.5 shows how the analysis has been instructed about such sanitizing methods, in order to reduce the number of false alarms.

2.2 Reachability-based Taintedness

This section introduces, informally, the abstract domain for taint analysis that will be formalized in Sec. 5. The goal is to show how it deals with fields of objects, to provide a reachability-based notion of taintedness.

Fig. 2 shows a servlet that collects names, received as input, into a doubly-linked list of `Nodes`. Each node holds a name and a string timestamp. Names are injected as request parameters and are consequently tainted. Timestamps are created by the logic of the servlet and are untainted. The return value of `getNodes` is tainted, since it is possible to reach the tainted `names` from it. The return value of `getTimestamps` is untainted, since it just holds the timestamps, not the names. It is interesting to see if a static analysis can reach this conclusion.

The return value of `getNodes` is just a pointer in memory, bound to `result` at line 26. As such, it is, by itself, just an untainted memory address; however, from that pointer it is possible to *reach* tainted data held in the field `name` of the nodes in `result`, by following the access path `result.elementData[n].name`, where `elementData` is the array holding the elements of a Java `ArrayList` and n is the arbitrary index of an element of that array.

Our abstraction approximates a program state by using Boolean formulas, whose Boolean variables are the program variables in that state. Hence, for instance, variable `result` at line 29 is approximated by the Boolean formula "$\hat{result}$ = true" (or the equivalent formula $\hat{result}$), meaning that the final value of variable `result` might reach tainted data there. By contrast, variable `result` at line 37 is approximated by the Boolean formula "$\hat{result}$ = false", meaning that the final value of variable `result` does not reach tainted data there, definitely. Therefore, our abstract domain completely abstracts fields away. Soundness requires that the value of a field $o.f$ of an object o can be inferred as untainted only if o has been inferred as untainted.

The analysis described so far suffers imprecision: the approximation for `cursor` at lines 28 and 36 is $\hat{cursor}$, since tainted data can be reached from `cursor`. This field-agnostic approximation conservatively and imprecisely entails that `cursor.timestamp` might be tainted.

This imprecision is overcome in Sec. 5.6. There, a field-sensitive domain is defined, by enriching the approximation with a possibly-tainted fields set TF, which is a sound overapproximation to the set of fully-qualified field names that might reach tainted data. A field that is not in TF cannot reach tainted data, definitely. In Fig. 2, the analysis in Sec. 5.6, using the set TF = {`AnotherSelvlet.start`, `Node.next`, `Node.previous`, `Node.name`, `java.lang.ArrayList.elementData`}, concludes that `cursor.timestamp` is untainted, since the field `Node.timestamp` is not in TF. The analysis infers that untainted data is stored at line 36 into the list `result` and the return value of `getTimestamps`, at line 37, is untainted. Sec. 5.6 shows how to automatically compute TF, through an iteration of the analysis. This iteration is finite since TF can only grow during the iteration and there is a finite number of field names in the program. Soundness, for the analysis in Sec. 5.6, requires that the value of a field $o.f$ of an object o can be inferred as untainted only if either o is untainted; or f is not in TF, regardless of the taintedness of o. The first possibility means that the analysis is object-sensitive; the second, that it is field-sensitive. Is this second possibility that lets the analyzer infer `cursor.timestamp` as untainted at line 36, although `cursor` is tainted there.

3 RELATED WORK

The identification of possible injections and the inference of information flows are well-studied topics. Nevertheless, no previous sound technique works on real Java code, not even just for explicit flows. Most injection identification

```
1  public class AnotherServlet extends HttpServlet {
2    private Node start;
3
4    public static class Node {
5      private Node next, previous;
6      private final String name;      // tainted data
7      private final String timestamp; // untainted data
8      private Node(String name) { this.name = name; this.timestamp = new Date().toString(); }
9      @Override public String toString() { return name; }
10   }
11
12   @Override
13   public void doGet(HttpServletRequest request, HttpServletResponse response) {
14     Node node = new Node(request.getParameter("name"));
15     if (start == null)
16       start = node;
17     else {
18       Node cursor = start;
19       while (cursor.next != null) cursor = cursor.next;
20       cursor.next = node; node.previous = cursor;
21     }
22   }
23
24   // This method returns a tainted list (when non-empty).
25   public List<Node> getNodes() {
26     List<Node> result = new ArrayList<>();
27     for (Node cursor = start; cursor != null; cursor = cursor.next)
28       result.add(cursor);
29     return result;
30   }
31
32   // This method returns an untainted list.
33   public List<String> getTimestamps() {
34     List<String> result = new ArrayList<>();
35     for (Node cursor = start; cursor != null; cursor = cursor.next)
36       result.add(cursor.timestamp);
37     return result;
38   }
39
40   public static int countPrevious(Node node) { // assumes node non-null
41     int counter = 0;
42     while (node.previous != null) { node = node.previous; counter++; }
43     return counter;
44   }
45 }
```

Fig. 2. A Java servlet that collects names into a doubly-linked list of nodes.

techniques are dynamic and/or unsound. Existing static information-flow analyses are not satisfactory for languages with reference types.

3.1 Identification of Injections

Data injections are security risks, so there is high industrial and academic interest in their automatic identification. Here, we mention only the most recent works regarding SQL-injection.

Almost all techniques aim at the dynamic identification of the injection when it occurs [13, 24, 25, 27, 33, 36, 46, 53, 55, 68], or at the generation of test cases of attacks [2, 31, 32] or at the specification of good coding practices [54] or at the classification of SQL injection attacks through explicit flows or implicit flows, such as timing channels [59].

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 the analysis issues a warning. A static analysis is *sound* if it finds all places where an injection might occur at run time (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). There is theoretical work that presents static analyses for detecting injection attacks. Namely, Xiao et al. [69] determines where injections are likely to occur, on the basis of the strings used in the program. The analysis is statistical and unsound. A sound static analysis is presented in Fu et al. [16]. For each program point where an injection could occur, it computes and solves a string constraint, whose solution describes the potential attack. Also Wassermann and Su [66] define a sound static analysis, which describes the strings used in query methods by using an automaton. Attacks are then found by querying the automaton language. The complexity of modern programming languages limits the scope of these works, which have remained at prototype level. Consequently, it is impossible to judge their applicability to real, large programs or benchmarks, both in terms of scalability and precision. Moreover, both works are limited to programs manipulating strings (which are typically immutable objects), while data in real programs can flow through any other data structure, also through mutable data.

Beyond our implementation inside the Julia analyzer (https://www.juliasoft.com), three static analyzers that identify injections in Java and that one can use and compare 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. 5.1. In 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; and CodePro Analytix warns at lines 15, 17, 20, and also, imprecisely, at the harmless line 22. Sec. 10 compares these and other tools with Julia in more detail. TAJ [62] is part of IBM AppScan. This tool compromises the soundness of the analysis in order to achieve scalability (see Sec. 6.1 and 6.2 of Tripp et al. [62]). Moreover, FlowDroid [3] applies taint analysis to Android applications. Both IBM AppScan and FlowDroid use a theoretical framework of *access paths*, that is discussed in Sec. 3.2, since it expresses a notion of reachability-based taintedness that is related (but distinct) from that formalized in this article. Here, we only add that IBM AppScan is one of the commercial tools considered in the OWASP benchmarks; while the results of these tools are anonymized, our analysis outperforms all of them as shown and discussed in Sec. 10. These experiments have been performed by OWASP; we are just reporting their numbers. More recently, machine learning has been applied to static analysis, also for the detection of injection vulnerabilities. An example is the LGTM tool described in Hélie et al. [20], that runs queries expressed in the QL language [4].

3.2 Reachability Computed via Access Paths

The backwards taint analysis in Tripp et al. [61] starts from each sink, where tainted data must not flow, and infers data access paths that hold such data. If such paths lead back to a source of tainted data, a warning is issued. A proof of soundness is sketched in [61]. Since access paths can be infinite, a widening is used to enforce termination of the analysis. The advantage of the technique is low cost, if there are few sinks in the program. Moreover, analyses can be recomputed when the programs under analysis are modified, which is a major motivation for its adoption in an industrial context. This section compares that reachability-based taint analysis to our analysis. It focuses on the expressive power of the two abstract domains, not on the implementations, which are completely different (ours is a whole-program analysis, whereas [61] defines a backwards demand-driven analysis).

Namely, [61] defines a static analysis where each variable is abstracted into an overapproximation of the set of tainted access paths rooted at that variable. Hence, in [61], taintedness is not a property of a value by itself, but rather a property of the memory reachable from that value, by following some access path. Soundness, for [61], requires that the value of a field $o.f$ of an object o can be inferred as untainted only if the approximation for o has no (tainted) access path that starts with $o.f$. (Contrast this with our notion of soundness in Sec. 2.2.)

Our approach follows the same idea of reachability-based taintedness, but uses a completely different abstraction than access paths. Namely, it abstracts concrete states into Boolean formulas, whose variables v can hold true if the corresponding program variable `v` can reach tainted data (see Sec. 2.2, or Sec. 5.1 for further details).

The approximation from [61] is incomparable to ours. [61] is stronger in that the sets of access paths used in [61] provide a finer definition of the access paths that must be followed to reach the tainted data. Ours is stronger in that the domain in Sec. 5.1 can express taintedness dependencies, such as $v \leftrightarrow w$, that arise from an assignment `v=w` or a conditional `if (v==w)`.

Moreover, the abstraction in [61] is potentially infinite: there are infinitely many access paths rooted at a given variable. For instance, the same tainted field `name` can be reached through arbitrarily long and distinct access paths:

```
result.elementData[n].name
result.elementData[n].previous.next.name
result.elementData[n].next.previous.name
result.elementData[n].previous.next.previous.next.name
result.elementData[n].next.previous.next.previous.name
...
```

[61] keeps the static analysis finite by fixing a maximal depth. A $*$ notation stands for an arbitrary sequence of fields: `result.elementData[n]*`. The same idea could be used for arrays, abstracting `result` at line 29 of Fig 1 into `result.elementData[*]*`: all its elements might reach tainted data. Soundness, for [61] with the introduction of $*$, requires that the value of a field $o.f$ of an object o can be inferred as untainted only if the approximation for o has no access path that starts with $o.f$ nor with `o*`. This finiteness problem does not exist in our analysis, where fields have been completely abstracted away: at each given program point, there is only a finite number of program variables in scope, hence the number of Boolean formulas built from those variables is finite. Consequently, our abstract domain does not use any form of widening. The drawback of our technique is that imprecision is shifted at the level of the base object, whose approximation does not distinguish precisely between the taintedness of specific access paths.

Using the abstraction in [61], variable `cursor` at lines 28 and 36 is approximated as `cursor*`, because of the merge-over-all-paths that occurs inside the loops. The $*$ operator stands for all fields. Hence, from that approximation the

analysis can only assume that `cursor.timestamp` is tainted, which is not actually the case. It follows that the analysis in [61] concludes that the return value of both `getNodes` and `getTimestamps` in Fig. 2 are tainted, conservatively. This result is sound but imprecise (for `getTimestamps`). By contrast, Sec. 2.2 shows how our analysis computes a precise result for `getTimestamps`, using the possibly-tainted field set *TF*.

The same information expressed by *TF* could be expressed as the set of all access paths, of arbitrary length, that can *only* pass through fields in *TF*. However, that set is infinite, in general, and cannot be expressed precisely with the introduction of $*$.

3.3 Modeling of Information Flow

Many static analyses model explicit and often also implicit information flows [49] in Java-like or Java bytecode programs. There are data/control-flow analyses [11, 29, 35, 50]; type-based analyses [6, 7, 17, 26, 56, 65] and analyses based on abstract interpretation [18]. They are satisfactory for variables of primitive type but impractical for heap-allocated data of reference type, such as strings and arbitrary structures, possibly mutable. Most analyses [7, 11, 17, 26, 29, 35, 50, 65] assume that the language has only primitive types; others [6, 18] 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. For these analyses, 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: most will 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* (*i.e.*, reachability-based) notion of taintedness, that fits for heap-allocated data of reference type. It has been defined through abstract interpretation, which has the advantage of providing its correctness proof in a formal and standard way.

3.4 Supporting Heap Analyses

Our taintedness analysis is reachability-based. Hence, field updates and method calls, that perform heap modifications, might modify the taintedness of variables. In order to model such side-effects, a reachability-based analysis needs supporting heap analyses, namely, reachability (used in Fig. 4) and sharing (used in Def. 5.16). Reachability analysis is implemented as Nikolic and Spoto [38] and sharing analysis as Secci and Spoto [52]; they are not aliasing analysis, although all are in the family of heap analyses.

3.5 Concrete Semantics

The concrete semantics that will be defined in Sec. 4 is given at the bytecode level. A possible alternative is to define a semantics for an intermediate representation of code such as Jimple [64]. This alternative would simplify the notion of state in Sec. 4, since there are only variables then, instead of both local variables and stack elements as in Java bytecode. It would recycle an infrastructure that has proved to be effective in many other static analysis tools. But our analysis has been implemented inside and for the Julia analyzer, that was built from 2003 to analyze Java bytecode, directly. After so many years of development, it is now too late to change Julia to let it analyze another intermediate representation: it would be a costly and purely engineering task, with no conceptual advantage. In particular, the use of an intermediate language like Jimple would not simplify the analysis conceptually: it would not change the set of interesting instructions that the analysis must consider and that this article must describe (variable assignment, object

creation, field access, method call and return, exception throw and catch: see Fig. 3). From the engineering side, Julia already provides the abstractions and analyses needed by our implementation (Sec. 3.4). An intermediate language like Jimple would oblige us to develop such supporting analyses from scratch.

4 DENOTATIONAL SEMANTICS OF JAVA BYTECODE

This section presents a denotational semantics for Java bytecode, which Sec. 5 will abstract into a taint analysis. The same semantics has been used for nullness analysis [57] and has been proved equivalent [44] to an operational semantics. The only difference is that, in this article, primitive values are decorated with their taintedness.

4.1 State of the JVM

As usual, in the following *bytecode* will refer both to the low-level language, resulting from the compilation of Java source code, and to each single instruction of that low-level language. We assume a Java bytecode program P given as a collection of graphs of *basic blocks* of code, one for each method. Bytecode instructions that might throw exceptions are linked to a handler starting with a `catch` bytecode, possibly followed by bytecode instructions routing the computation on the basis of the run-time class of the 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; we further assume that method parameters cannot be reassigned inside their body. These assumptions do not reduce the expressive power of the language but largely simplify formalization and proofs. Our implementation handles full Java bytecode, without such limitations.

Definition 4.1 (Classes). The set of *classes* $\mathbb{K}$ is partially ordered *w.r.t.* the *subclass relation* $\leq$. A *type* is an element of $\mathbb{K} \cup \{\texttt{int}\}$. A class $\kappa \in \mathbb{K}$ defines *instance fields* $\kappa.f : t$ (field named f of type t, defined in class κ) and *instance methods* $\kappa.m(t_1, \ldots, t_n) : t$ (method named m, with arguments of type $t_1, \ldots, t_n$, returning a value of type t, possibly `void`, defined in class κ). Constructors are considered as methods returning `void`. When the name is unambiguous, we simply 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 the `request` object or the string `user` in Fig. 1) will be defined later as a reachability property from the reference (Def. 5.1); by contrast, primitive tainted values are explicitly marked in the state.

Definition 4.2 (State). A *value* is an element of $\mathbb{Z} \cup \boxed{\mathbb{Z}} \cup \mathbb{L} \cup \{\texttt{null}\}$, 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 $\langle l \parallel s \parallel \mu \rangle$ 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 $\langle l \parallel s \parallel \mu \rangle$ if $v \in \mathbb{Z} \cup \boxed{\mathbb{Z}}$ and $t = \texttt{int}$, or $v = \texttt{null}$ and $t \in \mathbb{K}$, or $v \in \mathbb{L}$, $t \in \mathbb{K}$, and $\mu(v).\kappa \leq t$.

Example 4.3. Let state $\sigma = \langle [3, \texttt{null}, \boxed{4}, \ell] \parallel \boxed{3} :: \ell'' :: \ell'' \parallel \mu \rangle \in \Xi_{4,3}$, 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 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, even in the presence of nested exception handlers [30]. The semantics of a bytecode is then a *denotation* from a *pre-* to a *post*-state. Both states can be normal or exceptional.

Definition 4.4 (JVM State and Denotation). The set of *JVM states* (from now on 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 *pre*-state to an *output* or *post*-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 in the states. 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 $\delta_1(\sigma)$ is undefined or $\delta_2(\delta_1(\sigma))$ is undefined.

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

4.2 Concrete Semantics of Each Bytecode Instruction

This section reports the concrete semantics (denotation) of each single bytecode instruction. Java bytecode has more than a hundred bytecode instructions. Below, only the semantics of a few of them are formalized, which is typically representative of similar instructions. At each program point, the number i of local variables and j of stack elements and their types are statically known [30]; hence, one can assume that the semantics of the bytecodes is undefined for pre-states of wrong sizes or types. Such semantics is a function from pre- to post-state; hence, the lambda expression is used for its formalization.

Below, it is assumed that one analyzes type-checked programs, that is, Java bytecode programs where type inference succeeds [30]. For instance, it is assumed that, at a `return` t bytecode, the topmost stack element exists and has type t or a subtype of t. Consequently, that condition needn't be checked in the dynamic semantics.

The concrete semantics of a selection of Java bytecode instructions is in Fig. 3. We discuss it below in detail.

Basic instructions. Bytecode `const` v pushes $v \in \mathbb{Z} \cup \{\texttt{null}\}$ on the stack; hence, its semantics is formalized as the function $const\ v = \lambda\langle l \parallel s \parallel \mu\rangle.\langle l \parallel v :: s \parallel \mu\rangle$ (where s might be ε). The λ-notation defines a partial map, undefined on exceptional states since $\langle l \parallel s \parallel \mu\rangle$ is not underlined. That is, $const\ v$ is executed when the JVM is in a normal state. This holds for *all* bytecode instructions but `catch`, that starts the exceptional handlers from an exceptional state. Bytecode `dup` t duplicates the top of the stack, of type t: $dup\ t = \lambda\langle l \parallel top :: s \parallel \mu\rangle.\langle l \parallel top :: top :: s \parallel \mu\rangle$. Bytecode `load` $k\ t$ pushes on the stack the value of local variable number k, that must exist and have type t: $load\ k\ t = \lambda\langle l \parallel s \parallel \mu\rangle.\langle l \parallel l[k] :: s \parallel \mu\rangle$. Conversely, bytecode `store` $k\ t$ pops the top of the stack of type t and writes it in local variable k: $store\ k\ t = \lambda\langle l \parallel top :: s \parallel \mu\rangle.\langle l[k := top] \parallel s \parallel \mu\rangle$. If l has less than $k+1$ variables, the resulting set of local variables gets expanded. Binary arithmetic bytecodes such as `add` consume the operands, *i.e.*, the topmost two elements of the stack, and produce the arithmetic result: $add = \lambda\langle l \parallel v_1 :: v_2 :: s \parallel \mu\rangle.\langle l \parallel (v_2 + v_1) :: s \parallel \mu\rangle$. Here, + is the extended addition operator that yields a tainted sum if and only if at least one of its operands is tainted. The semantics of a conditional bytecode is defined as a state filter, that filters out those pre-states where its condition is false. For instance, `ifne` t checks if the top of the stack,

$$
\begin{aligned}
\mathit{const}\ v &= \lambda\langle l \parallel s \parallel \mu\rangle.\langle l \parallel v :: s \parallel \mu\rangle \\
\mathit{dup}\ t &= \lambda\langle l \parallel \mathit{top} :: s \parallel \mu\rangle.\langle l \parallel \mathit{top} :: \mathit{top} :: s \parallel \mu\rangle \\
\mathit{load}\ k\ t &= \lambda\langle l \parallel s \parallel \mu\rangle.\langle l \parallel l[k] :: s \parallel \mu\rangle \\
\mathit{store}\ k\ t &= \lambda\langle l \parallel \mathit{top} :: s \parallel \mu\rangle.\langle l[k := \mathit{top}] \parallel s \parallel \mu\rangle \\
\mathit{add} &= \lambda\langle l \parallel v_1 :: v_2 :: s \parallel \mu\rangle.\langle l \parallel (v_2 + v_1) :: s \parallel \mu\rangle \\
&\quad \text{where + yields a tainted sum if and only if any operand is tainted} \\
\mathit{ifne}\ t &= \lambda\langle l \parallel \mathit{top} :: s \parallel \mu\rangle.\begin{cases} \langle l \parallel s \parallel \mu\rangle & \text{if } \mathit{top} \notin \{0, \boxed{0}, \texttt{null}\}, \\ \mathit{undefined} & \text{otherwise} \end{cases} \\
\mathit{new}\ \kappa &= \lambda\langle l \parallel s \parallel \mu\rangle.\begin{cases} \langle l \parallel \ell :: s \parallel \mu[\ell := n]\rangle & \text{if there is enough memory} \\ \underline{\langle l \parallel \ell \parallel \mu[\ell := \mathit{oome}]\rangle} & \text{otherwise} \end{cases} \\
&\quad \text{with } \ell \in \mathbb{L} \text{ fresh and } \mathit{oome} \text{ new instance of } \texttt{java.lang.OutOfMemoryError} \\
\mathit{getfield}\ \kappa.f{:}t &= \lambda\langle l \parallel \mathit{rec} :: s \parallel \mu\rangle.\begin{cases} \langle l \parallel \mu(\mathit{rec}).f :: s \parallel \mu\rangle & \text{if } \mathit{rec} \neq \texttt{null}, \\ \underline{\langle l \parallel \ell \parallel \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{otherwise} \end{cases} \\
&\quad \text{with } \ell \in \mathbb{L} \text{ fresh and } \mathit{npe} \text{ new instance of } \texttt{java.lang.NullPointerException} \\
\mathit{putfield}\ \kappa.f{:}t &= \lambda\langle l \parallel \mathit{top} :: \mathit{rec} :: s \parallel \mu\rangle.\begin{cases} \langle l \parallel s \parallel \mu[\mu(\mathit{rec}).f := \mathit{top}]\rangle & \text{if } \mathit{rec} \neq \texttt{null}, \\ \underline{\langle l \parallel \ell \parallel \mu[\ell := \mathit{npe}]\rangle} & \text{otherwise} \end{cases} \\
&\quad \text{with } \ell \in \mathbb{L} \text{ fresh and } \mathit{npe} \text{ new instance of } \texttt{java.lang.NullPointerException} \\
\mathit{throw}\ \kappa &= \lambda\langle l \parallel \mathit{top} :: s \parallel \mu\rangle.\begin{cases} \underline{\langle l \parallel \mathit{top} \parallel \mu\rangle} & \text{if } \mathit{top} \neq \texttt{null}, \\ \underline{\langle l \parallel \ell \parallel \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{if } \mathit{top} = \texttt{null} \end{cases} \\
&\quad \text{with } \ell \in \mathbb{L} \text{ fresh and } \mathit{npe} \text{ new instance of } \texttt{java.lang.NullPointerException} \\
\mathit{catch} &= \lambda\underline{\langle l \parallel \mathit{top} \parallel \mu\rangle}.\langle l \parallel \mathit{top} \parallel \mu\rangle \\
\mathit{exception\_is}\ K &= \lambda\langle l \parallel \mathit{top} \parallel \mu\rangle.\begin{cases} \langle l \parallel \mathit{top} \parallel \mu\rangle & \text{if } \mathit{top} \in \mathbb{L},\ \mu(\mathit{top}).\kappa \leq \kappa \text{ for some } \kappa \in K, \\ \mathit{undefined} & \text{otherwise} \end{cases} \\
\mathit{return}\ t &= \lambda\langle l \parallel \mathit{top} \parallel \mu\rangle.\langle l \parallel \mathit{top} \parallel \mu\rangle \\
\mathit{return} &= \lambda\langle l \parallel \varepsilon \parallel \mu\rangle.\langle l \parallel \varepsilon \parallel \mu\rangle.
\end{aligned}
$$

Fig. 3. The concrete semantics of a selection of Java bytecode instructions.

of type t, is not 0 nor $\boxed{0}$ when $t = \texttt{int}$, and is not `null` otherwise. Its semantics *ifne t* is the filter

$$
\lambda\langle l \parallel \mathit{top} :: s \parallel \mu\rangle.\begin{cases} \langle l \parallel s \parallel \mu\rangle & \text{if } \mathit{top} \notin \{0, \boxed{0}, \texttt{null}\}, \\ \mathit{undefined} & \text{otherwise.} \end{cases}
$$

The use of filters for conditionals has been traditional in abstract interpretation since its very beginning [12]. The advantage is that all instructions become state transformers, a great simplification when their abstraction gets computed through abstract interpretation.

Memory-manipulating instructions. Some bytecodes deal with objects in memory. For instance, `new` κ pushes on the stack a reference to a new object n of class κ, with integer fields set to 0 and reference fields set to `null`. Its semantics

new κ is

$$\lambda\langle l \parallel s \parallel \mu\rangle. \begin{cases} \langle l \parallel \ell :: s \parallel \mu[\ell := n]\rangle & \text{if there is enough memory} \\ \underline{\langle l \parallel \ell \parallel \mu[\ell := \mathit{oome}]\rangle} & \text{otherwise} \end{cases}$$

with $\ell \in \mathbb{L}$ fresh and *oome* new instance of `java.lang.OutOfMemoryError`. This is the first bytecode that throws an exception. Another is bytecode `getfield` $\kappa.f{:}t$, which reads the field $\kappa.f{:}t$ of the object pointed to by the top *rec* (the *receiver*) of the stack, of type κ. Its semantics *getfield* $\kappa.f:t$ is

$$\lambda\langle l \parallel \mathit{rec} :: s \parallel \mu\rangle. \begin{cases} \langle l \parallel \mu(\mathit{rec}).f :: s \parallel \mu\rangle & \text{if } \mathit{rec} \neq \texttt{null}, \\ \underline{\langle l \parallel \ell \parallel \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{otherwise} \end{cases}$$

with $\ell \in \mathbb{L}$ fresh and *npe* new instance of `java.lang.NullPointerException`. This is the first example of a bytecode that might throw an exception while dereferencing a location (*rec*). Bytecode `putfield` $\kappa.f{:}t$ is another example. It moves the *top* of the stack, of type t, into the field $\kappa.f{:}t$ of the object pointed to by a value *rec* of type κ below *top*. Its semantics *putfield* $\kappa.f:t$ is (ℓ and *npe* are as before)

$$\lambda\langle l \parallel \mathit{top} :: \mathit{rec} :: s \parallel \mu\rangle. \begin{cases} \langle l \parallel s \parallel \mu[\mu(\mathit{rec}).f := \mathit{top}]\rangle & \text{if } \mathit{rec} \neq \texttt{null}, \\ \underline{\langle l \parallel \ell \parallel \mu[\ell := \mathit{npe}]\rangle} & \text{otherwise.} \end{cases}$$

Exception handling instructions. Some bytecode instructions can throw exceptions, for instance if a `null` value is dereferenced or if memory is exhausted (see examples in the previous paragraphs); others throw exceptions if a class, field, or method cannot be resolved; or if a stack overflow occurs at a method call. Our model considers all such exceptions, as enumerated in Lindholm et al. [30], by building a branch into an exception handler for such bytecode instructions. Instead, it does not consider other exceptions that might be thrown at any bytecode instruction, such as `java.lang.InternalError` and `java.lang.UnknownError`, because the JVM is in an inconsistent state.

Bytecode `throw` κ throws, explicitly, the object of type $\kappa \leq$ `java.lang.Throwable` pointed to by the top of the stack. Its semantics *throw* κ is (ℓ and *npe* are as before)

$$\lambda\langle l \parallel \mathit{top} :: s \parallel \mu\rangle. \begin{cases} \underline{\langle l \parallel \mathit{top} \parallel \mu\rangle} & \text{if } \mathit{top} \neq \texttt{null}, \\ \underline{\langle l \parallel \ell \parallel \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{if } \mathit{top} = \texttt{null}. \end{cases}$$

Bytecode `catch` starts an exception handler from an exceptional state: it transforms it into a normal one, subsequently used by the implementation of the handler:

$$\mathit{catch} = \lambda\underline{\langle l \parallel \mathit{top} \parallel \mu\rangle}.\langle l \parallel \mathit{top} \parallel \mu\rangle$$

where $\mathit{top} \in \mathbb{L}$ is an instance of `java.lang.Throwable`. After `catch`, there is a branch with more following bytecodes, one for each exception handler. Each handler starts with an `exception_is` K bytecode, that selects that handler only for the classes of exception object that it deals with. Otherwise, its post-state is undefined. Namely, `exception_is` K filters the states whose topmost stack element points to an instance of a class in the set $K \subseteq \mathbb{K}$ and ist semantics *exception_is* K is

$$\lambda\langle l \parallel \mathit{top} \parallel \mu\rangle. \begin{cases} \langle l \parallel \mathit{top} \parallel \mu\rangle & \text{if } \mathit{top} \in \mathbb{L}, \mu(\mathit{top}).\kappa \leq \kappa \text{ for some } \kappa \in K, \\ \mathit{undefined} & \text{otherwise.} \end{cases}$$

There is a branch after `catch` also for the case when no handler is selected and the execution falls through to the caller of the method. That branch contains an `exception_is` K for the remaining exception types, not considered by the handlers, followed by a `throw` κ, where κ is the most specific supertype of the types in K.

Method call and return instructions. At the beginning of a method $M = \kappa.m(t_1, \ldots, t_n) : t$, the stack is ε and the local variables hold exactly its $n + 1$ actual arguments (including `this`). (Calls to static methods have only n arguments on the stack, since there is no `this` reference. The formalisation is similar but simpler than that reported below and in accordance with the specification of `invokestatic` in Lindholm et al. [30].) At its end, a `return` t bytecode leaves on the stack only the return value, of type t, or a `return` bytecode just returns, for the case when t = `void`. Without loss of generality, one can assume that `return` is only executed when there is no other value on the stack, except for the returned value. Hence

$$return\ t = \lambda\langle l \parallel top \parallel \mu\rangle.\langle l \parallel top \parallel \mu\rangle$$
$$return = \lambda\langle l \parallel \varepsilon \parallel \mu\rangle.\langle l \parallel \varepsilon \parallel \mu\rangle.$$

Overall, the semantics of the code of M is hence a denotation δ from a state $\langle [v_0, \ldots, v_n] \parallel \varepsilon \parallel \mu\rangle$ to a state $\sigma = \langle l' \parallel top \parallel \mu'\rangle$, with $top = \varepsilon$ when t = `void`, if M returns normally; or to a state $\sigma = \underline{\langle l' \parallel top \parallel \mu'\rangle}$, with top pointing to an exception e, if M throws e. From the point of view of the caller of M, its i local variables l are not affected by the call and the actual arguments $v_0, \ldots, v_n$ are popped from its stack, of height $j = b + n + 1$, and replaced with top (if any). We model this through the operator $extend_M^{i,j} \in \Delta_{n+1,0\to i',r} \to \Delta_{i,j\to i,b+r}$, with $r = 0$ if t = `void` and $r = 1$ otherwise, such that $extend_M^{i,j}(\delta)$ is defined as follows (ℓ and npe are as before)

$$\lambda\langle l \parallel v_n :: \cdots :: v_0 :: s \parallel \mu\rangle. \begin{cases} \underline{\langle l \parallel \ell \parallel \mu[\ell := npe]\rangle} & \text{if } v_0 = \texttt{null} \\ \langle l \parallel top :: s \parallel \mu'\rangle & \text{if } v_0 \in \mathbb{L},\ \sigma \in \Xi, \\ \underline{\langle l \parallel top \parallel \mu'\rangle} & \text{if } v_0 \in \mathbb{L},\ \sigma \in \underline{\Xi}. \end{cases}$$

This allows us to define the semantics of a call to a given method M as $call\ M = extend_M^{i,j}(\delta)$, where δ is the behavior of method M. However, in an object-oriented language, method calls can lead to more runtime target methods, depending on the concrete type of the receiver object. The next section will expand the semantics to that case.

4.3 Fixpoint Denotational Semantics

Dynamic dispatch lets a method call lead to distinct implementations of the method signature. This is an important feature of object-oriented languages. However, it makes static analysis more complex, since it hides the link between callers and callees. There has been an extensive research effort for defining type inference analyses or *class analyses*, with the goal of identifying, for each method call, a superset $\{M_1, \ldots, M_q\}$ of the method implementations that might be called there. This leads to a conservative approximation of the callers/callees relationship. This article, as well as the implementation inside the Julia analyzer, uses the type inference for object-oriented languages defined in Palsberg and Schwartzbach [42]. This builds and solves set-constraints about the possible run-time types of each program variable, method return value and field. For assignments, Palsberg and Schwartzbach [42] build set-constraints that propagate sets of types from rightvalue to leftvalue; for method calls, it builds set constraints that propagate types from actual to formal parameters and from return values to the variables they are assigned to. Other algorithms might be used as well, such as those described in Tip and Palsberg [60], with different precision. The analysis in this article remains sound,

as long as $\{M_1, \ldots, M_q\}$ is an *over*approximation of the actual run-time targets. In the following, we assume that this overapproximation has been already computed and is available, as a decoration of each method call instruction.

Traditionally, the denotational semantics of a program is an *interpretation* that specifies the behavior of each function of the program. In the case of Java bytecode, the interpretation $\iota \in \mathbb{I}$ of P specifies the behavior of each *block* b in P by providing a set $\iota(b)$ of denotations. Here, we use sets since, in an object-oriented language, method calls might be dispatched to more target implementations of the method. Such sets represent possible executions starting at b and continuing with b's successor blocks until a block with no successor is reached (hence ending with `return` or `throw`). Another reason for using sets is that abstract interpretation needs a concrete *collecting* semantics [12], able to model *properties* of denotations, not only the behavior of the program from each single concrete state. For instance, from a very concrete perspective, the constructor of `Node` in Fig. 2 starts in a pre-state where variable `name` points to a string s and ends in a post-state where `name` is unchanged, while `this` is bound to a new location whose field `name` is bound to s and whose field `timestamp` is bound to a new location holding a new string. This description is correct but uselessly concrete for taintedness analysis. Namely, an abstract interpretation, that is only interested in which variables are tainted, would express the semantics of the same constructor as the *set* of denotations where pre-state and post-state keep the taintedness of `name` unchanged, while `this` in the post-state is tainted only if `name` was tainted in the pre-state.

Since the collecting semantics works over sets, the operators *extend* and ; over denotations are consequently extended to sets of denotations.

Definition 4.5 (Interpretation). An *interpretation* is a map from P's blocks into $\wp(\Delta)$, where $\wp$ is the powerset operator, which contains all sets of denotations. The set of interpretations, written as $\mathbb{I}$, is ordered by pointwise set-inclusion. □

Given $\iota \in \mathbb{I}$, we define the set $[\![b]\!]^\iota \subseteq \Delta$ of all the executions, induced by ι, that start at b and continue with b's successors until a block with no successors is reached: we compose sequentially the denotations of the instructions inside b and then compose the result with those of the successor blocks $b_1, \ldots, b_n$, as given by ι. For `call`s, we *extend* the denotations of the first block of the called method(s), as given by ι.

Definition 4.6 (Denotations of Instructions and Blocks). Let $\iota \in \mathbb{I}$. The *denotations in* ι *of an instruction* are

$$[\![\texttt{ins}]\!]^\iota = \{ins\} \qquad \text{if } \texttt{ins} \text{ is not a } \texttt{call}$$

$$[\![\texttt{call } M_1, \ldots, M_q]\!]^\iota = \cup_{1 \le s \le q} extend^{i,j}_{M_s}(\iota(b_{M_s})) \qquad \text{otherwise}$$

where $\{M_1, \ldots, M_q\}$ is a superset of the methods that might be called (this superset can be computed by some class analysis [42]), b_{M_s} is the block where method M_s starts, i the number of local variables and j the height of the stack at the program point where the `call` occurs. Function $[\![\_]\!]^\iota$ is extended to blocks containing instructions $ins_1 \cdots ins_n$, linked to $k \ge 0$ subsequent blocks $b_1, \ldots, b_k$ as

$$\left[\!\!\left[\begin{array}{|c|} \hline ins_1 \\ \cdots \\ ins_n \\ \hline \end{array} \begin{array}{c} \rightarrow \\ \cdots \\ \rightarrow \end{array} \begin{array}{c} b_1 \\ \cdots \\ b_k \end{array} \right]\!\!\right]^\iota = [\![ins_1]\!]^\iota ; \cdots ; [\![ins_n]\!]^\iota \underbrace{; (\iota(b_1) \cup \cdots \cup \iota(b_k))}_{Cont}$$

where the *continuation Cont* is missing when $k = 0$. □

Note that Def. 4.6 uses an operator $\cup$ over $\wp(\Delta)$.

Loops and recursion make the blocks of P interdependent; hence, a denotational semantics needs a fixpoint computation: it starts from the *empty* interpretation $\iota_0 \in \mathbb{I}$, such that $\iota_0(b) = \emptyset$ for all blocks b of P, computes $\iota_1 = T_P(\iota_0)$ and iterates the application of T_P until a fixpoint, which is the *denotational semantics* of P.

Definition 4.7 (Denotational Semantics). We define $T_P : \mathbb{I} \to \mathbb{I}$ as $T_P(\iota)(b) = [\![b]\!]^\iota$ for every $\iota \in \mathbb{I}$ and block b of P. Its least fixpoint exists and can be computed with a (possibly infinite) iterative application of T_P from ι_0 [44]. It is the *denotational semantics* of P. □

Def. 4.7 provides a formal definition of the concrete fixpoint collecting semantics of Java bytecode, explicitly targeted at abstract interpretation. Namely, its abstractions only need to abstract the three concrete operators ;, ∪, and *extend* on $\wp(\Delta)$, *i.e.*, on the subsets of Δ, and the denotation of each single bytecode distinct from `call`. Def. 4.7 does not provide of course an effective way for computing the least fixpoint of T_P, since it might require an infinite number of iterations. But abstractions of T_P over a domain with no infinite ascending chain (such as that in Sec. 5) reach the abstract fixpoint in a finite number of iterations.

Our implementation performs smaller fixpoints on each strongly-connected component of blocks rather than a huge fixpoint over all blocks (as done instead in Def. 4.7). This is irrelevant here for the theoretical results, since any iteration strategy for computing a denotational analysis, without widening and where the operators ; and ∪ are distributive, leads to the same fixpoint. In practice, however, Sec. 7.3 experimentally confirms that smaller local fixpoints are more efficient than a single huge fixpoint.

5 TAINT ANALYSIS

This section defines an abstract interpretation [12] 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 collecting semantics of Sec. 4 works over $\wp(\Delta)$ (sets of denotations). Its formal definition uses sets for each single bytecode instruction, which are actually singletons (sets made of a single $\delta \in \Delta$) since bytecode instructions are deterministic. Such singletons are then merged into larger (potentially infinite) sets of denotations by using three operators ;, ∪, and *extend*, in a bottom-up definition where loops and recursion are saturated by fixpoint. By following the general theory of abstract interpretation [12], an abstract semantics is defined by providing correct abstractions of those singleton sets and of those three operators.

The abstract interpretation of this section can be implemented in an analysis engine for abstract interpretation, as we did with Julia (Sec. 8). The result is a flow-sensitive static analysis, since denotations distinguish a pre-state from a post-state and this gets reflected also in the abstract domain (Def. 5.4). The analysis is also inter-procedural and context-sensitive, since the denotation of method calls is plugged in at each distinct calling point (Def. 5.16). The analysis is based on an object-sensitive notion of taintedness (Def. 5.1) and can be made field-sensitive (Sec. 5.6).

5.1 Abstract Interpretation

First of all, one must formalize the meaning of taintedness for a value. Intuitively, this means that the value is computed from input that the user can freely inject in the application. However, this is perfectly clear for primitive values only. If, instead, a reference value (that is, an object, in our simplified semantics of Sec. 4) gets passed as a parameter to a sink, it is not the exact value (the memory location) that can induce the sink to run a dangerous operation, but rather any value that can be *reached* from that location. In general, the code implementing a sink method might start from that location, follow the pointers and use any reachable information to run an SQL query, build an HTML page, and so on. This means that taintedness for references should be defined as the possibility of *reaching* a tainted primitive value from the reference.

This is reflected in the following formalization of taintedness for a value. Primitive values are explicitly marked as tainted (Def. 4.2), while taintedness for references is indirectly defined in terms of reachability of tainted values. In particular, this notion allows the values of distinct but synonym fields $a.f$ and $b.f$ to have distinct taintedness, depending on the taintedness of variables a and b. In other terms, the choice of a reachability-based notion of taintedness gives rise to the following object-sensitive definition for taintedness.

Definition 5.1 (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).

In the direction of defining an abstraction map from concrete denotations to their taintedness behaviors, 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. States are abstracted into a set of syntactical placeholders l_k, standing for the k-th local variable, s_k, standing for the k stack element (s_0 is the base of the stack), and e, standing for the fact that the concrete state was exceptional.

Definition 5.2 (Tainted Variables). Let state $\sigma \in \Sigma_{i,j}$ be a normal or exceptional state with i local variables l and j stack elements s. Its *tainted variables* are

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

Example 5.3. Consider σ from Ex. 4.3. 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''.

In order to abstract denotations, one must be able to distinguish the abstraction of the pre-state from that of the post-state. Hence, we use distinct variables to abstract the pre-state of a denotation (marked with $\check{}$) and its post-state (marked with $\hat{}$). The mnemonic of these markers is the following: $\check{v}$ stands for the value of v set at the beginning of the denotation; instead, $\hat{v}$ stands for the final value of v, at the end of the denotation, provided to the subsequent instructions. By splitting the same variable at distinct contexts, moreover, one gets a flow-sensitive analysis, since the analysis can approximate the same variable in different ways at distinct program points. If S is a set of identifiers, we extend those notations to sets as $\check{S} = \{\check{v} \mid v \in S\}$ and $\hat{S} = \{\hat{v} \mid v \in S\}$. The abstract domain will consist of Boolean formulas that constrain the relative taintedness of local variables and stack elements. For instance, $\check{l}_1 \rightarrow \hat{s}_2$ states that if local variable l_1 is tainted in the pre-state of a denotation, then the stack element s_2 is tainted in its post-state.

Definition 5.4 (Taintedness Abstract Domain $\mathbb{T}$). Let $i_1, j_1, i_2, j_2 \in \mathbb{N}$. The taintedness abstract domain $\mathbb{T}_{i_1,j_1 \rightarrow i_2,j_2}$ 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 5.5. An example of an abstract domain element is the formula $\phi = (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\check{l}_2 \leftrightarrow \hat{l}_2) \wedge (\check{l}_3 \leftrightarrow \hat{l}_3) \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{s}_0 \leftrightarrow \hat{l}_0) \in \mathbb{T}_{4,1 \rightarrow 4,0}$. It will be shown later (Ex. 5.10) that this is the abstraction of a bytecode instruction `store 0` at a program point with $i = 4$ local variables and $j = 1$ stack elements.

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 5.6 (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 concretization map $\gamma : \mathbb{T}_{i_1,j_1\to i_2,j_2} \to \wp(\Delta_{i_1,j_1\to i_2,j_2})$ given by*

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

In order to prove Prop. 5.6, we need the following lemma.

Lemma 5.7. *The map γ of Prop. 5.6 is co-additive.*

Proof. Let $i_1, i_1, j_2, j_2 \in \mathbb{N}$, $I \subseteq \mathbb{N}$ and $\{\phi_i\}_{i\in I} \subseteq \mathbb{T}_{i_1,j_1\to i_2,j_2}$. We prove that $\gamma(\wedge_{i\in I}\phi_i) = \cap_{i\in I}\gamma(\phi_i)$:

$$\begin{aligned}
\gamma(\wedge_{i\in I}\phi_i) &= \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \;\middle|\; \begin{array}{l} \text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t. } \delta(\sigma) \text{ is defined} \\ \check{tainted}(\sigma) \cup \hat{tainted}(\delta(\sigma)) \models \wedge_{i\in I}\phi_i \end{array} \right\} \\
&= \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \;\middle|\; \begin{array}{l} \text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t. } \delta(\sigma) \text{ is defined} \\ \check{tainted}(\sigma) \cup \hat{tainted}(\delta(\sigma)) \models \phi_i \text{ for all } i \in I \end{array} \right\} \\
&= \bigcap_{i\in I} \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \;\middle|\; \begin{array}{l} \text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t. } \delta(\sigma) \text{ is def.} \\ \check{tainted}(\sigma) \cup \hat{tainted}(\delta(\sigma)) \models \phi_i \end{array} \right\} \\
&= \cap_{i\in I}\gamma(\phi_i).
\end{aligned}$$

□

It is now possible to prove Prop. 5.6.

Proof. The domain $\mathbb{T}_{i_1,j_1\to i_2,j_2}$ is a complete lattice *w.r.t.* logical entailment with $\wedge$ as greatest lower bound operator. The domain $\wp(\Delta_{i_1,j_1\to i_2,j_2})$ is a complete lattice *w.r.t.* set inclusion with $\cap$ as greatest lower bound operator. The map γ is co-additive (Lemma 5.7). Hence, the thesis follows by a general result of abstract interpretation [12]. □

Example 5.8. Consider formula ϕ from Ex. 5.5 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 $((\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\check{l}_2 \leftrightarrow \hat{l}_2) \wedge (\check{l}_3 \leftrightarrow \hat{l}_3))$; moreover, that bytecode 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 $(\check{s}_0 \leftrightarrow \hat{l}_0)$.

5.2 Abstract Semantics of Non-Calling Instructions

Fig. 4 defines correct abstractions for the bytecodes from Sec. 4, except for `call`, which will be considered later in Sec. 5.3. Namely, the semantics of `call` is not a denotation but a transformer of denotations: it plugs the denotation of the callee in the calling point. Therefore its definition and proof of correctness are more involved.

The execution of each single bytecode instruction affects the topmost part of the operand stack, or a local variable; other stack elements and local variables remain unaffected. The exact specification of which stack elements or local variables are affected is in Lindholm et al. [30] where, in particular, it is specified what is *consumed* from the stack or *produced* on the stack by each instruction. All other stack elements or local variables are said to *survive* the execution of the instruction and hence keep their value unchanged. For them, the following definition builds a formula U (for *unchanged*), *i.e.*, a frame condition for input local variables L and stack elements S that survive the execution of a bytecode instruction: the formula U states that their taintedness remains unchanged. The property of remaining

$$
\begin{aligned}
(const\ v)^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge \neg\hat{s}_j \\
(load\ k\ t)^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{l}_k \leftrightarrow \hat{s}_j) \\
(store\ k\ t)^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{l}_k) \\
(add)^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\hat{s}_{j-2} \leftrightarrow (\check{s}_{j-2} \vee \check{s}_{j-1})) \\
(throw\ \kappa)^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge \hat{e} \wedge (\hat{s}_0 \rightarrow \check{s}_{j-1}) \\
(new\ \kappa)^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{s}_j) \wedge (\hat{e} \rightarrow \neg\hat{s}_0) \\
(catch)^{\mathbb{T}} &= U \wedge \check{e} \wedge \neg\hat{e} \\
(getfield\ \kappa.f:t)^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge (\neg\hat{e} \rightarrow (\hat{s}_{j-1} \rightarrow \check{s}_{j-1})) \wedge (\hat{e} \rightarrow \neg\hat{s}_0) \\
(putfield\ \kappa.f:t)^{\mathbb{T}} &= \wedge_{v\in L} R_j(v) \wedge \left(\neg\hat{e} \rightarrow \wedge_{v\in S} R_j(v)\right) \wedge (\hat{e} \rightarrow \neg\hat{s}_0) \wedge \neg\check{e} \\
&\qquad \text{where } R_j(v) = \check{v} \leftrightarrow \hat{v} \text{ if } \neg reach(v, s_{j-2}), \text{ and } R_j(v) = (\check{v} \vee \check{s}_{j-1}) \leftarrow \hat{v}, \text{ if } reach(v, s_{j-2}).
\end{aligned}
$$

Fig. 4. Bytecode abstraction for taintedness, in a program point with j stack elements. Bytecodes not reported in this figure are abstracted into the default $U \wedge \neg\check{e} \wedge \neg\hat{e}$. Sets L (of local variables) and S (of stack elements) are those at the end of the normal execution of `putfield`.

unchanged is always required for the local variables. For the stack elements, U enforces the requirement of being unchanged only when no exception is thrown, since otherwise the JVM drops the full operand stack and the only output stack element is the exception object (Def. 4.4).

Definition 5.9. Let sets S (of stack elements) and L (of local variables) be the input variables that survive all executions of a given bytecode in a given program point (only after the normal executions for S), with unchanged value. Then $U = \wedge_{v\in L}(\check{v} \leftrightarrow \hat{v}) \wedge (\neg\hat{e} \rightarrow \wedge_{v\in S}(\check{v} \leftrightarrow \hat{v}))$.

Consider Fig. 4. A bytecode runs only if the preceding one does not throw any exception ($\neg\check{e}$) except for `catch`, which requires instead an exception to be thrown ($\check{e}$). Bytecode `const` v pushes a constant, hence 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. 4 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 that bytecode 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. 5.1). 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 execution of 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, it must be used $\leftarrow$ instead of $\leftrightarrow$ since the reachability analysis is a *possible* approximation of actual (undecidable) reachability. This is expressed by formula $R_j(v)$, used in Fig. 4.

Example 5.10. According to Fig. 4, 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. 5.5.

Example 5.11. 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. 4.3. According to Fig. 4, the abstraction of that bytecode at p is $\phi' = (\check{l}_0 \leftrightarrow \hat{l}_0) \wedge (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\check{l}_2 \leftrightarrow \hat{l}_2) \wedge (\check{l}_3 \leftrightarrow \hat{l}_3) \wedge (\neg\hat{e} \rightarrow ((\check{s}_0 \vee \check{s}_2) \leftarrow \hat{s}_0)) \wedge (\hat{e} \rightarrow \neg\hat{s}_0) \wedge \neg\check{e} \in \mathbb{T}_{4,3\rightarrow 4,1}$.

The approximation of `putfield` f shows the drawback of using a notion of taintedness based on reachability. Namely, instructions that modify the heap might affect the taintedness of *all* variables that might reach the modified location in the heap, since the portion of reachable heap gets modified by the update. This complicates the approximation of instructions with possible side-effects on the heap, namely, of `putfield` and `call`, since their precise approximation needs the support of preliminary heap analyses, such as possible reachability among variables. More precisely, our taint analysis assumes that three other supporting analyses have been performed in advance and have computed the following predicates:

(1) $reach(v, v')$, which is true if (the location held in) v' is reachable from (the location held in) v.
(2) $share(v, v')$, which is true if from v and v' one can reach a common location.
(3) $updated_M(l_k)$, which 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 undecidable program properties. Our implementation computes these predicates as in Nikolic and Spoto [38], Secci and Spoto [52], and Genaim and Spoto [19], respectively.

The following lemma is used in the proof of correctness for the abstractions in Fig. 4 (Prop. 5.13). It proves that the frame condition is correct for all bytecode instructions distinct from `putfield` and `call`.

Lemma 5.12. *Let* `ins` *be a bytecode distinct from* `putfield` *and* `call` *and let* U *be the formula constructed for* `ins` *according to Def. 5.9. Then* U *is correct w.r.t. ins, i.e.,* $ins \in \gamma(U)$.

Proof. Let S and L be as in Def. 5.9 and $\sigma \in \Sigma$ be such that $\sigma' = ins(\sigma)$ is defined.

Let $v \in L$. Since v survives to all executions of `ins`, it is a local variable of both σ and σ' where it has the same value. For the hypothesis on `ins`, no object reachable from that value is modified by *ins*. Hence, either $\{\check{v}, \hat{v}\} \subseteq \check{tainted}(\sigma) \cup \hat{tainted}(\sigma')$ or $\{\check{v}, \hat{v}\} \cap (\check{tainted}(\sigma) \cup \hat{tainted}(\sigma')) = \emptyset$. In both cases we conclude that $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \check{v} \leftrightarrow \hat{v}$, *i.e.*, $ins \in \gamma(\check{v} \leftrightarrow \hat{v})$.

Let now $v \in S$. If $\sigma' \in \Xi$, since v survives to all normal executions of `ins`, it is a stack element of both σ and σ' where it has the same value. For the hypothesis on `ins`, no object reachable from that value is modified by *ins*. Hence, $\hat{e} \notin \hat{tainted}(\sigma')$ and either $\{\check{v}, \hat{v}\} \subseteq \check{tainted}(\sigma) \cup \hat{tainted}(\sigma')$ or $\{\check{v}, \hat{v}\} \cap (\check{tainted}(\sigma) \cup \hat{tainted}(\sigma')) = \emptyset$, so that $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \neg\hat{e} \rightarrow (\check{v} \leftrightarrow \hat{v})$. If $\sigma' \in \underline{\Xi}$ we have $\hat{e} \in \hat{tainted}(\sigma')$ and also in this case $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \neg\hat{e} \rightarrow (\check{v} \leftrightarrow \hat{v})$. We conclude that $ins \in \gamma(\neg\hat{e} \rightarrow (\check{v} \leftrightarrow \hat{v}))$.

The result follows by Lemma 5.7. □

Proposition 5.13. *The approximations in Fig. 4 are correct w.r.t. the denotations of Sec. 4, i.e., for all bytecode* `ins` *distinct from* `call` *we have* $ins \in \gamma(ins^{\mathbb{T}})$.

Proof. We consider each bytecode instruction. We start from `putfield`, which modifies the memory and whose proof is consequently more complex and interesting.

`putfield` $\kappa.f : t$

Let σ be such that $\sigma' = (\mathit{putfield}\ \kappa.f:t)(\sigma)$ is defined. Let ϕ be the formula for this instruction in Fig. 4. The execution of this instruction can only modify field $\kappa.f:t$ of the object bound to s_{j-2} (the receiver) in σ. For a given variable v in σ, that still exists in σ', if $\neg \mathit{reach}(v, s_{j-2})$ before the instruction then the memory reachable from v in σ is not affected by the execution of this bytecode and the taintedness of v is not affected either (Def. 5.1). It follows that, in this case, $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models \check{v} \leftrightarrow \hat{v}$. If, instead, $\mathit{reach}(v, s_{j-2})$, then from v one reaches tainted data in σ' if that was already possible in σ or if that data was made reachable by this instruction through $\kappa.f:t$, that has been updated to s_{j-1}, from which it must have been possible to reach tainted data then. Hence, in this case we have $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models (\check{v} \vee \check{s}_{j-1}) \leftarrow \hat{v}$. That is, in both cases, we have $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models R_j(v)$. The variables in σ that still exist in σ' are those in L and, if $\sigma' \in \Xi$, also those in S. It follows that $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models \wedge_{v\in L} R_j(v) \wedge (\neg\hat{e} \rightarrow \wedge_{v\in S} R_j(v))$. If $\sigma' \in \underline{\Xi}$ then the top of the stack of σ' is a reference to an untainted exception object and we have $\hat{s}_0 \notin \hat{\mathit{tainted}}(\sigma')$. Moreover, this instruction is only executed from a normal state; hence, $\sigma \in \Xi$. We conclude that $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models (\hat{e} \rightarrow \neg\hat{s}_0) \wedge \neg\check{e}$. We conclude that $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models \wedge_{v\in L} R_j(v) \wedge \big(\neg\hat{e} \rightarrow \wedge_{v\in S} R_j(v)\big) \wedge (\hat{e} \rightarrow \neg\hat{s}_0) \wedge \neg\check{e}$, and hence $\mathit{putfield}\ \kappa.f:t \in \gamma(\phi)$.

For the other bytecodes, by Lemma 5.12 we know that $\mathit{ins} \in \gamma(U)$. Let σ be such that $\sigma' = \mathit{ins}(\sigma)$ is defined. If `ins` is not `catch` then $\sigma \in \Xi$ (Sec. 4). Hence, $\check{e} \notin \check{\mathit{tainted}}(\sigma)$ and $\mathit{ins} \in \gamma(\neg\check{e})$. If instead `ins` is `catch`, we must have $\sigma \in \underline{\Xi}$; hence, $\check{e} \in \check{\mathit{tainted}}(\sigma)$. Then $\mathit{ins} \in \gamma(\check{e})$. By Lemma 5.7, it remains to prove that $\mathit{ins} \in \gamma(\phi)$, where ϕ is the portion of the formulas in Fig. 4 that follows the prefix $U \wedge \neg\check{e}$ (or $U \wedge \check{e}$ for `catch`). We prove it for each kind of bytecode instruction.

`const` v

We have $\phi = \neg\hat{e} \wedge \neg\hat{s}_j$. Since $\sigma' \in \Xi$, we have $\hat{e} \notin \hat{\mathit{tainted}}(\sigma')$. Moreover, the top s_j of the stack of σ' holds v. If $v \in \mathbb{Z}$ or $v =$ `null` then $\hat{s}_j \notin \hat{\mathit{tainted}}(\sigma')$. Value v is a constant value, so it is untainted. We conclude that $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models \phi$; hence, $\mathit{const}\ v \in \gamma(\phi)$.

`load` $k\ t$

We have $\phi = \neg\hat{e} \wedge (\check{l}_k \leftrightarrow \hat{s}_j)$. Since $\sigma' \in \Xi$ we have $\hat{e} \notin \hat{\mathit{tainted}}(\sigma')$. Moreover, the ith local variable of σ is a copy of the top of the stack of σ'. Hence, they are both tainted, in which case $\{\check{l}_k, \hat{s}_j\} \subseteq \check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma')$, or they are both untainted, in which case $\{\check{l}_k, \hat{s}_j\} \cap (\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma')) = \emptyset$. In both cases we have $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models \phi$ and hence $\mathit{load}\ k\ t \in \gamma(\phi)$.

`store` $k\ t$

We have $\phi = \neg\hat{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{l}_k)$. Since $\sigma' \in \Xi$ we have $\hat{e} \notin \hat{\mathit{tainted}}(\sigma')$. Moreover, the top of the stack of σ is a copy of the kth local variable of σ'. Hence, they are both tainted, in which case $\{\check{s}_{j-1}, \hat{l}_k\} \subseteq \check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma')$, or they are both untainted, in which case $\{\check{s}_{j-1}, \hat{l}_k\} \cap (\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma')) = \emptyset$. In both cases we have $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models \phi$; hence, $\mathit{store}\ k\ t \in \gamma(\phi)$.

`add`

We have $\phi = \neg\hat{e} \wedge (\hat{s}_{j-2} \leftrightarrow (\check{s}_{j-2} \vee \check{s}_{j-1}))$. Since $\sigma' \in \Xi$ we have $\hat{e} \notin \hat{\mathit{tainted}}(\sigma')$. Moreover, the top of the stack in σ' (the result) is tainted only if at least one of the operands is tainted in σ. So $\{\hat{s}_{j-2}, \check{s}_{j-2}\} \subseteq \check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma')$ or $\{\hat{s}_{j-2}, \check{s}_{j-1}\} \subseteq \check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma')$ or $\{\hat{s}_{j-2}, \check{s}_{j-2}, \check{s}_{j-1}\} \cap (\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma')) = \emptyset$. In all cases we have $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models \phi$; hence, $\mathit{add} \in \gamma(\phi)$.

`ifne` t

We have $\phi = \neg\hat{e}$. Since $\sigma' \in \Xi$ we have $\hat{e} \notin \hat{tainted}(\sigma')$. We conclude that $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \phi$; hence, $ifne\ t \in \gamma(\phi)$.

`new` κ

We have $\phi = (\neg\hat{e} \rightarrow \neg\hat{s}_j) \wedge (\hat{e} \rightarrow \neg\hat{s}_0)$. If $\sigma' \in \underline{\Xi}$ then the top of the stack is a reference to an untainted exception object, and we have $\hat{s_0} \notin \hat{tainted}(\sigma')$. If $\sigma' \in \Xi$, then the top of the stack of σ' is a reference to a new object of class t, hence untainted. We conclude that $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \phi$; hence, $new\ \kappa \in \gamma(\phi)$.

`throw` κ

We have $\phi = \hat{e} \wedge (\hat{s}_0 \rightarrow \check{s}_{j-1})$. Since $\sigma' \in \underline{\Xi}$ then $\hat{e} \in \hat{tainted}(\sigma')$. Moreover if $\hat{s}_0 \in \hat{tainted}(\sigma')$ then $\check{s}_{j-1} \in \check{tainted}(\sigma)$. Hence, we have $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \phi$. In conclusion, $throw\ \kappa \in \gamma(\phi)$.

`catch`

We have $\phi = \neg\hat{e}$. Moreover, $\sigma' \in \Xi$ and hence $\hat{e} \notin \hat{tainted}(\sigma')$. Then we have $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \phi$. In conclusion, $catch \in \gamma(\phi)$.

`exception_is` K

We have $\phi = \neg\hat{e}$. Since $\sigma' \in \Xi$, we have $\hat{e} \notin \hat{tainted}(\sigma')$. Then we have $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \phi$. In conclusion, $exception\_is\ K \in \gamma(\phi)$.

`return` t

We have $\phi = \neg\hat{e}$. Since $\sigma' \in \Xi$, we have $\hat{e} \notin \hat{tainted}(\sigma')$. Hence, we have $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \phi$. In conclusion, $return\ t \in \gamma(\phi)$.

`return`

We have $\phi = \neg\hat{e}$. Since $\sigma' \in \Xi$, we have $\hat{e} \notin \hat{tainted}(\sigma')$. Hence, we have $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \phi$. In conclusion, $return \in \gamma(\phi)$.

`getfield` $\kappa.f:t$

We have $\phi = (\neg\hat{e} \rightarrow (\hat{s}_{j-1} \rightarrow \check{s}_{j-1})) \wedge (\hat{e} \rightarrow \neg\hat{s}_0)$. If $\sigma' \in \underline{\Xi}$ then the top of the stack is a reference to an untainted exception object, and we have $\hat{s_0} \notin \hat{tainted}(\sigma')$. If $\sigma' \in \Xi$ then, by the definition of taintedness (Def. 5.1), if $\hat{s}_{j-1} \in \hat{tainted}(\sigma')$ then $\check{s}_{j-1} \in \check{tainted}(\sigma)$. We conclude that $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \phi$; hence, $getfield\ \kappa.f:t \in \gamma(\phi)$. □

5.3 Abstract Operators and Abstract Semantics of `call`

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, this is implemented by first renaming them apart into temporary variables and then projecting the temporary variables away.

Definition 5.14. Let $\phi_1, \phi_2 \in \mathbb{T}$. Their *abstract sequential composition* $\phi_1;^{\mathbb{T}}\phi_2$ is

$$\exists_{\overline{V}} \left(\phi_1[\overline{V}/\hat{V}] \wedge \phi_2[\overline{V}/\check{V}]\right)$$

where $\overline{V}$ are fresh overlined variables.

Example 5.15. Consider the execution of `putfield` f at program point p and then `store 0`, as in Ex. 5.11. The former is abstracted by ϕ' from Ex. 5.11; the latter by ϕ from Ex. 5.10. 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*, which makes the analysis context-sensitive by plugging the behavior of a method in each distinct calling context. Let ϕ approximate the taintedness behavior of method $M = \kappa.m(t_1, \ldots, t_n) : t$. The variables occurring in ϕ 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 another b stack elements. The operator plugs ϕ in 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 5.16. 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 $(extend_M^{i,j})^{\mathbb{T}} : \mathbb{T}_{n+1,0\rightarrow i',r} \rightarrow \mathbb{T}_{i,j\rightarrow i,b+r}$ with $r = 0$ if $t =$ `void` and $r = 1$ otherwise, as

$$\neg\check{e} \wedge \exists_{\{\bar{l}_0,\ldots,\bar{l}_{i'}\}}\big(\phi[\hat{s}_b/\hat{s}_0][\bar{l}_k/\hat{l}_k \mid 0 \leq k < i'][\check{s}_{k+b}/\check{l}_k \mid 0 \leq k \leq n] \wedge \bigwedge_{0\leq k<i} A_{b,M}(l_k) \wedge \big(\neg\hat{e} \rightarrow \bigwedge_{0\leq k<b} A_{b,M}(s_k)\big)\big)$$

with

$$SA_{b,M,v} = \{l_k \mid 0 \leq k \leq n,\ share(v, s_{b+k}) \text{ and } updated_M(l_k)\}$$

$$A_{b,M}(v) = \begin{cases} \check{v} \leftrightarrow \hat{v} & \text{if } SA_{b,M,v} = \emptyset \\ (\check{v} \vee (\bigvee_{w\in SA_{b,M,v}} \overline{w})) \leftarrow \hat{v} & \text{otherwise.} \end{cases}$$

Proposition 5.17. *The operators* $;^{\mathbb{T}}$, $extend^{\mathbb{T}}$ *and* $\cup^{\mathbb{T}}$ *are correct.*

Proof. Let $\phi_1, \phi_2 \in \mathbb{T}$, $d_1 \subseteq \gamma(\phi_1)$, and $d_2 \subseteq \gamma(\phi_2)$. We must prove that $d_1; d_2 \in \gamma(\phi_1;^{\mathbb{T}} \phi_2)$. Let $\delta_1 \in d_1$ and $\delta_2 \in d_2$. It is enough to prove that $\delta_1; \delta_2 \in \gamma(\phi_1;^{\mathbb{T}} \phi_2)$. Let σ be such that $(\delta_1; \delta_2)(\sigma)$ is defined, *i.e.*, both $\sigma' = \delta_1(\sigma)$ and $\sigma'' = \delta_2(\sigma')$ are defined (Def. 4.4). From $\delta_1 \in \gamma(\phi_1)$ we conclude that $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \phi_1$. From $\delta_2 \in \gamma(\phi_2)$ we conclude that $\check{tainted}(\sigma') \cup \hat{tainted}(\sigma'') \models \phi_2$. Hence

$$\check{tainted}(\sigma) \cup \{\overline{v} \mid \hat{v} \in \hat{tainted}(\sigma')\} \models \phi_1[\overline{V}/\hat{V}]$$
$$\{\overline{v} \mid \check{v} \in \check{tainted}(\sigma')\} \cup \hat{tainted}(\sigma'') \models \phi_2[\overline{V}/\check{V}]$$

so that $\check{tainted}(\sigma) \cup \{\overline{v} \mid v \in tainted(\sigma')\} \cup \hat{tainted}(\sigma'') \models \phi_1[\overline{V}/\hat{V}] \wedge \phi_2[\overline{V}/\check{V}]$. We conclude that $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma'') \models \exists_{\overline{V}}(\phi_1[\overline{V}/\hat{V}] \wedge \phi_2[\overline{V}/\check{V}]) = \phi_1;^{\mathbb{T}} \phi_2$. Hence, $\delta_1; \delta_2 \in \gamma(\phi_1;^{\mathbb{T}} \phi_2)$.

Let $\phi \in \mathbb{T}_{n+1,0 \to i',r}$ as in Def. 5.16. Let $d \subseteq \gamma(\phi)$. We must prove that for all $i, j \in \mathbb{N}$ with $j = b + n + 1$ and $b \geq 0$ we have $extend_M^{i,j}(d) \subseteq \gamma((extend_M^{i,j})^{\mathbb{T}}(\phi))$, where $extend_M^{i,j}$ has been defined in Sec. 4. Let $\delta \in d$. It is enough to prove that $extend_M^{i,j}(\delta) \in \gamma((extend_M^{i,j})^{\mathbb{T}}(\phi))$. Let $\sigma = \langle l \parallel v_n :: \cdots :: v_0 :: s \parallel \mu \rangle$ be such that $\sigma' = extend_M^{i,j}(\delta)(\sigma)$ is defined. This corresponds to an execution of M from $\sigma'' = \langle [v_0, \ldots, v_n] \parallel \epsilon \parallel \mu \rangle$ to some $\sigma''' = \langle l' \parallel top \parallel \mu' \rangle$. By the definition of $extend_M^{i,j}$, we know that σ and σ' have the same set of local variables with unchanged values; and that when $\sigma' \in \Xi$, the b lowest stack elements are in both σ and σ' and with unchanged value. Let v be one of such unchanged variables. The taintedness of v might well change during the execution of M, but only if M can access a location reachable from v and updates one of the fields of the object at that location, *i.e.*, only if v shares with a parameter of M that gets updated. Hence, if $SA_{b,M,v} = \emptyset$ then the taintedness of v cannot be changed by the call to M and $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \check{v} \leftrightarrow \hat{v}$. If, instead, $SA_{b,M,v} \neq \emptyset$ then v might be tainted at the end of the call if it was already tainted before or if a formal parameter of M in $SA_{b,M,v}$, sharing with v at call-time and updated during the call, is tainted at the end of the call, that is, in σ''' (we have assumed that method parameters cannot be reassigned inside its body, see Sec. 4). Hence, $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \cup \{\overline{x} \mid x \in tainted(\sigma''')\} \models (\check{v} \vee (\bigvee_{w \in SA_{b,M,v}} \overline{w})) \leftarrow \hat{v}$. Since the stack elements remain unchanged only if the call does not throw any exception, by Lemma 5.7 we have $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \cup \{\overline{x} \mid x \in tainted(\sigma''')\} \models \bigwedge_{0 \leq k < i} A_{b,M}(l_k) \wedge (\neg \hat{e} \to \bigwedge_{0 \leq k < b} A_{b,M}(s_k))$. By the definition of σ, σ', σ'', and σ''' and from $\delta \in \gamma(\phi)$ we have $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \cup \{\overline{x} \mid x \in tainted(\sigma''')\} \models \phi[\hat{s}_b/\hat{s}_0][\bar{l}_k/\hat{l}_k | 0 \leq k < i'][\check{s}_{k+b}/\check{l}_k | 0 \leq k \leq n]$. By Lemma 5.7, we conclude that $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \exists_{\{\bar{l}_0, \ldots, \bar{l}_{i'}\}} \big(\phi[\hat{s}_b/\hat{s}_0][\bar{l}_k/\hat{l}_k | 0 \leq k < i'][\check{s}_{k+b}/\check{l}_k | 0 \leq k \leq n] \wedge \bigwedge_{0 \leq k < i} A_{b,M}(l_k) \wedge (\neg \hat{e} \to \bigwedge_{0 \leq k < b} A_{b,M}(s_k))\big)$. By the definition of $extend_M^{i,j}$, we know that $\sigma \in \Xi$, so that $\check{e} \notin \check{tainted}(\sigma)$ and $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models \neg \check{e}$. By Lemma 5.7 and Def. 5.16 we conclude that $\check{tainted}(\sigma) \cup \hat{tainted}(\sigma') \models (extend_M^{i,j})^{\mathbb{T}}(\phi)$. Since σ and σ' are arbitrary, we conclude that $extend_M^{i,j}(\delta) \in \gamma((extend_M^{i,j})^{\mathbb{T}}(\phi))$.

Let $\phi_1, \phi_2 \in \mathbb{T}$, $d_1 \subseteq \gamma(\phi_1)$, and $d_2 \subseteq \gamma(\phi_2)$. We must prove that $d_1 \cup d_2 \subseteq \gamma(\phi_1 \cup^{\mathbb{T}} \phi_2) = \phi_1 \vee \phi_2$. Let $\delta \in d_1 \cup d_2$. It is enough to prove that $\delta \in \gamma(\phi_1 \vee \phi_2)$. If $\delta \in d_1$ then $\delta \in \gamma(\phi_1) \subseteq \gamma(\phi_1 \vee \phi_2)$. If $\delta \in d_2$ then $\delta \in \gamma(\phi_2) \subseteq \gamma(\phi_1 \vee \phi_2)$. □

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 can be implemented as a static analysis tool if one specifies the sources of tainted information and the sinks where the latter should not flow.

5.4 Context-Sensitivity and Recursion

The abstraction of the `call` bytecode plugs the denotation of the callee in the calling context, by using the *extend* operator on denotations (Def. 5.16). Since denotations are functions from a pre-state to a post-state, their post-state can be distinct at each different program point, where a different pre-state occurs. This leads to a context-sensitive analysis. For instance, the analysis of the program in Fig. 1 distinguishes the call to `wrapQuery` at line 14 from that at line 22.

```
private String wrapQuery(String s) {
  return wrapQuery_1(s);
}

private String wrapQuery_1(String s) {
  return wrapQuery_2(s);
}

private String wrapQuery_2(String s) {
  return wrapQuery_3(s);
}

...

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

Fig. 5. An arbitrary long chain of calls. The abstract analysis of `wrapQuery` concludes that its return value is tainted only if its argument is tainted, independently from the length of the chain and independently from any other call that might occur to `wrapQuery_x` elsewhere in the program.

The analyzer infers that the former returns potentially tainted data, whereas the latter returns definitely untainted data. There is no limit to the depth of context-sensitivity that can be achieved by a denotational semantics: the same conclusion is obtained if `wrapQuery` is part of arbitrary long chain of calls, as shown in Fig. 5; as well as if methods `wrapQuery_x` were called from other program points, with any (tainted or untainted) parameter. The analysis always proves that the return value of `wrapQuery` is untainted at any context where the call occurs with an untainted parameter.

As always with denotational semantics, the fixpoint that computes the abstract semantics saturates all possible execution paths, also in recursive methods. Since at each program point there are a fixed number of stack elements and local variables, there are finitely many abstract domain elements at each program point (that is, Boolean formulas about such elements and variables). Hence the fixpoint is always computed in a finite number of fixpoint iterations. For instance, because of the context-sensitivity of the abstraction of `call`, the analysis of the recursive method `padLeft` in Fig. 6 terminates in finite time and concludes that the return value of `padLeft` is tainted only if its parameter is tainted.

5.5 Specification of Sources, Sinks, and Sanitizers

The previous sections have formalized an abstract interpretation that propagates tainted data. This becomes an actual injection analysis if one specifies the origin of tainted data in a web application (sources) and where that tainted data must not flow into (sinks). Moreover, it is advisable to specify *sanitizing* methods, that is, methods that edit data in such a way to make it useless to build an injection attack. Such methods are provided by most web application development frameworks. The exact list of sources, sinks, and sanitizers that our implementation natively recognizes is maintained at https://static.juliasoft.com/docs/latest/injection_sources_sinks.html. Moreover, our implementation also allows the explicit specification of sources, sinks, and sanitizers through program annotations, if they are not already in that

```
// add padding to the left of s until it reaches at least 100 characters
String padLeft(String s) {
  if (s.length() >= 100)
    return s;
  else
    return padLeft(" " + s);
 }
```

Fig. 6. A recursive method: the analysis infers that its return value is tainted only if its parameter is tainted. This result is independent from the specific constant used in `padLeft` (here, 100).

native list.[1] This section clarifies how sources, sinks, and sanitizers fit into the general-purpose analysis of the previous section.

Sources. Some formal parameters or return values must be considered as sources of tainted data, since they can be freely provided by the external world. Our implementation uses a database of library methods for that. They fall into two categories: either they receive tainted data from their formal arguments, such as the `request` argument of `doGet` and `doPost` methods of servlets; or they *generate* tainted data through their return value, as in the case of the return value of console input methods and database query methods. For the first category, the abstract denotations in Fig. 4 are modified at the first bytecode of such methods; for the second category, the abstract semantics of *return* is modified. In both cases, the modified semantics forces to true the formal arguments or return values that are injected tainted data, respectively.

Sinks. Our implementation has a database of library methods that need untainted parameters. That is, 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 functional behavior of P's methods and does not approximate the states *at* the internal program point where a `call` occurs. For that, a preliminary *magic-sets transformation* [44] of the program P adds new blocks of code whose denotation gives information at internal program points, as traditional in denotational static analysis. The analysis of the transformed program computes a formula ψ that holds just before the execution of the `call` instruction. If ψ entails $\neg\hat{v}$ then the `call` receives untainted data for v. Otherwise, the analysis issues an injection alarm.

Sanitizers. Our implementation contains a database of sanitizer methods. A sanitizer method returns a value that is useless for building injection attacks, although it might be computed from user input. In other words, a sanitizer always returns untainted data, which is computed from its (potentially tainted) arguments. Since there is an information flow from their tainted arguments into their return value, the abstract analysis of the previous section concludes that the return value is tainted as well, which is a correct but very conservative approximation. Hence, if the analysis is not manually instructed about the special semantics of sanitizing methods, the analysis of code using such methods will likely result in false alarms.

We have accommodated sanitizing methods into the semantics of the previous section, by modifying the abstraction of *return* inside such methods, in such a way to force to false the abstraction of the returned value. Since, for instance, a function that sanitizes HTML might not be a correct sanitizer for JavaScript, the use of sanitizers requires to run a distinct taint analysis for each kind of source that is sanitized. This increases the cost of the analysis, although the

[1] Experiments in Sec. 7, 8, 9 and 10 have been performed with the native list only, without any extra manual annotation.

supporting analyses are always computed once and the distinct taint analyses share the same computation caches for the BDDs. In our experiments, this occurs only in the OWASP Benchmark (Sec. 10).

5.6 Making the Analysis Field-Sensitive

The approximation of `getfield` f in Fig. 4 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 the latter is conservatively assumed to be tainted. This choice is sound and object-sensitive, but field-insensitive: when $\check{s}_{j-1}$ is tainted, *all* its fields are conservatively assumed to be tainted. This is imprecise: if the program never assigns tainted data to a field f, then the value of field f of $\check{s}_{j-1}$ is untainted, regardless of the taintedness of its container $\check{s}_{j-1}$.

To improve precision (report fewer false positives), we apply a technique pioneered in Spoto [57]. It uses a set of fields *TF* (the possibly-tainted fields, also called the "oracle" *O* [57]) which is a sound over-approximation of *all* fields that might ever hold tainted data.

For `getfield` f, it uses a better approximation than that in Fig. 4: it assumes that the value found in field f can be tainted only if the container of f is tainted *and* $f \in TF$. If, instead, field f is not in *TF*, its value can only be untainted:

$$(getfield\ \kappa.f:t)^{\mathbb{T}} = \begin{cases} U \wedge \neg\check{e} \wedge (\neg\hat{e} \rightarrow (\hat{s}_{j-1} \rightarrow \check{s}_{j-1})) \wedge (\hat{e} \rightarrow \neg\hat{s}_0) & \text{if } \kappa.f:t \in TF \\ U \wedge \neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\check{s}_{j-1}) \wedge (\hat{e} \rightarrow \neg\hat{s}_0) & \text{otherwise.} \end{cases} \tag{1}$$

We soundly compute *TF* through an iterative algorithm [57]. The algorithm starts with $TF = \emptyset$ and runs the analysis in Sec. 5, but using (1) for `getfield` instead of the equation in Fig. 4. Since this analysis is performed with $TF = \emptyset$, it will likely be unsound. However, at its end, the algorithm scans the program looking for `putfield` g instructions and adding g to *TF* if the analysis did not infer the value stored there inside field g as definitely untainted. This process is iterated with a new analysis, this time using this larger set *TF*. Since this analysis uses a larger *TF* than the previous analysis, it will infer fewer variables as untainted. Hence, at its end, there might be more fields g to add to *TF*. This mechanism is iterated until no more fields can be added to *TF*. As proved in Spoto [57], this iteration of unsound analyses eventually converges to a sound overapproximation of *TF* and the last analysis of the iteration is sound.

In practice, repeated analyses with larger and larger *TF* are made efficient by caching abstract computations, so that identical abstractions are reused in subsequent iterations. On average, this process converges in around 5 iterations, even for large programs. By using caching, this only doubles the overall time of the analysis. Since preliminary heap analyses are typically more expensive than information flow analysis though Boolean formulas, this technique increases the total time by around 25% on average. (Sec. 9 and 10 show effects on cost and precision.)

This technique is not identical to statically, manually classifying fields as tainted and untainted, as Barthe et al. [6], and Genaim and Spoto [18] do. The classification of the fields is here dynamic, depending on the program under analysis, and completely automatic. Moreover, the analysis is also object-sensitive, hence a field might be in *TF* (that is, potentially tainted) but the analyzer might still consider its value as untainted, because its container is untainted. Of course, as always in abstract interpretation, this technique is approximate and can induce imprecision. For instance, if two variables `a` and `b`, of the same type, are such that `a.f` and `b.g` are assigned tainted values in the program, but `a.g` and `b.f` are only assigned untainted values, then both fields `f` and `g` will be in the *TF* set and both variables will be marked as tainted, since both reach tainted data. Hence the analysis will falsely consider `a.g` and `b.f` as tainted. This

approximation can introduce false alarms and justifies why a thorough experimental evaluation is needed in order to assess the precision of the analysis in practice (Sec. 7, 8, 9 and 10).

6 IMPLEMENTATION

Our ideas can be applied to any language and implemented in any framework. We implemented the analysis of Sec. 5 inside the Julia static analyzer for Java and Android [58], a commercial tool distributed by JuliaSoft Srl (http://www.juliasoft.com). We chose Julia because it simplified our work. Julia contains frameworks for denotational and constraint-based analyses that have already been used for the implementation of many static analyses. Our implementation uses the the framework for denotational analyses to define our denotational, bottom-up taint analysis via the three abstract operators and the abstract denotations of Sec. 5. Julia handles code parsing and provides a fixpoint engine and an oracle mechanism for using and soundly computing the possibly-tainted field set *TF* (Sec. 5.6). Moreover, Julia includes a magic-set transformation for Java bytecode that can recover abstract information at program points from any denotational analysis [44]. Julia also includes implementations of the supporting heap analyses, such as sharing and reachability.

The implementation of the analysis of Sec. 5 consisted in coding the specific abstraction of each bytecode instruction (Fig. 4) and of the *extend* operator (Def. 5.16). The implementation of disjunction is trivial, while that of the sequential operator (Def. 5.14) coincides with that for other Boolean-based static analyses such as nullness [57] and we just copied it from there.

Boolean formulas have been implemented through binary decision diagrams [9] (BDDs), by using the JavaBDD library [67]. The Julia analyzer already includes a framework for developing denotational analyses through BDDs. This framework maps, into Boolean variables, the exception mark e and each stack element s_k and local variable l_k, in pre-state, post-state (Def. 5.4) and overlined (Def. 5.14) versions. Function η formalizes this mapping into Boolean variables, numbered through natural numbers. From the point of view of correctness and precision of the analysis, the exact specification of η is irrelevant, since it affects neither correctness nor precision, as long as it is injective. However, different choices of η change the size of the BDDs and consequently also the cost of the analysis, as it will be shown in Sec. 7.4.

A first, injective definition of η could be the following:

$$\begin{aligned}
\eta(\check{e}) = 0, \quad \eta(\hat{e}) = 1, \quad \eta(\overline{e}) = 2 \\
\eta(\check{s}_k) &= 3(2k) \\
\eta(\hat{s}_k) &= 3(2k) + 1 \\
\eta(\overline{s}_k) &= 3(2k) + 2 \\
\eta(\check{l}_k) &= 3(2k + 1) \\
\eta(\hat{l}_k) &= 3(2k + 1) + 1 \\
\eta(\overline{l}_k) &= 3(2k + 1) + 2
\end{aligned}$$

which corresponds to a distribution of e followed by stack elements and local variables, alternatively. The distinction of pre-state, post-state and overlined variables is achieved by multiplying by 3 the code of the variable and adding a different offset (0, 1 or 2).

The first drawback of the previous definition of η is that, at each given program point, there are local variables or stack elements that are irrelevant for the analysis, since their use is forbidden. This happens because such variables are defined only for some but not all execution paths that reach the instruction (see the type inference algorithm in Lindholm et al. [30]), or because they are the second half of a 64-bit value (`long` or `double`). Allocating a slot for representing such variables is overkill: it increases the time of analysis by increasing the number of BDD variables, the size of the BDDs, and the amount of garbage collection on the BDD table. Another drawback, with the same negative effects, is that, at each given program point, the number of stack elements and local variables can be quite different (in general, there are many more local variables than operand stack elements). When that happens, the previous definition of η produces *holes* in the representation in terms of BDD variables, since it keeps mapping, in alternation, stack elements and local variables.

In order to overcome the previous drawbacks, Julia uses another definition of η, which compacts the BDDs by using smaller, program-point-specific numbers to represent stack elements and local variables. Namely, consider a program point with i stack elements and j local variables in scope. Let

$$\text{index used to represent } s_k\text{:} \quad \xi_s(k) = k - (\text{number of unused stack elements from } s_0 \text{ to } s_{k-1})$$

$$\text{index used to represent } l_k\text{:} \quad \xi_l(k) = k - (\text{number of unused local variables from } l_0 \text{ to } l_{k-1})$$

$$\text{index after which stack and locals do not alternate:} \quad m = \min(\xi_s(i), \xi_l(j)) \ .$$

Note that Julia knows which variables are unused at each given program point, since it applies the type inference algorithm in Lindholm et al. [30]. The definition of η used by Julia is:

$$\eta(\check{e}) = 0, \quad \eta(\hat{e}) = 1, \quad \eta(\overline{e}) = 2$$

$$\eta(\check{s}_k) = \begin{cases} 3(2\xi_s(k)) & \text{if } \xi_s(k) \leq m \\ 3(\xi_s(k) + m) & \text{if } \xi_s(k) > m \end{cases}$$

$$\eta(\hat{s}_k) = \begin{cases} 3(2\xi_s(k)) + 1 & \text{if } \xi_s(k) \leq m \\ 3(\xi_s(k) + m) + 1 & \text{if } \xi_s(k) > m \end{cases}$$

$$\eta(\overline{s}_k) = \begin{cases} 3(2\xi_s(k)) + 2 & \text{if } \xi_s(k) \leq m \\ 3(\xi_s(k) + m) + 2 & \text{if } \xi_s(k) > m \end{cases}$$

$$\eta(\check{l}_k) = \begin{cases} 3(2\xi_l(k) + 1) & \text{if } \xi_l(k) \leq m \\ 3(\xi_l(k) + m) & \text{if } \xi_l(k) > m \end{cases}$$

$$\eta(\hat{l}_k) = \begin{cases} 3(2\xi_l(k) + 1) + 1 & \text{if } \xi_l(k) \leq m \\ 3(\xi_l(k) + m) + 1 & \text{if } \xi_l(k) > m \end{cases}$$

$$\eta(\overline{l}_k) = \begin{cases} 3(2\xi_l(k) + 1) + 2 & \text{if } \xi_l(k) \leq m \\ 3(\xi_l(k) + m) + 2 & \text{if } \xi_l(k) > m. \end{cases}$$

Since Julia transparently implements this compaction already, the developer of a static analysis gets its benefit without even noticing the existence of this optimization.

Program	Category	LoC	LoC with Libs	Time (seconds)
Snake&Ladder	game	794	17818	24
MediaPlayer	entertainment	2634	87368	134
EmergencySNRest	web service	3663	42540	53
FarmTycoon	game	4005	69659	128
Abagail	mach. learn.	12270	49243	41
JCloisterZone	game	19340	116858	199
JExcelAPI	scientific	34712	67944	101
Colossus	game	77527	194994	446

Fig. 7. The open source programs analyzed. **LoC** are the non-blank non-commented lines of source code; **LoC with Libs** includes the reachable lines of the libraries that Julia analyzed, as counted by Julia during the construction of the model of the program to analyze. This number is inferred by using the debug information in the bytecode, that contains the source line of each bytecode instruction. **Time** is the full time of analysis of the **LoC with Libs**; hence this includes the parsing of the application, the computation of the supporting analyses and of the taint analysis, as well as the identification of potential injection attacks, for all categories of injections.

As reported in Sec. 5, our taint analysis requires three supporting analyses: reachability [38], sharing [52] and update or constancy [19]. They are already implemented inside Julia. Hence, the existence of the Julia frameworks for static analysis reduced the implementation effort to around two weeks, followed by two more weeks for the look-up and specification of sources, sinks, and sanitizers from the literature (Sec. 5.5), for frequently-used Java frameworks such as Spring and Hibernate. The code explicitly written for taint analysis and injection identification, hence without the supporting analyses and code that were already available in the Julia framework, amounts to around 3000 non-comment non-blank new Java lines.

The next two sections show that our analysis is scalable and effective on real-world code, as demonstrated by analysis of open-source code (Sec. 7) and of closed-source code of customers of JuliaSoft Srl (Sec. 8). The following two sections compare our analysis to other techniques, on standard benchmarks for static analysis of security properties of Java code (Sec. 9–10).

The experiments were performed on a Linux machine based on an Intel quad-core i5-4460 CPU running at 3.20GHz, with 16 gigabytes of RAM. Julia is a whole-program analysis that analyzes a program together with its libraries.

7 EXPERIMENTS ON OPEN SOURCE APPLICATIONS

We ran the injection analysis[2] on a set of open-source applications. These experiments show that our analysis applies to real, third-party software finding significant and non-trivial security issues. They also enable other researchers to reproduce our results and compare their tools to ours. Complete applications have been chosen, instead of libraries, so that there is no need to specify which method parameters should be assumed to receive tainted data from outside, during the analysis. We have chosen applications with a variety of input forms, such as servlet requests, graphical widgets, external files, and sockets. Moreover, the size of such applications has been chosen to show the scalability of the analysis and yet allow manual verification of every warning.

For each application (Fig. 7), the injection analysis of Julia reports a list of warnings. We manually classified each warning as a true alarm or a false alarm: see Fig. 8. In a few cases, we could not determine this classification, because the code was too large and complex. In those cases, we have conservatively classified the dubious warnings as

[2]Henceforth, *our injection analysis* means analysis of Sec. 5, with the field-sensitive approach in Sec. 5.6 unless stated otherwise.

Program	Addr	DOS	Eval	HS	Log	Path	Refl	Res	Sess	SQL	URL	XSS	Total
Snake&Ladder	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/1	**0**/0	**0**/0	**4**/0	**32**/0	**0**/0	**4**/0	**40**/1
MediaPlayer	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0
EmergencySNRest	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**3**/1	**0**/0	**0**/0	**3**/1
FarmTycoon	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**1**/0	**0**/0	**0**/0	**4**/0	**0**/0	**0**/0	**5**/0
Abagail	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0
JCloisterZone	**1**/0	**0**/2	**0**/0	**0**/0	**0**/0	**1**/0	**5**/5	**0**/0	**0**/0	**0**/0	**1**/0	**0**/0	**8**/7
JExcelAPI	**0**/0	**0**/0	**0**/0	**0**/0	**0**/0	**3**/1	**0**/0	**0**/0	**0**/0	**0**/0	**1**/1	**0**/0	**4**/2
Colossus	**0**/0	**2**/0	**0**/0	**0**/0	**663**/189	**23**/11	**0**/4	**14**/0	**0**/0	**0**/0	**0**/1	**0**/0	**702**/205

Fig. 8. Warnings issued by Julia for the open-source applications of Fig. 7. Each cell of the table reports true and false alarm counts as **#true**/**#false**. **Addr** are untrusted data used as an IP address; **DOS** are untrusted data that can determine a denial-of-service; **Eval** are injections into an expression evaluation routine, that is, a special case of command injection; **HS** are injections into Http redirection routines (*http-splitting*); **Log** are untrusted data dumped on logs; **Path** are untrusted data flowing into file or directory names; **Refl** are untrusted data flowing into reflective methods; **Res** are untrusted data used for socket creations (*resource injection*); **Sess** are untrusted data stored into servlet sessions; **SQL** are untrusted data used for SQL queries; **URL** are untrusted data used for connecting to URLs; **XSS** are untrusted data dumped to the output of a servlet (*cross-site scripting*).

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

  String bname=request.getParameter("Bname");
  Connection con=null;
  Statement st=null;
  ResultSet rs=null;
  response.setContentType("text/plain");
  PrintWriter out=response.getWriter();
  DBConnection db=new DBConnection();
  con=db.getConnection();
  st=con.createStatement();
  rs=st.executeQuery("select * from game_details where game_name='"+bname+"';"); // line 53
  ...
}
```

Fig. 9. A possible SQL-injection in class `com.lbp.jsoclasses.CheckBname.java` in Snake&Ladder.

false alarms. All experimental data and the source code of the applications are available at https://github.com/spoto/Injection-Experiments.git.

In total, Julia issued 978 warnings. Of these, 762 are true alarms and 216 are false alarms (or could not be classified). Most true alarms are due to user input that flows into sink methods; or to data read from external files that flows to sink methods; or program options that can be freely modified by system administrators and flow to sink methods. In Colossus, in particular, there are hundreds of injections into log files. This is because that game logs many messages containing the name of the opponents, that the user can freely insert. They are hence actual injections, although probably harmless. In some cases, however, they can be very dangerous, as explained below.

We now discuss a representative set of true and false alarms that we found with the injection analysis.

```
// in com.lbp.SessionListenerClasses.RemoveSession.java
public void removeSession(String uname) {
  Connection con=null;
  Statement st=null;
  ResultSet rs=null;
  DBConnection db=new DBConnection();
  con=db.getConnection();
  st=con.createStatement();
  rs=st.executeQuery(                              // line 25
    "select game_name from game_details where player_name='"+uname+"';"); ...
}

// in com.lbp.jspclasses.Logout.java
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
  HttpSession s=request.getSession();
  RemoveSession rs=new RemoveSession();
  rs.removeSession(s.getAttribute("uname").toString()); // line 33
  s.invalidate();
  response.sendRedirect("index.jsp");
}

// in com.lbp.servletclasses.Login.java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html");
  DBConnection db=new DBConnection();
  Connection con=db.getConnection();
  Class.forName("com.mysql.jdbc.Driver").newInstance();
  Statement st=con.createStatement();
  String uname=request.getParameter("Email");
  String pass=request.getParameter("Password");
  ResultSet rs=st.executeQuery("select * from user_details where email='"+uname+"'");
  HttpSession s=request.getSession();
  s.removeAttribute("uname");
  s.removeAttribute("Name");

  if(rs.next()){ ...
    s.setAttribute("uname", uname); // line 60: attribute uname becomes tainted here
  } ...
}
```

Fig. 10. An interprocedural flow of tainted data in Snake&Ladder, passing through a session attribute uname, that leads to an SQL-injection.

7.1 True Alarms

Julia issues an SQL-injection warning at line 53 of class `com.lbp.jspclasses.CheckBname.java` in Snake&Ladder. The (simplified) code snippet is in Fig. 9. This is a true alarm, direct and intraprocedural, from the `Bname` request parameter of a servlet through the `bname` local variable into the `executeQuery` method, which runs an SQL query on the database.

In the previous example, tainted data comes from a request parameter, which is always a source, since it can be freely controlled by the user. Consider now another SQL-injection warning, which Julia issues at line 25 of class `com.lbp.SessionListenerClasses.RemoveSession.java`, again in Snake&ladder. The code snippet is the upper part of Fig. 10. Variable `uname` is a formal parameter of method `removeSession`: this time, Julia applies interprocedural reasoning. Namely, this method is called at line 33 of class `com.lbp.jspclasses.Logout.java`, shown in the middle of the same figure, where a session attribute is passed as an actual argument. Session attributes cannot be explicitly set by the user; hence, method `getAttribute` is not a source of tainted data, in general. Nevertheless, the session object can become tainted if it can reach tainted data, as for all objects in our reachability-based taint analysis. This is what happens at line 60 of `com.lbp.servletclasses.Login.java`, shown at the end of Fig. 10. Here, tainted data coming from the servlet parameter `Email` is stored, at login time, into attribute `uname`, which is later recovered at logout time and sent to `removeSession`, where it is used to build an SQL query. This allows a user to trigger an SQL-injection.

Julia issues an XSS-injection at line 64 of class `com.lbp.jspclasses.PositionUpdate.java`, whose code is shown (in simplified form) in Fig. 11. This is a true and exploitable alarm. Also in this case, its explanation requires to reason in an interprocedural way. Namely, at line 64, the return value of `gn.getPlayerNames(bname)` is printed on the response output stream of a servlet and consequently leads to an XSS-injection if that return value is tainted. This can actually be the case. There are two possible attacks here. The first is simpler to spot but harder to exploit. The second is more complex but easily exploitable.

Let us start from the first attack. The attribute `bname` passed to `getPlayerNames` is tainted since it holds data that can be freely provided by the user, as shown at line 76 of `com.lbp.servletclasses.Join.java` (Fig. 11, in the middle). Hence `getPlayersNames` can return tainted data if its last instruction `return bname` is reached. This happens, for instance, if there is an error while opening the database connection or while running the query against it. Hence, this is a real attack path, but unlikely to be exploitable if the attacker cannot control the status of the database.

The second attack scenario is, instead, fully exploitable. To understand how, it must first be understood how the Snake&Ladder game works. That is a multiplayer networked game, that allows one to define shared boards to which many players can take part and compete. A player joins a board by specifying the name `bname` of the shared board and by choosing a color `die`, not yet taken by any other player. That color is just an arbitrary string: the program does not check its integrity. While a player is playing, his username is held in the session attribute `uname` and the name of the board is held in the session attribute `bname`. The second attack scenario occurs because the SQL query run at line 75 of `com.lbp.servletclasses.Join.java` taints the third column of the database table `player_details`: variable `die` is the color chosen by the user of the servlet, is in his full control and is stored as third element of a tuple inserted in that table (Fig. 11). That color (actually, an arbitrary string) is then extracted in `getPlayersName` and used to build the response page held in `str`.

The following is hence an exploitation attack pattern:

(1) the attacker joins a shared board of Snake&Ladder, named `bname`. The exact name is irrelevant here, but the more players (*i.e.*, potential victims of the attack) the better. For that, it uses servlet `com.lbp.servletclasses.Join` (Fig. 11). The critical bit is the choice of the serlvet parameter `die`: the attacker will choose his color `die` in such a

```
// in com.lbp.jspclasses.PositionUpdate.java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
  PrintWriter out=response.getWriter();
  HttpSession hs=request.getSession();
  String bname=hs.getAttribute("bname").toString();
  ...
  GetPlayerNames gn=new GetPlayerNames();
  out.print(gn.getPlayerNames(bname)); // line 64
}

// in com.lbp.servletclasses.Join.java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
  String bname=request.getParameter("bname"), die=request.getParameter("die");
  ... open database connection and create SQL statement st ...
  HttpSession s=request.getSession();
  String name=s.getAttribute("uname").toString();
  // ... if color not taken ...
  // in the following query, the third column receives tainted data from variable die
  st.addBatch("insert into player_details values('" + name + "','" + bname + "','" + die ...;");
  ...
  st.executeBatch(); // line 75: this taints the third column of table player_details
  s.setAttribute("bname", bname); // line 76: attribute bname becomes tainted here
  ...
}

// in com.lbp.jspclasses.GetPlayersName.java
public String getPlayerNames(String bname){
  try {
    // ...open DB connection...
    Statement st=con.createStatement();
    ResultSet rs=st.executeQuery("select * from player_details where game_name='"+bname+"';");
    String str="<table style=\"border:none;\" id='players'>";
    ... starts looping oever rs and building str
    // the following is repeated at both lines 30 and 33
    str=str+"<td colspan=\"2\"> <p Style=\"color:'Black'; background-color:"+rs.getString(3)+ ...;
    ...
    return str;
  }
  catch (Exception ex) {}
  return bname;
}
```

Fig. 11. An interprocedural flow of tainted data in Snake&Ladder, passing through a session attribute `bname`, that leads to an XSS-injection.

way that, when concatenated into `str` inside `getPlayerName`, it closes the ⟨p⟩ HTML tag (by specifying an arbitrary color), inserts arbitrary (typically, harmful) HTML code and opens another ⟨p⟩ tag: s'' =`"white⟩ arbitrary HTML code ⟨p"`;

(2) at this moment, the attacker has inserted a bomb into the game: if another player of the same board `bname` (the victim) will run the `com.lbp.jspclasses.PositionUpdate` servlet, this will call `getPlayerNames` and show a dump of the information about the players playing at the same board `bname`. The concatenation of color s'' inside `str` (lines 30 and 33 of `getPlayerNames`) will insert arbitrary, harmful HTML code into the HTML page held in `str` and returned by `getPlayerName`;

(3) that page is sent to the browser of the victim (line 64 of `com.lbp.jspclasses.PositionUpdate`), that shows the page and runs the harmful HTML code.

This second scenario is actually a stored XSS attack: first tainted, harmful data is stored in the database (step 1), then that data is extracted from the database and sent to the victim's browser (steps 2 and 3). Moreover, the severeness of this attack is unaffected by the fact that Snake&Ladder is just a game: the attack is severe since arbitrary HTML code ends being run in the victim's browser. Snake&Ladder is just a Trojan horse here.

Let us consider now some of the many log injections that the analyzer issues in Colossus (Fig. 8). A dangerous one is at line 2054 of `net.sf.colossus.webclient.WebClient.java`, shown in the topmost snippet in Fig. 12. There the name, password, and confirmation code of every newly registered player are logged. They are formal parameters of method `createRegisterWebClientSocketThread`. Interprocedural reasoning leads Julia to match them with variables `name`, `newPW1`, and `providedConfCode` passed at line 396 of `net.sf.colossus.webclient.RegisterPasswordPanel.java`, shown at the bottom of the same figure. The value of variable `providedConfCode` is inserted into a `JOptionPanel`, which is a Java dialog where the user can type any input string. Hence, the analyzer considers it as a source of tainted data. Variables `name` and `newPW1` can instead be traced back, interprocedurally, to line 316 of the same source file, where they are read from the `JTextField rploginField` and the `JPasswordField newPW1`. Both are Java widgets where the user can freely insert any string. Hence, the analyzer considers them as sources of tainted data. In conclusion, anybody having access to the logs can read the unencrypted name, password, and confirmation code of all registered players.

Another dangerous log injection occurs at line 788 of `net.sf.colossus.webserver.WebServerClient.java`, in Colossus, shown in the middle of Fig. 13. There, a chat `message` received from `sender` is logged, in full and unencrypted. The value of those variables can be traced back, interprocedurally, to line 485 of the same source file, shown above in the same figure. That is method `parseLine`, which runs different commands on the basis of the input formal parameter `fromClient`. For case `IWebServer.ChatSubmit`, `sender` and `message` are extracted from the string `fromClient`. Tracing back the latter, interprocedurally, leads to method `run` of a thread, shown in the bottom of Fig. 13, where `fromClient` is a line read from an input stream `in`, connected to a `socket`. Hence, any data read from that stream is tainted, since it can be arbitrarily injected from outside. In this example, the log injection is both a privacy issue (any chat message is logged and becomes available to anybody having access to the logs) and a risk of DOS attacks (a malicious server can send very long strings and explode the size of the client's logs).

Julia warns of a DOS injection at line 180 of `net.sf.colossus.common.WhatNextManager.java`, shown at the top of Fig. 14. There, the current thread is made to sleep for `millis` milliseconds. Interprocedural reasoning traces formal parameter `millis` back to the value of formal parameter `beepInterval` passed for it at line 3641 of `net.sf.colossus.webclient.WebClient.java` (Fig. 14). Going back further, line 609 of `net.sf.colossus.webclient.WebClientSocketThread.java` passes `Long.parseLong(tokens[6])` for that formal parameter, which is part of an arbitrary string `fromServer` coming from a

```
// in net.sf.colossus.webclient.WebClient.java
public String createRegisterWebClientSocketThread
    (String username, String password, String email, String confCode)
{
  LOGGER.info("Creating a RegisterWCST, username " + username // line 2054
              + " password " + password + " and confcode " + confCode); ...
}

// in net.sf.colossus.webclient.RegisterPasswordPanel.java
private void buttonPressedActualAction() { ...
  String name = rploginField.getText();
  String newPW1 = new String(rpNewPW1.getPassword()); ...
  if (isRegister) { ...
    String email = rpEmailField.getText(); ...
    handleConfirmation(name, newPW1, email); // line 316
  }
}

// again in net.sf.colossus.webclient.RegisterPasswordPanel.java
private void handleConfirmation(String name, String newPW1, String email) {
  boolean done = false;
  while (!done) {
    String providedConfCode = JOptionPane.showInputDialog(this, ...);
    if (providedConfCode == null) { done = true }
    else if (providedConfCode.equals(User.TEMPLATE_CONFCODE)  providedConfCode.equals(""))
      JOptionPane.showMessageDialog(this, "Confirmation code must not be empty'');
    else {
      providedConfCode = providedConfCode.trim();
      String reason2 = webClient.createRegisterWebClientSocketThread // line 396
        (name, newPW1, email, providedConfCode); ...
    } ...
  }
}
```

Fig. 12. Colossus logs the email, password, and confirmation code of every newly-registered player.

remote server: see the implementation of `getOneLine` and the assignment to field `in` in method `register` in Fig. 14. In conclusion, a malicious server can send a large value that makes the program sleep for a long time, freezing the game. This example shows that the analyzer interprets the semantics of exception handling constructs correctly (see method `getOneLine`) and tracks information flow through fields (such as `in`). Later (Sec. 7.6) it will be shown an example where this ability of the analyzer to follow exceptional paths and tainted exception objects in a precise way is necessary to identify a dangerous remote command injection and execution.

```
// in net.sf.colossus.webserver.WebServerClient.java
public boolean parseLine(String fromClient) { ...
  String[] tokens = fromClient.split(sep);
  String command = tokens[0]; ...
  if (command.equals(IWebServer.ChatSubmit)) {
    String chatId = tokens[1];
    String sender = tokens[2];
    String message = tokens[3];
    processChatLine(chatId, sender, message); // line 485
  } ...
}

public void processChatLine(String chatId, String sender, String message) { ...
  LOGGER.finest("Chat msg from user " + sender + ": " + message); // line 788
  ...
}

// in class net.sf.colossus.webserver.WebServerClientSocketThread.java
public void run() {
  String fromClient;
  BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream())); ...
  while (!done && fromClient != null) { ...
    fromClient = in.readLine();
    if (fromClient != null) { ...
      done = theClient.parseLine(fromClient); ...
    } ...
  } ...
}
```

Fig. 13. Three snippets from classes of Colossus: they log any chat message received from a remote opponent through a socket, with no check on the length of the logged message.

7.2 False Alarms

Julia issues a path-injection warning at line 60 of `lbp.jspclasses.DBConnection.java`, shown in Fig. 15. There, a file is created, whose name is in variable `cleaner`, *i.e.*, in the formal parameter `s` of method `trustMe`. The latter is called only at line 48 of `com.lbp.jspclasses.GetColors.java` where tainted data, coming from servlet parameter `bnamee`, is passed as `s`. As a consequence, the analyzer suspects that a servlet parameter can be used to access any file in the file system, allowing a path-traversal attack. However, the first parameter `t` passed to `trustMe` at line 48 is fixed to 2; hence, line 60 of `lbp.jspclasses.DBConnection.java` is not reached. Since our analysis abstracts away the exact numerical value of variables, it cannot determine this fact and issues a false alarm.

Julia issues a reflection-injection warning at line 138 of `com.jcloisterzone.ui.Client.java`, shown in Fig. 16. There, a call to the reflection method `getField` gets the static field of class `java.awt.Color` named `colorName`. Namely, library class `java.awt.Color` contains public static fields for common colors. Method `getField` accesses one such static field, if `colorName` is not the RGB specification of a color, which is handled a few lines above. It is actually the case that `colorName`

```
// in net.sf.colossus.common.WhatNextManager.java
public static void sleepFor(long millis) {
  try { Thread.sleep(millis); } // line 180
  catch (InterruptedException e) { ... }
}

// in class net.sf.colossus.webclient.WebClient.java
public void requestAttention(..., int beepCount, long beepInterval, ...) { ...
  for (int i = 1; i <= beepCount; i++) { ...
    if (i < beepCount)
      WhatNextManager.sleepFor(beepInterval); // line 3641
  }
}

// in class net.sf.colossus.webclient.WebClientSocketThread.java
public void run() {
  String fromServer; ...
  while (!done && (fromServer = getOneLine()) != null) {
    String[] tokens = fromServer.split(sep, -1);
    String command = tokens[0]; ...
    if (command.equals(IWebClient.requestAttention)) { ...
      int beepCount = Integer.parseInt(tokens[5]);
      long beepInterval = Long.parseLong(tokens[6]);
      webClient.requestAttention(..., beepCount, beepInterval, ...); // line 609
    }
  } ...
}

// in class net.sf.colossus.webclient.WebClientSocketThread.java
public String getOneLine() throws IOException {
  String line = "No line - got exception!";
  try { line = this.in.readLine(); }
  catch (IOException e) { ... throw e; }
  return line;
}

// in class net.sf.colossus.webclient.WebClientSocketThread.java
private void register() throws ... {
  ... this.in = new BufferedReader(new InputStreamReader(socket.getInputStream(), charset)); ...
}
```

Fig. 14. Snippets from classes of Colossus: they receive an arbitrary numerical value from a remote server, interpret it as a milliseconds delay and sleep for that time.

```
// in com.lbp.jspclasses.DBConnection.java
public String trustMe(int t, String s) {
  String cleaner = s;
  if (t == 0) { ... }
  else if (t == 1) {
    File dPath = new File(cleaner); // line 60
    ...
  } ...
}

// in com.lbp.jspclasses.GetColors.java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

  DBConnection db=new DBConnection(); ...
  String bname=request.getParameter("bnamee"); ...
  bname = db.trustMe(2, bname); // line 48
  ...
}
```

Fig. 15. Snippets from classes of Snake&Ladder: they pass a servlet parameter to method `trustMe`, where however the second branch is never taken.

comes from line 165 of the same file (Fig. 16), where it is read from an external configuration file. Hence, it is considered as tainted. However, there are no other public fields in `java.awt.Color` than the color constants that the programmer wants to access here. Hence, there is no risk of unexpectedly accessing any other field here. The worst that could happen is that `colorName` is not a valid color name, but the `catch` clause will properly handle that situation. For these reasons, we consider this as a false alarm of the analyzer. In general, method `getField` allows one to read the value of any field (also private fields) of any object. Injections into that method might for instance allow access to fields holding passwords or credit card numbers. This is why it is considered as a potential injection sink. However, its use, in this example, is harmless since it is constrained to accessing the fields of `java.awt.Color` only.

7.3 Iteration Strategy

As noted at the end of Sec. 4, the iteration strategy for computing the denotational semantics does not change its fixpoint, hence does not affect its correctness or precision, but will affect its efficiency. In particular, the implementation of the fixpoint computation in Julia builds the strongly-connected components of blocks of code and analyzes them backwards, from the leaf components towards the root component. This choice follows from efficiency considerations. Fig. 17 shows that there are many strongly-connected components in the code of the applications that we have analyzed, each relatively small. By using a single huge fixpoint (column **Naive**), the full time of analysis increases by 42% on the average *w.r.t.* the use of small local fixpoints on each component (column **SCC-optimized**, this is the default in Julia). Note that, as theoretically expected, the result of the analysis does not change: always the same warnings as in Fig. 8 are generated, with both strategies.

```
// in com.jcloisterzone.ui.Client.java
private Color stringToColor(String colorName) {
  if (colorName.startsWith("#")) { // RGB format
    int r = Integer.parseInt(colorName.substring(1,3),16);
    int g = Integer.parseInt(colorName.substring(3,5),16);
    int b = Integer.parseInt(colorName.substring(5,7),16);
    return new Color(r,g,b);
  } else { // constant format
    java.lang.reflect.Field f;
    try {
      f = Color.class.getField(colorName); // line 138
      return (Color) f.get(null);
    } catch (Exception e1) {
      logger.error("Invalid color name in config file: " + colorName);
      return Color.BLACK;
    }
  }
}

// in com.jcloisterzone.ui.Client.java
public void init() { ...
  List<String> colorNames = config.get("players").getAll("color");
  playerColors = new Color[colorNames.size()];
  for (int i = 0; i < playerColors.length; i++ )
    playerColors[i] = stringToColor(colorNames.get(i)); // line 165
  ...
}
```

Fig. 16. Snippets from classes of JCloisterZone: they read color specifications from a configuration file and build `Color` objects from RGB triples or from explicit color names.

7.4 BDD Compaction

Fig. 18 shows the effect, on the full time of analysis of the open-source applications, of the BDD compaction discussed in Sec. 6. This effect is in general positive, but not always. In total, without compaction time increases by around 7%. When this is not the case (as for FarmTycoon and JExcelAPI), we suspect that, since compaction uses different mappings of variables at different program points, it increases the size for the BDD representing the U formula of Def. 5.9. It is well-known that the BDD for implication has minimal size when implying and implied variables are represented by contiguous numbers (see Andersen at al. [1], in particular compare Fig. 3 to Fig. 6 there). That is often not the case with compaction.

7.5 Scalability

The open-source applications analyzed so far (Fig. 7) are non-trivial but small enough that each single warning has been manually checked to determine if it is a true or a false alarm (Fig. 8). However, it is interesting to test the scalability

			Iteration Strategy	
Program	**Components**	**Average Size**	**SCC-optimized**	**Naive**
Snake&Ladder	79119	1.21	24	32
MediaPlayer	291759	1.61	134	183
EmergencySNRest	174314	1.31	53	78
FarmTycoon	251905	1.54	128	161
Abagail	212092	1.34	41	60
JCloisterZone	425904	1.55	199	270
JExcelAPI	301469	1.18	101	189
Colossus	703366	1.60	446	627
		Total	**1126**	**1600 (+42%)**

Fig. 17. The effect of the iteration strategy on the analysis of the open source applications. For each application, **Components** is the number of its strongly-connected components of blocks of code; **Average Size** is the average number of blocks of code in each component; **SCC-optimized** is the full time of analysis (in seconds) performing smaller fixpoints on each strongly-connected component of blocks (this is the default of the analyzer, also reported in Fig. 7); **Naive** is the full time of analysis performing a huge fixpoint over all blocks: it turns out to be 42% more than **SCC-optimized**.

Program	**Time Compacted**	**Time Uncompacted**
Snake&Ladder	24	28
MediaPlayer	134	145
EmergencySNRest	53	58
FarmTycoon	128	126
Abagail	41	75
JCloisterZone	199	230
JExcelAPI	101	97
Colossus	446	448
Total	**1126**	**1207 (+7%)**

Fig. 18. The effect of the compaction of the BDDs on the analysis of the open source applications. For each application, **Time Compacted** is the full time of analysis (in seconds) with BDD compaction (this is the default of the analyzer, also reported in Fig. 7); **Time Uncompacted** is the same time without BDD compaction: it turns out to be 7% more than **Time Compacted**.

of the analysis as well. Hence, this section analyzes three larger open-source applications, to see how the analysis scales. It only reports raw numbers of warnings for the result, since the size and the complexity of the applications makes it humanly hard to check, manually, each single warning resulting from the analysis. Fig. 19 reports these larger applications, their size and the results of the analysis. Their analyses could be completed in up to six minutes, which shows the scalability of the technique. They are discussed below.

BitcoinJ 0.14.7. This is an open-source node and wallet for the bitcoin cryptocurrency peer-to-peer network. As such, it is both a client and a server in the network. Julia issues only 7 injection warnings on this example. The complexity of the application makes it hard to understand if they are true or false alarms.

GestCV 1.0. This is an open-source curriculum management web application. It is the largest application analyzed in Tripp et al. [62] (finding 7 warnings) and one of the two largest applications analyzed in Tripp et al. [61] (finding 89 warnings, 10 of which have been checked and partitioned into true and false warnings). In these two articles, both the

Program	LoC w. Libs	Time	Addr	DOS	Eval	HS	Log	Path	Refl	Res	Sess	SQL	URL	XSS	Total
BitcoinJ 0.14.7	145980	354	3	0	0	0	0	1	1	1	0	0	1	0	7
GestCV 1.0	107494	124	0	0	0	0	5	6	20	0	2	0	2	12	47
Struts 2.5.10	104615	181	0	0	3	12	496	28	29	0	4	0	6	9	587

Fig. 19. Warnings issued by Julia for larger open-source applications. **LoC w. Libs** is the number of source lines of code, including libraries. **Time** is the full analysis time, in seconds; it includes parsing of the bytecode, supporting analyses, taint analysis and injection identification. The number of warnings is raw, that is, they are not partitioned into true and false warnings. **Addr** are untrusted data used as an IP address; **DOS** are untrusted data that can determine a denial-of-service; **Eval** are injections into an expression evaluation routine, that is, a special case of command injection; **HS** are injections into Http redirection routines (*http-splitting*); **Log** are untrusted data dumped on logs; **Path** are untrusted data flowing into file or directory names; **Refl** are untrusted data flowing into reflective methods; **Res** are untrusted data used for socket creations (*resource injection*); **Sess** are untrusted data stored into servlet sessions; **SQL** are untrusted data used for SQL queries; **URL** are untrusted data used for connecting to URLs; **XSS** are untrusted data dumped to the output of a servlet (*cross-site scripting*).

code of the application and that of the supporting libraries is analyzed, the source lines have been counted for both and warnings have been reported for both as well. Hence, the same analysis configuration is used here. GestCV includes some JSPs, that is, HTML scripts embedding some Java code, that get compiled on-the-fly by the servlet container. Julia currently does not analyze JSPs (see Sec. 11), unless they are manually compiled into Java bytecode and added to the jar to analyze. Hence the analysis of GestCV might not cover its full code and might have failed to identify some injection.

Struts 2.5.10. Struts is a largely used library for the development of web applications. Hence, bugs in that library can become bugs in the applications that use it. Since this is a library, rather than an application, we have analyzed it by assuming that every public method can be called from outside and receive untainted parameters from outside, with the exception of servlets, that receive a tainted request, with tainted request parameters. Fig. 19 shows that Julia issues some warnings in the library. Most are log injections, probably harmless, but others are potentially more serious. In particular, there are three eval-injections, where tainted data flows into an expression evaluation routine that could allow a remote code execution. These are special cases of command-injections. Sec. 7.6 describes one of these three, which turns out to be one of the most dangerous software bugs of 2017.

7.6 A Dangerous Remote Code Execution Found in Struts 2.5.10 (CVE-2017-5638)

Julia issues the following eval-injection in the code of Struts 2.5.10:

```
com/opensymphony/xwork2/util/TextParseUtil.java:166: [Injection: EvalInjectionWarning - CWE95]
dynamic evaluation of an expression from data injected into the "expression" parameter of "evaluate"
```

This is a true alarm. Actually, this is the September 2017 Equifax data breach. According to public news [40], this bug led to sensitive information exposure (customers' addresses, security and credit card numbers). NIST classifies this event as CVE-2017-5638 [39]. Actual exploitation patterns (injections) of this bug in Struts were already observed a few months before [8]. The execution path that leads to this bug is rather complex and passes through exceptions. Julia finds it and issues the warning reported above. The complete execution path is reconstructed by Gotham Digital Science [51]. We refer the interested reader to that post. Below, we only report the portions of the path that are interesting for this article.

Julia signals the bug at line 166 of `TextParseUtil.java`:

```
public static Object translateVariables(char[] openChars, String expression, ...) {
  ParsedValueEvaluator ognlEval = new ParsedValueEvaluator() { ... }
```

```
  TextParser parser = ...;
  return parser.evaluate(openChars, expression, ognlEval, maxLoopCount); // line 166
}
```

The call to `evaluate` goes into a library that evaluates an expression in an Object-Graph Navigation Language (OGNL). As reported by Gotham Digital Science [51], *many of these libraries do offer mechanisms to help mitigate remote code execution, such as sandboxing, but they tend to be disabled by default or trivial to bypass.* Hence, this is a sink for an expression evaluation injection, that allows a remote code execution. Following, backwards, the execution path that leads to method `translateVariables`, one finds that its parameter `expression` comes from a calls to method `LocalizedTextUtil.findText`, which in turn comes from class `FileUploadInterceptor.java`. In particular, at line 262 of the latter class one can read:

```
for (LocalizedMessage error : multiWrapper.getErrors()) // line 262
  if (validation != null)
    validation.addActionError(LocalizedTextUtil.findText
      (error.getClazz(), error.getTextKey(), ActionContext.getContext().getLocale(),
       error.getDefaultMessage(), error.getArgs()));
```

That is, data held in the set of errors contained in `multiWrapper.getErrors` flows into `LocalizedTextUtil.findText` and hence into line 166 of of `TextParseUtil.java`. It can be verified that such set of errors is populated at line 79 of class `JakartaMultiPartRequest.java`:

```
public void parse(HttpServletRequest request, String saveDir) ... {
  try { ...
    processUpload(request, saveDir);
  }
  catch (FileUploadException e) { ...
    LocalizedMessage errorMessage;
    if (e instanceof FileUploadBase.SizeLimitExceededException) {
      FileUploadBase.SizeLimitExceededException ex = (FileUploadBase.SizeLimitExceededException) e;
      errorMessage = buildErrorMessage(e, new Object[]{ex.getPermittedSize(), ex.getActualSize()});
    }
    else
      errorMessage = buildErrorMessage(e, new Object[]{});

    if (!errors.contains(errorMessage)) {
      errors.add(errorMessage); // line 79
    }
  } ...
}
```

Above, the set `errors` is populated with an `errorMessage` that, a few lines before, is built from data held in the exception object `e`. Hence, it is important to be aware of exceptional paths and of tainted exceptions in order to identify the

potential taintedness of the set `errors`. In particular, a static analyzer that disregards exceptional execution paths will never understand that `errors` might contain tainted data when the exception object `e` is tainted. However, this is exactly what happens here: the call to `processUpload` might raise a `FileUploadException` if the `request` is malformed. Interesting, in can be verified that the exception `e`, that it throws in that event, reports the header of the request as its message. This explains the attack patterns described in Biasini [8]: an attacker can send a malformed request, whose header contains a shell command to execute. This makes `processUpload` fail by raising a `FileUploadException e` whose message embeds the shell command. This exception is caught and collected into a set of errors (line 79 of `JakartaMultiPartRequest.java`); the elements of that set finally flow into line 166 of `TextParseUtil.java`, which evaluates an OGNL expression built from those elements and runs the shell command as part of that evaluation. By injecting a command in the header of a malformed request, the attacker can use Struts to run that command, remotely, on the attacked machine.

8 EXAMPLES OF VULNERABILITIES FOUND IN CODE FROM CUSTOMERS OF JULIASOFT SRL

This section reports a few examples of injection attacks that the implementation from Sec. 6 found in code of customers of JuliaSoft Srl. We are not allowed to disclose the name of the customers and can only show anonymized code. However, we can say that such customers are banks and insurance companies, among the largest in Italy, and that such code has been running for years inside their servers and is exposed to the external world (hence, to potential attackers). Most of the code has been bought from external providers, including well-known multinational software companies.

Differently from the examples in Sec. 7 and from the benchmarks in Sec. 9 and 10, these code snippets show relatively simple and direct injection attacks, mostly intraprocedural. This is possibly the standard programming pattern used in banking software. The reality of most software is however more complex and interprocedural and more complex examples of injections will be shown later (see for instance Sec. 7.6).

Fig. 20 is an example of code vulnerable to XSS attacks. It is so simple that it looks like the prototype example of what a programmer should *not* write in a servlet. Namely, the `SQL` parameter is fetched from the request (line 5) and sent to the output of the servlet (line 8), without any intermediate sanitization. Hence, any malicious code can be injected through the `SQL` parameter and run on the browser, which is the precise definition of an XSS attack.

```
1  public void doPost(HttpServletRequest request, HttpServletResponse response)
2      throws IOException, ServletException {
3
4    PrintWriter out = response.getWriter();
5    String sql = request.getParameter("SQL");
6    ... print HTML header to out
7    if (sql != null) {
8      out.print(sql);
9    }
10   ... print closing HTML tags
11 }
```

Fig. 20. An XSS attack is possible here.

Fig. 21 shows a snippet of code that reads the `directory` parameter of a servlet request and uses its value to access the file system, list the contents of the directory, and dump it into an HTML response. This code allows an attacker to

```
1 String directory = request.getParameter("directory");
2 ArrayList<File> listDirectories = new ArrayList<File>();
3 ArrayList<File> listFiles = new ArrayList<File>();
4
5 File dir = new File(directory);
6 File[] files = dir.listFiles();
7 if (files != null && files.length > 0)
8 ... add files to listFiles and directories to listDirectories
9 ... dump list of files and directories into an HTML response
```

Fig. 21. A path traversal attack is possible here.

```
1 String name = request.getParameter("url");
2 int length = name.length();
3 URL url = new URL(name);
4
5 if (name != null && !name.trim().equals("") && name.charAt(length - 1) == '/') {
6   URLConnection urlConn = url.openConnection();
7   ... access resource through urlConn
8 }
```

Fig. 22. A URL injection attack is possible here.

specify any file path as `directory`, access the file system at such path (as long as it is allowed by the access rights of the running application), and see its structure. Moreover, an attacker might repeatedly request access to a large directory, which can result in the slow-down of the server or even denial-of-service.

Fig. 22 reports an example of URL-injection, where the `url` parameter of a servlet is used as the name of a web resource, which is later accessed. The danger, here, is that an attacker can specify any URL, routing the connection to any third-party server. By using a collaborative server that feeds a very long stream of data, an attacker can consequently flood the attacked application and induce it into an exception or out of memory. Again, a denial-of-service can be easily triggered. As a side comment, note that variable `name` is dereferenced at line 2 before being checked for non-nullness at line 5, which sheds a troubling light on the skills of the programmer.

Fig. 23 is a snippet of code of a servlet that reads the `actions` parameter, splits it at each semicolon, and uses each resulting element as the name of a class. That class gets resolved (line 7, which implicitly executes the class's static initialization method), instantiated through its default constructor (line 8), and used as a recipe for building the response of the servlet: content type (line 9) and header (line 10). A reflection injection attack is possible at lines 7 and 8, since an arbitrary class name can be specified as parameter, inducing the application to resolve that class. An attacker might cleverly specify a class whose static initializer allocates a pool of objects, never released later. If, instead, the attacker specifies the name of an unknown class, an exception is thrown at line 7, possibly leaving the application in an inconsistent state. Moreover, the header of the response is built from the request (line 10), using all parameters of the request, that can be freely specified by an attacker. This allows an attacker to inject any header initialization parameter in the response, not just the `Content-Disposition` header parameter, since clever use of newline characters allows one to

```
1  String par = request.getParameter("actions");
2  if (par != null && !par.equalsIgnoreCase("")) {
3    StringTokenizer st = new StringTokenizer(par, ";");
4    while (st.hasMoreElements()) {
5      par = (String) st.nextElement();
6      if (par != null) {
7        Class cls = Class.forName(par);
8        MyListener al = (MyListener) cls.newInstance();
9        response.setContentType(this.getContentType(al));
10       response.setHeader("Content-Disposition", this.getHeader(al, request));
11       ...
12     }
13   }
14 }
```

Fig. 23. A reflection injection attack and an HTTP response splitting attack are possible here.

Test Collection	Attack	LoC
CWE80	XSS	43619
CWE81	XSS	21720
CWE83	XSS	21720
CWE89	SQL	399451

Fig. 24. Test cases from the Juliet Suite for Java, version 1.2. LoC is their number of non-blank non-comment application source lines of code. Classes from the standard Java library are not counted but are included in the analysis.

forge extra parameter names. This attack, called HTTP response splitting, is known to be able to poison the cache of any proxy server that users need to access the application. As side comments, note that it would have been better to call `nextToken` at line 5, hence avoiding the cast to `String`, and that the test at line 6 is useless since string tokenizers never return `null` elements. Moreover, variable `par` first holds the whole string (line 1) and later holds one of its parts (line 5), definitely a bad programming style.

9 EXPERIMENTS ON THE JULIET SUITE

Experiments on standard test suites are important since they enable comparison with other technology. The National Institute of Standards and Technology (NIST) publishes the Juliet Test Suite for Java [37]. Its version 1.2 includes test cases for 112 different common software weaknesses (identified by *CWE* id). Fig. 24 shows its test collections for XSS and SQL injection attacks. Each test collection contains thousands of servlets that exercise different attack patterns. The first three categories contain examples of XSS attacks, distinguished *w.r.t.* the kind of disruption incurred by the attacked system. The fourth category includes examples of SQL injection attacks. Although the tests are numerous, they are not very diverse: they belong to 37 data propagation patterns, cyclically repeated for distinct sources and sinks. We ran tools on each of CWE80/81/83, considered as three big programs, instead of analyzing each servlet in isolation.

Test	Tool	True Positives	False Positives	False Negatives	Analysis Time (minutes)
CWE80	CodePro Analytix	180	0	486	9
	FindBugs	19	0	647	<1
	Fortify SCA	282	0	384	590
	Julia	666	0	0	2
	Julia field-insensitive	666	0	0	2
CWE81	CodePro Analytix	0	0	333	<1
	FindBugs	19	0	314	<1
	Fortify SCA	141	0	192	303
	Julia	333	0	0	2
	Julia field-insensitive	333	0	0	2
CWE83	CodePro Analytix	90	0	243	5
	FindBugs	19	0	314	<1
	Fortify SCA	141	0	192	296
	Julia	333	0	0	2
	Julia field-insensitive	333	0	0	2

Fig. 25. Experiments with the identification of XSS attacks in the Juliet Suite. Times include all supporting analyses.

We ran five tools that identify injections: those from Sec. 3 that we could run for free and that provide some form of static analysis for the identification of XSS attacks or SQL injections; Julia; and a field-insensitive variant of Julia that locks its field-sensitive analysis (Sec. 5.6).

Fig. 25 and 26 report the results for identification of XSS and SQL injection vulnerabilities, respectively. True positives are real vulnerabilities that the tool finds, false positives are tool reports that do not correspond to any actual threat, and false negatives are real vulnerabilities that the tool does not find.

In each case, Julia is the only sound tool. (Fig. 34 shows comparisons against more tools: Julia remains the only sound tool and outperforms other tools.) Julia is perfectly precise (it issued no false positive warnings) because these tests are large but relatively simple. They just propagate information, without side-effects that might degrade the precision of Julia. (Side effects are considered in Def. 5.16; we do not know if and how other tools deal with side effects.) Therefore, the Juliet suite demonstrates coverage of sources and sinks but does not evaluate the accuracy of the analysis tool *w.r.t.* the propagation of information.

Another indication of the simplicity of the Juliet test suite is that the field-insensitive version of Julia, which does not use the tainted field set from Sec. 5.6, is just as accurate as the default field-sensitive version. This shows that data does not flow in complex ways through fields. Moreover, this suggests that field-sensitivity might not be relevant when object sensitivity is used to distinguish different objects, which is always the case in our analysis. The more realistic experiments of Sec. 10 demonstrate the need for field-sensitivity.

Analysis time indicates the efficiency of the tools, but only roughly: CodePro Analytix and FindBugs work on the client machine inside Eclipse, and Fortify SCA and Julia run in the cloud (controlled by a local IDE). We suspect that Fortify uses some form of queue of tasks; hence, its actual running time is unknown to us.

10 EXPERIMENTS ON THE OWASP CYBERSECURITY BENCHMARK

The OWASP Benchmark Project is the most realistic comprehensive cybersecurity benchmark that we know of. Its version 1.2 is a suite of 2740 small Java programs (servlets) containing security threats. In total, the benchmark contains 141050 non-blank, non-comment lines of application source code, without libraries. According to its web page [41]:

Test	Tool	True Positives	False Positives	False Negatives	Analysis Time (minutes)
CWE89	CodePro Analytix	1332	0	1046	20
	FindBugs	1776	2400	602	2
	Fortify SCA	700	0	1678	3600
	Julia	2378	0	0	24
	Julia field-insensitive	2378	0	0	15

Fig. 26. Experiments with the identification of SQL injection attacks in the Juliet Suite. Times include all supporting analyses.

```
1  public void doPost(HttpServletRequest request, HttpServletResponse response) throws ... {
2     response.setContentType("text/html;charset=UTF-8");
3     String param = "";
4     if (request.getHeader("Referer") != null)
5        param = request.getHeader("Referer");
6     param = java.net.URLDecoder.decode(param, "UTF-8");
7     String bar = param;
8     if (param != null && param.length() > 1) {
9        StringBuilder sbxyz67327 = new StringBuilder(param);
10       bar = sbxyz67327.replace(param.length()-"Z".length(), param.length(),"Z").toString();
11    }
12    response.setHeader("X-XSS-Protection", "0");
13    Object[] obj = { "a", "b" };
14    response.getWriter().format(java.util.Locale.US, bar, obj);
15 }
```

Fig. 27. Simplified OWASP Benchmark 146: This test suffers from a real XSS attack and Julia spots it.

The OWASP Benchmark for Security Automation is a free and open test suite designed to evaluate the speed, coverage, and accuracy of automated software vulnerability detection tools and services. Without the ability to measure these tools, it is difficult to understand their strengths and weaknesses, and compare them to each other.

The benchmark has benefited from the contributions of many organizations, so it also categorizes real threats. It is considered *the* standard benchmark for the comparison of security analysis tools. Most tests in the OWASP Benchmark are servlets that allow unconstrained information flow from their inputs to dangerous routines. A few tests are not related to injections, but rather to unsafe cookie exchange or to the use of inadequate cryptographic algorithms, hash functions, or random number generators. The benchmark sets traps for tools, *i.e.*, it also contains harmless servlets that *seem* to feature security threats, at least at a superficial analysis. An ideal tool would achieve 100% true positives (that is, real vulnerabilities reported by the tool) and 0% false positives (that is, vulnerabilities reported by the tool that are not real issues). However, existing tools make a compromise between soundness, precision, and efficiency of the analysis.

This benchmark comes with scripts that generate comparative *scorecards* that plot coverage and accuracy of the tools [41]. Commercial tools are anonymized in the scorecards. The scorecards in Fig. 32, 33 and 31 were automatically generated by OWASP benchmark scripts.

Julia has been run on the OWASP Benchmark considered as one big program, hence as a single test case, instead of analyzing each servlet in isolation. During its analysis, Julia processes 141050 non-blank non-commented lines of code

```
1  public void doPost(HttpServletRequest request, HttpServletResponse response) throws ... {
2     response.setContentType("text/html;charset=UTF-8");
3     String param = "";
4     java.util.Enumeration<String> headers = request.getHeaders("Referer");
5     if (headers != null && headers.hasMoreElements())
6        param = headers.nextElement();
7     param = java.net.URLDecoder.decode(param, "UTF-8");
8     String bar = org.springframework.web.util.HtmlUtils.htmlEscape(param);
9     response.setHeader("X-XSS-Protection", "0");
10    response.getWriter().print(bar);
11 }
```

Fig. 28. Simplified OWASP Benchmark 278: No XSS attack is possible here and Julia does not issue any warning.

from the benchmark. Including the libraries reached and analyzed during the analysis, Julia processes 215533 non-blank non-commented lines of code. Below, we present a true positive, a true negative, and a false positive produced by Julia. Then, Sec. 10.1 presents the overall results of Julia on the OWASP Benchmark.

True positive: Fig. 27 shows a (simplified) OWASP Benchmark test. Julia warns about a possible XSS attack at line 14, since the `bar` parameter to method `format` is tainted. In fact, this parameter is built from the content of local variable `param` (line 9), that has been assigned (line 5) an input that the user can control (the header of the connection). Julia correctly spots the information flow through the constructor of `StringBuilder` at line 9 and the call to `replace` at line 10.

True negative: Julia does not issue any warning for the test in Fig. 28. Actually, no XSS attack is possible, since variable `param` (tainted at line 6) is sanitized into `bar` by Spring method `htmlEscape` (line 8), which appears in Julia's dictionary of sanitizing methods (Sec. 5.5).

False positive: Julia falls into Fig. 29's trap and issues a spurious warning about a potential XSS attack at the call to `format`, since Julia thinks that `bar` (and hence `obj`) is tainted. But this is not actually the case, since this test manipulates a `valueList` in such a way that, although the list does contain a tainted element, the value finally stored into `bar` is untainted. This list manipulation is too complex for the taint analysis of Julia, that cannot distinguish each single element of the list and conservatively assumes all elements of the list to be tainted.

False positive only in the field-insensitive analysis: The field-insensitive analysi is slightly less precise than Julia's default field-sensitive analysis (Fig. 31). Fig. 31 and 33 show that a notable example is testing for path traversal attacks. Fig. 30 is an example where the field-insensitive analysis underperforms. The field-sensitive analyzer correctly infers that variable `fileName` holds untainted data, since it is composed by prefixing the untainted variable `bar` with the untainted static field `testfileDir`. By contrast, the field-insensitive analyzer imprecisely infers `testfileDir`, and hence variable `fileName`, to be potentially tainted. This is because a field-insensitive analysis cannot distinguish `testfileDir` from the other static fields. If `testfileDir` were an instance field, then a reachability-based analysis like ours could still prove it untainted by proving that its holder is untainted (see the first case of Eq. 1 in Sec. 5.6). That means that the field-insensitive analysis is very imprecise *w.r.t.* the use of static fields. The pattern shown in Fig. 30 is repeated in most test cases for path traversal in the OWASP Benchmark, which explains the bad score in Fig. 31 with the field-insensitive

```
1  public void doPost(HttpServletRequest request, HttpServletResponse response) throws ... {
2     response.setContentType("text/html;charset=UTF-8");
3     String param = "";
4     if (request.getHeader("Referer") != null)
5        param = request.getHeader("Referer");
6     param = java.net.URLDecoder.decode(param, "UTF-8");
7     String bar = "alsosafe";
8     if (param != null) {
9        java.util.List<String> valuesList = new java.util.ArrayList<String>();
10       valuesList.add("safe");
11       valuesList.add(param);
12       valuesList.add("moresafe");
13       valuesList.remove(0); // remove the 1st safe value
14       bar = valuesList.get(1); // get the last 'safe' value
15    }
16    response.setHeader("X-XSS-Protection", "0");
17    Object[] obj = { "a", bar };
18    response.getWriter().format("Formatted like: %1$s and %2$s.", obj);
19 }
```

Fig. 29. Simplified OWASP Benchmark 147: No XSS attack is possible here, but Julia issues a false alarm.

```
1  public void doPost(HttpServletRequest request, HttpServletResponse response) throws ... {
2    response.setContentType("text/html;charset=UTF-8");
3    ...
4    String bar = ... untainted data
5
6    String fileName = org.owasp.benchmark.helpers.Utils.testfileDir + bar;
7    FileInputStream fis = new java.io.FileInputStream(new java.io.File(fileName));
8    ...
9  }
10
11 // inside org.owasp.benchmark.helpers.Utils.java:
12 public static final String testfileDir = System.getProperty("user.dir") + File.separator
13   + "testfiles" + File.separator;
```

Fig. 30. Simplified OWASP Benchmark 63: No path traversal attack is possible here, but the field-insensitive version of our analysis issues a false alarm at the file creation at line 7. The static field `testfileDir` is untainted, since system properties are never set in the program to tainted values. However, specific information on a given static field can only be expressed with a field-sensitive analysis.

analysis. Note that this imprecision of *fi* is not specific to path traversal in any way, but is a consequence of the way the OWASP test cases for path traversal are written.

	Julia			field-insensitive Julia		
Category	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**
Command Injection	126	20	0	126	20	0
XSS	246	21	0	246	21	0
Insecure Cookie	36	0	0	36	0	0
LDAP Injection	27	4	0	27	4	0
Path Traversal	133	22	0	133	119	0
SQL Injection	272	36	0	272	36	0
Trust Boundary Violation	83	12	0	83	12	0
Weak Encryption Algorithm	130	0	0	130	0	0
Weak Hash Algorithm	129	0	0	129	0	0
Weak Random Number	218	0	0	218	0	0
XPath Injection	15	2	0	15	2	0
Total	1415 (*92%*)	117 (*8%*)	0	1415 (*87%*)	214 (*13%*)	0

Fig. 31. Results of the analysis of the OWASP Benchmark test suite with Julia 2.3.4 with its standard (field-sensitive) taint analysis and with its field-insensitive version. **TP** are true positives, **FP** are false positives, and **FN** are false negatives. Boldface categories are instances of injection attacks. (Julia's success in the other categories depends on different techniques, unrelated to this article.) The *Total* row reports the percentage of warnings in each category. Namely, Julia issues 1532 warnings, 1415 of which are true positives, *i.e.*, 92%. Field-insensitive Julia issues 1629 warnings, the same 1415 of which are true positives, *i.e.*, 87%.

10.1 Scorecards for Julia

Fig. 31 reports the results of Julia's analysis of the OWASP Benchmark, using its standard (field-sensitive) taint analysis (Sec. 5.6) and a less precise field-insensitive taint analysis. This table (automatically computed by OWASP) shows again that Julia is sound (there are no false negatives: the analyzer finds all threats). There are false alarms, but they make up a manageable 8% and 13% of the warnings, respectively.

OWASP uses the numbers in Fig. 31 to generate the graphical scorecards in Fig. 32 and 33. A theoretical, perfect static analyzer should lie at the top left corner of the scorecards, and analyzers that are closer to that corner are better. In the scorecards, the *true positive rate* is the number of true positives found by the tool, over the total number of true positives in the benchmark; the *false positive rate* is the number of false positive *traps* erroneously considered by the analyzer as a threat, over the total number of traps. Julia is sound (it finds all true positives), so it appears on the 100% y coordinate. The horizontal distance from the y axis represents the precision of the tool (false positive rate, *i.e.*, number of false alarms).

10.2 Comparison with Other Static Analyzers

The OWASP Benchmark enables comparison of distinct static analyzers. Its distribution already includes the results of the run of some free and commercial analyzers. The six commercial tools Checkmarx CxSAST, Coverity Code Advisor, HP Fortify, IBM AppScan, Parasoft Jtest, and Veracode SAST are reported in the scorecard as SAST-01$\cdots$SAST-06, but we do not know which is which. This is because the license agreements of commercial companies (other than JuliaSoft Srl) forbid divulging analysis results. This is a significant limitation for scientific research, but OWASP circumvents it by publishing anonymized data for commercial analyzers.

Fig. 34 reports the global scorecard, with the scores of all (free and commercial) analyzers. It also reports the average of the commercial analyzers (besides Julia), in order to highlight how much better are Julia's results *w.r.t.* those obtained with competing commercial tools (Julia is point L; Julia with with a field-insensitive taint analysis is point K). Julia is

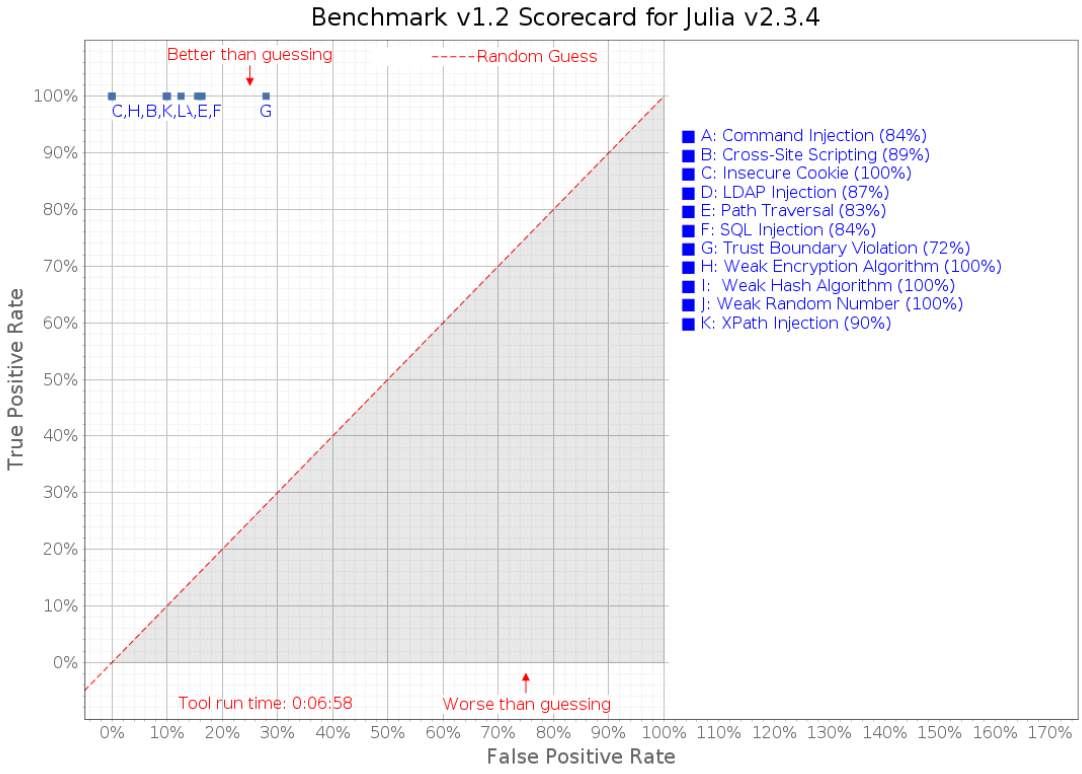

Fig. 32. The OWASP scorecards for Julia version 2.3.4. The percentage reported by OWASP on the right of each category of threat is the difference between the true positive rate of the analyzer and the false positive rate of the analyzer, for that category. This is different from the percentages reported in Fig. 31, which are ratios over the warnings issued by Julia, not over the total number of test cases in the benchmark.

the only sound analyzer in this comparison, *i.e.*, the only analyzer that lies on the 100% y coordinate. Julia achieves this precision in acceptable run time (a few minutes, including all supporting analyses: see Fig. 32). This time is comparable to that of other free analyzers [41]. OWASP does not divulge the run times for the commercial analyzers.[3]

10.3 How to Run Julia on the OWASP Benchmark

The experiments in this section about the OWASP Benchmark are completely reproducible via the following steps.

(0) Download the OWASP Benchmark, run `mvn compile` and `mvn eclipse:eclipse` (where `mvn` is Maven 3), and ensure that the benchmark compiles within Eclipse. Please refer to the OWASP Benchmark page [41] for further information on compiling the benchmark.

(1) Register for Julia's online analysis service at https://portal.juliasoft.com.

[3]We have sent the results of this section to OWASP. Consequently, in the OWASP Benchmark page [41], Julia is already listed as one of the tools that have been used to analyze the OWASP Benchmark test suite. However, as of this writing, results for Julia, such as the scorecard in Fig. 34, are not yet reported there. According to the maintainer of the benchmark, this is going to happen soon. Please contact dave.wichers@owasp.org for further detail.

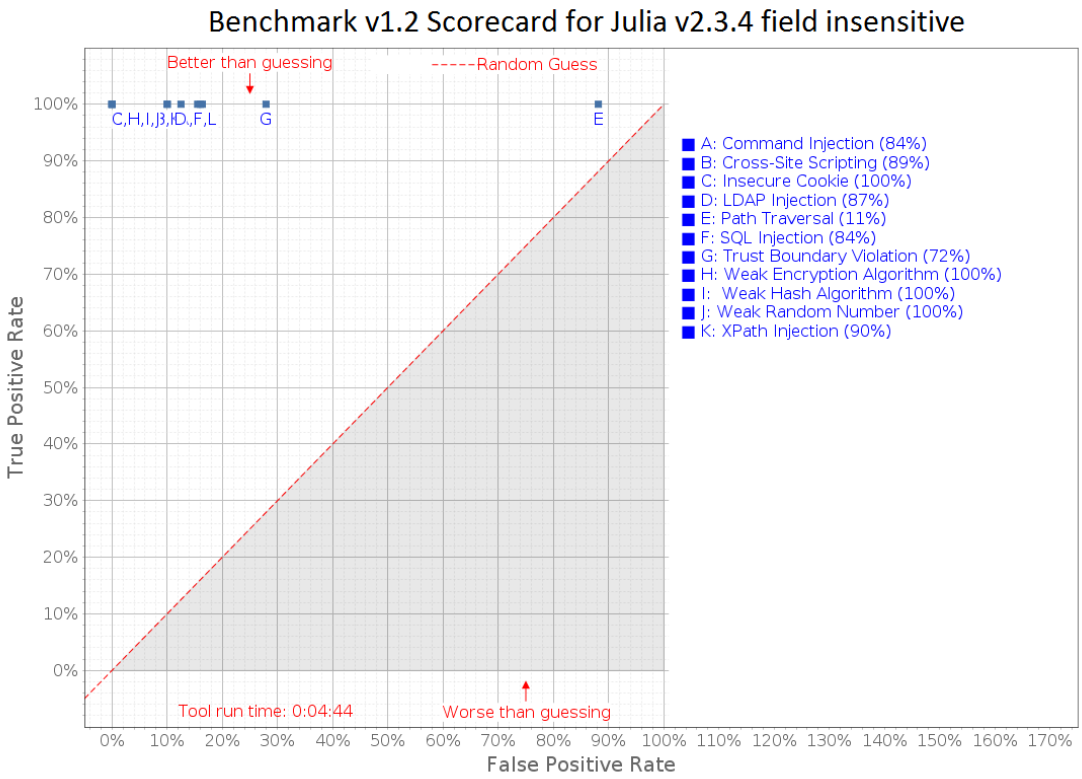

Fig. 33. The OWASP scorecards for Julia version 2.3.4, with a field-insensitive taint analysis. The percentage reported by OWASP on the right of each category of threat is the difference between the true positive rate of the analyzer and the false positive rate of the analyzer, for that category. This is different from the percentages reported in Fig. 31, which are ratios over the warnings issued by Julia, not over the total number of test cases in the benchmark.

(2) Install Julia's Eclipse plugin and configure it with your credential information. Instructions and credentials are available at login time to Julia's service and also under your user profile.
(3) Increase the maximum number of warnings shown to 5000.
(4) Make sure you have enough credit to run the analysis (150000 credits are needed); otherwise, please contact JuliaSoft Srl to get more free credits.
(5) Select the OWASP Benchmark project in Eclipse and analyze it with Julia's Eclipse plugin: in the first screen, flag both options *Only main* and *Include .properties files*. The latter is needed since the benchmark assumes that the analyzer has access to property files, which is normally turned off for privacy.
(6) In the next screen of Julia's Eclipse plugin, select only the Basic checkers Cryptography, Cookie, and Random and the Advanced checker Injection; for field-insensitive analysis, flag the noOracle option of the latter checker.
(7) Click Finish and wait until the analysis terminates. This should take a few minutes, unless there are other analyses in the queue. You can check the progress of the analysis in the console view of Eclipse.

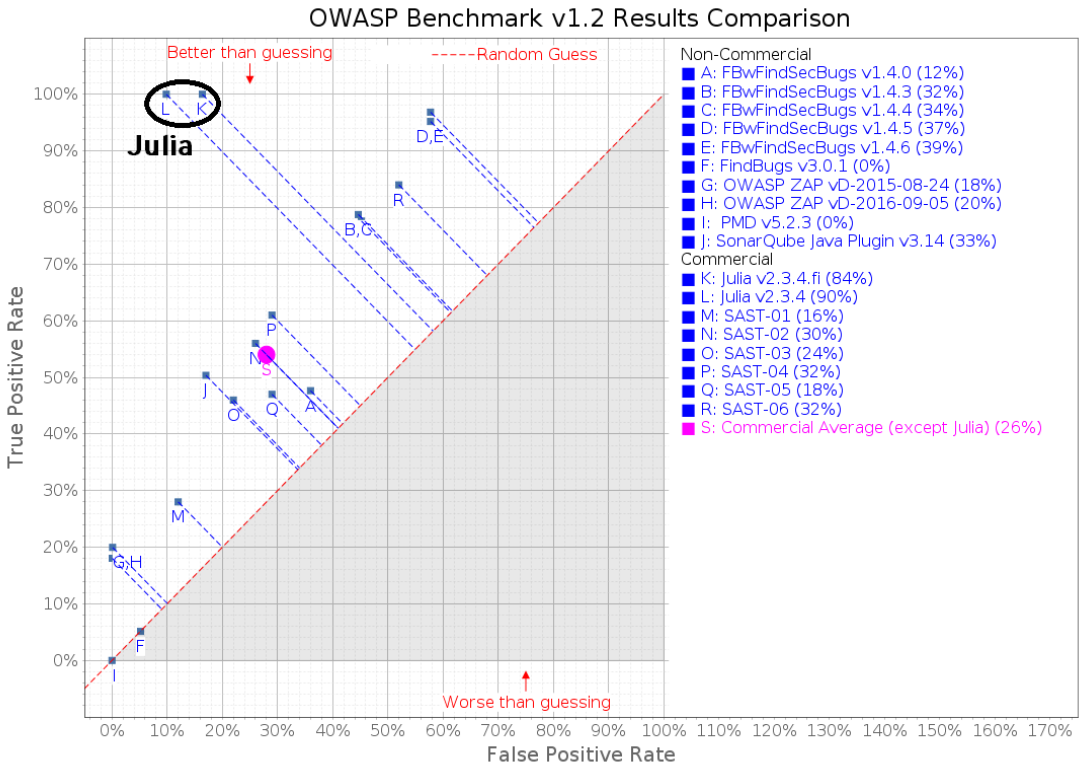

Fig. 34. The OWASP scorecard collecting the average scores for all static analyzers, free and commercial. Julia with its default configuration is point L. The field-insensitive (*fi* in the legend) version of Julia is point K.

(8) Once the analysis has terminated, you can see the warnings in the Eclipse view of Julia and export the XML file of the warnings with the icon of that view that looks like a downwards arrow.

The result of these steps is an XML file with results of Julia's analysis of all the test cases of the OWASP Benchmark, including all warnings generated by Julia. The XML created by Julia can be used to generate the scorecards for the OWASP Benchmark, by following the instructions at their site [41]. The company website http://www.juliasoft.com has user manuals and short tutorials about the Eclipse plugin and the web interface of Julia.

11 LIMITATIONS OF THE ANALYSIS

This section explains limitations of our analysis.

Our analysis does not distinguish elements of collections. The analysis can infer that a variable is bound to a list that contains tainted data at some position, but cannot distinguish the taintedness of each single list element (see example in Fig. 29). The same holds for arrays, which are considered tainted as soon as at least an element is tainted. This choice is sound but can cause false alarms.

As noted in Sec. 1, the analysis is inherently limited to explicit flows. It does not model implicit flows nor hidden channels, such as timing channels or energy consumption.

This work is currently limited to Java. For multi-lingual applications, including native methods and JSP or similar non-Java technology, the analysis might be unsound, since only part of the code is analyzed. Class-loaders can load arbitrary code that our analysis would miss analyzing. Our analysis makes a worst-case assumption about the return value of native or reflective calls: their return value can be tainted only if at least one of their arguments is tainted. However, this model is incomplete (and therefore unsound) since such calls might read tainted static fields and might also have arbitrary effects on the heap. A few, frequently used native methods with heap effects have been modeled explicitly in the analysis (for instance, `System.arraycopy()`). A full and sound treatment of reflection is a notable hard problem [28], though sound approaches exist that perform well in common practice [5, 63].

The analysis can be unsound for multithreaded programs, since a thread might asynchronously taint a field or array, which is not modeled by our sequential analysis.

Our analysis is defined at the Java bytecode level and can be applied to binaries created by any programming language, such as Java, Scala, Clojure, Kotlin, and Groovy. The results for non-Java languages are currently worse, for three reasons. (1) These languages include runtime and instrumentation code in the `.class` files. Julia has no hint about the fact that that code is instrumented: any warning issued there will not be understood by the programmer, who will see warnings at unknown program locations, not present in the code that he wrote. (2) Julia does not have a model of the language runtime's behavior; for example, it conservatively assumes the existence of side effects that never happen at run time. (3) Julia does not contain a a list of sources and sinks for non-Java languages. It would be straightforward to extend Julia to other languages.

12 CONCLUSION

We have formalized an object-sensitive notion of taintedness that can be applied to reference types. We have built a new, object-, flow-, context-, and field-sensitive static taint analysis based on this notion, proved it sound, implemented it, and demonstrated that it provides much more precise results than other analyzers, in acceptable time, and scales to large Java applications. As far as we know, this is the first formal definition of an object-sensitive taint analysis. The resulting analysis is sound, by contrast with other analyzers (in Fig. 34, only Julia is on the 100% y-coordinate). As with any static analysis, soundness is jeopardized by reflection, concurrency, or non-standard class loaders. However, our soundness claim is still relevant since it gives confidence in the results, up to the use of those features. Julia deals with the full bytecode generated by Java 8, including the `invokedynamic` instruction used to implement lambda expressions. For the latter, Julia uses a technique known as *desugaring*, *i.e.*, it creates synthetic bridge classes that implement the functional interface by calling the lambda's body. Such classes have fields for the variables captured by the closure. This is exactly the same technique used, for instance, in Android Studio 3.0 (see https://developer.android.com/studio/write/java8-support.html, which aims to run `invokedynamic` on Android bytecode, which has no such instruction). After that desugaring, the `invokedynamic` instruction disappears and is replaced by a `call` bytecode to the bridge. The resulting code, without `invokedynamic`, is then analyzed.

The analysis is already able to analyze Android code, as long as the latter is first translated from Dalvik bytecode into Java bytecode (there are free tools for that). Julia has a plugin for Android Studio, where this process is automated. Android is typically programmed in Java, with a different runtime. Julia desugars non-programmatic parts of the applications, provided in Android as XML files, and understands the lifecycle of Android components. The description of the analysis of Android applications is out of the scope of this article, which explicitly concentrates on Java only. The

interested reader can find the technical description of the Android support in Payet and Spoto [45]. Actual experiments of injection analysis of Android automotive apps are reported in Panarotto et al. [43].

The novelty of the approach stems from defining a property of reference types as a reachability property (Def. 5.1), whose relevance goes beyond the case of taint analysis. Here, we mean reachability of data from a memory reference, which is not reachability of abstract states through execution paths as in Reps et al. [48] and in Reps [47]. Def. 5.1 results in an object-sensitive analysis: the taintedness of an object determines that of its fields; a drawback is that the analysis, to remain sound, must consider side-effects at `putfield` and `call` instructions (see the proofs of Prop. 5.13 and 5.17). The analysis then becomes field-sensitive through an oracle-based approach (Sec. 5.6), just as previously done for nullness analysis [57]. The oracle is a general technique for building sound field-sensitive static analyses.

A number of extensions are possible. (1) Extending 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 an analysis, to be sound *w.r.t.* implicit flows, would end up being highly conservative and imprecise. Declassification might be helpful here, but its meaning for reference types (not just primitive values) must be studied. (2) Extending this work to the analysis of JSP would avoid missed alarms in hybrid web applications. JSP are non-Java code mixed and interacting with Java code, currently not analyzed by Julia (only partially by other current tools). (3) Explaining warnings to the users, with an execution trace where data flows from sources into sinks. Fortify SCA already provides some support in that direction. The technique described in this article is currently not satisfactory from this point of view, since the notion of taintedness is represented as a single (Boolean) token; hence, it is impossible to distinguish different sources of taintedness. In recent work, we have shown how the results of our taint analysis can be exploited to reconstruct, backwards, some actual execution paths that explain the injection [15].

Experiments with the NIST Juliet Suite and the OWASP Benchmark show the importance of objective, third-party benchmarks to compare different static analyzers. All commercial static analyzers claim to cope with most injection attacks, at least XSS and SQL injections. License agreements with secrecy provisions hinder scientific research that wishes to compare sophisticated techniques like the ones in this paper against the simpler, faster, imprecise analyses implemented in most commercial tools. Nonetheless, results on these benchmarks show that our approach is sound (unlike all other tools), more precise than other tools, and fast.

REFERENCES

[1] H. R. Andersen. An Introduction to Binary Decision Diagrams. Available at http://configit.com/configit_wordpress/wp-content/uploads/2013/07/bdd-eap.pdf, 1999.

[2] 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.

[3] 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.

[4] P. Avgustinov, O. de Moor, M. Peyton Jones, and M. Schäfer. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP'16)*, volume 56 of *LIPIcs*, pages 2:1–2:25, Rome, Italy, July 2016. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[5] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 669–679, Lincoln, NE, USA, November 2015.

[6] 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.

[7] G. Barthe, T. Rezk, and A. Basu. Security Types Preserving Compilation. *Computer Languages, Systems & Structures*, 33(2):35–59, 2007.

[8] N. Biasini. Content-Type: Malicious – New Apache Struts2 0-day under Attack. https://blog.talosintelligence.com/2017/03/apache-0-day-exploited.html, March 2017.
[9] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.
[10] E. Burato, P. Ferrara, and F. Spoto. Security Analysis of the OWASP Benchmark with Julia. In *First Italian Conference on Security (ITASEC)*, Venice, Italy, January 2017.
[11] D. Clark, C. Hankin, and S. Hunt. Information Flow for ALGOL-like Languages. *Computer Languages*, 28(1):3–28, April 2002.
[12] 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.
[13] 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.
[14] M. D. Ernst, A. Lovato, D. Macedonio, C. Spiridon, and F. Spoto. Boolean Formulas for the Static Identification of Injection Attacks in Java. In *Proc. of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'20)*, volume 9450 of *Lecture Notes in Computer Science*, pages 130–145, Suva, Fiji, November 2015. Springer.
[15] P. Ferrara, L. Olivieri, and F. Spoto. Tailoring Taint Analysis to GDPR. In *Annual Privacy Forum, Revised Selected Papers*, volume 11079 of *Lecture Notes in Computer Science*, pages 63–76, Barcelona, Spain, June 2018. Springer.
[16] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A Static Analysis Framework for Detecting SQL Injection Vulnerabilities. In *31st Annual International Computer Software and Applications Conference, COMPSAC*, volume 1, pages 87–96, Beijing, China, 2007.
[17] S. Genaim, R. Giacobazzi, and I. Mastroeni. Modeling Secure Information Flow with Boolean Functions. In Peter Ryan, editor, *WITS'04*, April 2004.
[18] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In R. Cousot, editor, *VMCAI*, pages 346–362, Paris, France, 2005. Springer-Verlag.
[19] S. Genaim and F. Spoto. Constancy Analysis. In M. Huisman, editor, *FTfJP*, Paphos, Cyprus, July 2008. Radboud University.
[20] J. Hélie, I. Wright, and A. Ziegler. Measuring Software Development Productivity: A Machine Learning Approach. In *Proc. of the Machine Learning for Programming Workshop, affiliated with FLoC'18*, Oxford, UK, July 2018.
[21] Oracle Inc. Java Platform, Enterprise Edition. http://www.oracle.com/technetwork/java/javaee/overview/index.html. Last checked on December 26, 2017.
[22] Oracle Inc. JavaServer Pages Technology. http://www.oracle.com/technetwork/java/javaee/jsp/index.html. Last checked on December 26, 2017.
[23] Pivotal Software Inc. Spring. https://spring.io. Last checked on December 26, 2017.
[24] Y.-S. Jang and J.-Y. Choi. Detecting SQL Injection Attacks using Query Result Size. *Computers & Security*, 44:104–118, 2014.
[25] D. Kar, S. Panigrahi, and S. Sundararajan. SQLiGoT: Detecting SQL Injection Attacks Using Graph of Tokens and SVM. *Computers & Security*, 60:206–225, 2016.
[26] N. Kobayashi and K. Shirane. Type-based Information Flow Analysis for Low-Level Languages. In *APLAS*, 2002.
[27] D. G. Kumar and M. Chatterjee. MAC based Solution for SQL Injection. *Journal of Computer Virology and Hacking Techniques*, 11(1):1–7, 2015.
[28] D. Landman, A. Serebrenik, and J. J. Vinju. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In *Proc. of the International Conference on Software Engineering (ICSE'17)*, pages 507–518, Buenos Aires, Argentina, May 2017.
[29] P. Laud. Semantics and Program Analysis of Computationally Secure Information Flow. In *ESOP*, pages 77–91. Springer-Verlag, 2001.
[30] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.
[31] 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.
[32] L. Liu, J. Xu, H. Yang, C. Guo, J. Kang, S. Xu, B. Zhang, and G. Si. An Effective Penetration Test Approach Based on Feature Matrix for Exposing SQL Injection Vulnerability. In *40th IEEE Annual Computer Software and Applications Conference, COMPSAC*, pages 123–132, Atlanta, GA, USA, 2016.
[33] 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.
[34] MITRE/SANS. Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25, September 2011. Last checked on December 26, 2017.
[35] M. Mizuno. A Least Fixed Point Approach to Inter-Procedural Information Flow Control. In *NCSC*, pages 558–570, 1989.
[36] N. M. Naghmeh Moradpoor Sheykhkanloo. Employing Neural Networks for the Detection of SQL Injection Attack. In *SIN*, page 318, Glasgow, Scotland, UK, 2014.
[37] National Institute of Standards and Technology. Juliet Test Suite for Java. https://samate.nist.gov/SRD/testsuite.php. Last checked on December 26, 2017.
[38] Đ. Nikolić and F. Spoto. Reachability Analysis of Program Variables. *ACM Transactions on Programming Languages and Systems*, 35(4):14, 2013.
[39] NIST. CVE-2017-5638 Detail. https://nvd.nist.gov/vuln/detail/CVE-2017-5638, September 2017.
[40] S. A. O'Brien. Giant Equifax Data Breach: 143 Million People Could Be Affected. http://money.cnn.com/2017/09/07/technology/business/equifax-data-breach/index.html, September 2017.
[41] OWASP. Benchmark. https://www.owasp.org/index.php/Benchmark. Last checked on December 26, 2017.
[42] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proc. of the 6th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 146–161, Phoenix, Arizona, USA, 1991.

[43] F. Panarotto, A. Cortesi, P. Ferrara, A. Mandal, and Spoto F. Static Analysis of Android Apps Interaction with Automotive CAN. In M. Qiu, editor, *SmartCom*, volume 11344 of *Lecture Notes in Computer Science*, pages 114–123, Tokyo, Japan, December 2018. Springer.

[44] É. Payet and F. Spoto. Magic-Sets Transformation for the Analysis of Java Bytecode. In *SAS*, pages 452–467. Springer, 2007.

[45] É. Payet and F. Spoto. Static Analysis of Android Programs. *Information & Software Technology*, 54(11):1192–1201, 2012.

[46] T. F. A. Rahman, A. G. Buja, K. A. Jalil, and F. M. Ali. SQL Injection Attack Scanner Using Boyer-Moore String Matching Algorithm. *JCP*, 12(2):183–189, 2017.

[47] T. W. Reps. Program Analysis via Graph Reachability. *Information & Software Technology*, 40(11-12):701–726, 1998.

[48] T. W. Reps, S. Horwitz, and S. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL'95*, pages 49–61, San Francisco, California, USA, January 1995.

[49] A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[50] 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.

[51] Gotham Digital Science. An Analysis of CVE-2017-5638. https://blog.gdssecurity.com/labs/2017/3/27/an-analysis-of-cve-2017-5638.html, March 2017.

[52] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *SAS*, pages 320–335. Springer, 2005.

[53] H. Shahriar and M. Zulkernine. Information-Theoretic Detection of SQL Injection Attacks. In *HASE*, pages 40–47, Omaha, NE, USA, 2012.

[54] L. K. Shar and K. Tan, H. B. Defeating SQL Injection. *IEEE Computer*, 46(3):69–77, 2013.

[55] B. Simic and J. Walden. Eliminating SQL Injection and Cross Site Scripting using Aspect Oriented Programming. In *ESSoS*, pages 213–228, Paris, France, 2013.

[56] C. Skalka and S. Smith. Static Enforcement of Security with Types. In *ICFP*, pages 254–267. ACM press, 2000.

[57] F. Spoto. Nullness Analysis in Boolean Form. In *SEFM*, pages 21–30, Washington, DC, USA, 2008. IEEE.

[58] F. Spoto. The Julia Static Analyzer for Java. In X. Rival, editor, *Static Analysis Symposium (SAS)*, volume 9837 of *Lecture Notes in Computer Science*, pages 39–57, Edinburgh, UK, 2016. Springer.

[59] M. Stampar. Inferential SQL Injection Attacks. *I. J. Network Security*, 18(2):316–325, 2016.

[60] F. Tip and J. Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. In *Proc. of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, pages 281–293, Minneapolis, Minnesota, USA, October 2000.

[61] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *Fundamental Approaches to Software Engineering (FASE)*, pages 210–225, Rome, Italy, March 2013.

[62] 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.

[63] M. S. Tschantz and M. D. Ernst. Javari: Adding Reference Immutability to Java. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230, San Diego, CA, USA, October 2005.

[64] R. Vallée-Rai, É. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In *9th International Conference on Compiler Construction (CC)*, pages 18–34, Berlin, Germany, March 2000.

[65] 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.

[66] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proc. of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, pages 32–41, San Diego, California, USA, 2007.

[67] J. Whaley. Java Binary Decision Diagram Library. http://javabdd.sourceforge.net/. Last checked on December 26, 2017.

[68] 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.

[69] L. Xiao, S. Matsumoto, T. Ishikawa, and K. Sakurai. SQL Injection Attack Detection Method Using Expectation Criterion. In *Fourth International Symposium on Computing and Networking, CANDAR*, pages 649–654, Hiroshima, Japan, 2016.