TAJ: Effective Taint Analysis of Web Applications

Omer Tripp — IBM Software Group — omert@il.ibm.com
Marco Pistoia, Stephen Fink, Manu Sridharan — IBM T. J. Watson Research Center — {pistoia,sjfink,msridhar}@us.ibm.com
Omri Weisman — IBM Software Group — weisman@il.ibm.com

Abstract

Taint analysis, a form of information-flow analysis, establishes whether values from untrusted methods and parameters may flow into security-sensitive operations. Taint analysis can detect many common vulnerabilities in Web applications, and so has attracted much attention from both the research community and industry. However, most static taint-analysis tools do not address critical requirements for an industrial-strength tool. Specifically, an industrial-strength tool must scale to large industrial Web applications, model essential Web-application code artifacts, and generate consumable reports for a wide range of attack vectors.

We have designed and implemented a static Taint Analysis for Java (TAJ) that meets the requirements of industry-level applications. TAJ can analyze applications of virtually any size, as it employs a set of techniques designed to produce useful answers given limited time and space. TAJ addresses a wide variety of attack vectors, with techniques to handle reflective calls, flow through containers, nested taint, and issues in generating useful reports. This paper provides a description of the algorithms comprising TAJ, evaluates TAJ against production-level benchmarks, and compares it with alternative solutions.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Static Analysis, Web Applications, Security

Keywords Security, Static Analysis, Taint Analysis, Information Flow, Integrity, Web Applications

1. Introduction

Information-flow violations [8] comprise the most serious security vulnerabilities in today's Web applications. In fact, according to the Open Web Application Security Project (OWASP) [26], they constitute the top six security problems. Automatically detecting such vulnerabilities in real-world Web applications may be difficult due to their size and complexity. Manual code inspection is often ineffective for such complex programs, and security testing may remain inconclusive due to insufficient coverage.

This paper proposes a static-analysis solution that detects four of the aforementioned top six security vulnerabilities [26]:

- *Cross-site scripting (XSS) attacks* (the most common vulnerability) may occur when a Web application accepts data originating from a user and sends it to another user's browser without first validating or encoding it. For example, suppose an attacker embeds malicious JavaScript code into his or her profile on a social Web site. If the site fails to validate such input, that code may execute in the browser of any other user who visits that profile.
- *Injection flaws* (the second most frequent vulnerability) arise when a Web application accepts input from a user and sends it to an interpreter as part of a command or query, without first validating it. *Via* this vulnerability, an attacker can trick the interpreter into executing unintended commands or changing data. The most common attack of this type is Structured Query Language injection (SQLi).
- *Malicious file executions* (the third most common vulnerability) happen when a Web application improperly trusts input files or uses unverified user data in stream functions, thereby allowing hostile content to be executed on the server.
- *Information leakage and improper error-handling attacks* (the sixth most common vulnerability) take place when a Web application leaks information about its own configuration, mechanisms, and internal problems. Attackers use this weakness to steal sensitive data or refine their attacks.

Each of these vulnerabilities can be cast as a problem in which tainted information from an untrusted "source" propagates, through data and/or control flow, to a high-integrity "sink" without being properly *endorsed* (*i.e.*, corrected or validated) by a "sanitizer".

To address these vulnerabilities, the research community has focused much attention on static analysis for information-flow security of Web applications. Unfortunately, many of the published approaches do not immediately apply to industrial Web applications. Many existing solutions require use of complex, non-standard type systems, which are unlikely to enjoy broad adoption [36; 24; 30]. Other solutions, based on precise program slicing [16], have not been shown to be sufficiently scalable [31; 13].

In this paper, we present Taint Analysis for Java (TAJ), a tool designed to be precise enough to produce a low false-positive rate, yet scalable enough to allow the analysis of large applications. TAJ incorporates a number of techniques to produce useful results on extremely large applications, even when constrained to a given time or memory budget. Furthermore, TAJ supports many complex features of Java Platform, Enterprise Edition (Java EE) Web applications that were often omitted from discussion in previous work.

In addition to a general overview of TAJ, this paper makes the following specific contributions:

- **Hybrid thin slicing.** We present a novel thin-slicing algorithm [33] that combines flow-insensitive data-flow propagation through the heap with flow- and context-sensitive data-flow propagation through local variables.

PLDI'09, June 15–20, 2009, Dublin, Ireland.

- **An effective model for static analysis of Web applications.** TAJ models reflective calls, tainted flows through containers, detection of taint in the internal state of objects, the JavaServer Pages (JSP), Enterprise JavaBeans (EJB), the Struts and Spring frameworks, and many other challenging features that have largely been ignored in the literature but are essential for effective analysis of Web applications.
- **A set of bounded analysis techniques.** When applications are extremely large and the end user still requires the analysis to terminate in a short time or stay below a given memory-consumption level, TAJ supports a prioritization policy that focuses the analysis on portions of the Web application that are likely to participate in taint propagation.
- **Implementation and evaluation.** We have implemented TAJ on top of the T. J. Watson Libraries for Analysis (WALA) [35] and shipped it as part of a commercial product [17]. We present implementation details of TAJ and experimental results obtained by running TAJ on industrial codes. These results include comparisons between our techniques and alternative approaches.

The remainder of this paper proceeds as follows: Section 2 presents a code sample demonstrating the challenges faced by taint analysis. Section 3 details the TAJ framework and underlying algorithms. In Section 4, we discuss code-modeling techniques to boost the accuracy and scalability of TAJ, and in Section 5, we present a technique that reduces the number of redundant reports presented to the user. Techniques for bounded analysis appear in Section 6. TAJ is evaluated in Section 7. Related work is described in Section 8. We conclude and discuss future work in Section 9.

2. Motivating Example

The example in Figure 1 shows some of the challenges faced by static taint-analysis tools.[1] Designed for expository purposes, the example shows a Java Web application reading untrusted data from servlet parameters. The example also shows some manipulation of data via reflection, which often occurs in the implementation of Web frameworks such as Struts or Spring. Since a practical tool cannot anticipate all Web frameworks that customer code may rely on, the tool is often forced to analyzed framework implementatations directly, including reflective calls.

In the code sample, the tainted strings `t1` and `t2` get their values from the calls to `getParameter` at lines 13 and 14, respectively. Next, a series of reflective calls acquire a reference to class `Motivating` at line 18, gain access to method `id` at lines 19-26, and invoke it at lines 31-36. Specifically, `id` is invoked (reflectively) three times: first with a tainted argument (the value corresponding to key `"fName"` in map `m`) at lines 31-32, then with a sanitized argument (the value corresponding to key `"lName"`, sanitized by a call to `URLEncoder.encode`) at lines 33-34, and finally with a non-tainted argument (the value corresponding to key `"date"`) at lines 35-36.

Since only the first call to `id` involves a tainted value, string `s1`—which flows into the constructor of object `i1` at line 37—is tainted, whereas `s2` and `s3`—which flow to the constructors of objects `i2` and `i3` at lines 38 and 39, respectively—are not. Method `println` is considered an XSS sink because it renders the string value of its input to the screen. Thus, the call to `println` with argument `i1` at line 40 poses a security issue. The other two calls to `println`—at lines 41 and 42—are benign.

[1] The example is partially inspired by the `Refl1` case in Stanford SecuriBench Micro [34].

This example illustrates many of the challenges posed by static taint analysis of real programs. To analyze this code precisely, the analysis must track flow through reflective calls [19] as well as through containers (the map `m`) and object fields (the objects flowing into `println` are `i1`, `i2` and `i3`, rather than `s1`, `s2` and `s3`). Static-analysis algorithms that cannot disambiguate the three calls to `invoke` on `idMethod`, as well as the three instances of `Motivating$Internal`, will fail to distinguish between the vulnerable and benign calls to `println`. Similar problems would arise from an inability to distinguish the results of the three calls to method `get` on map `m`, based on the key provided as an argument.

```
1:  public class Motivating {
2:   private static class Internal {
3:    private String s;
4:    public Internal(String s) {
5:     this.s = s;
6:    }
7:    public String toString() {
8:     return s;
9:    }
10:  }
11:  protected void doGet(HttpServletRequest req,
12:    HttpServletResponse resp) throws IOException {
13:   String t1 = req.getParameter("fName");
14:   String t2 = req.getParameter("lName");
15:   PrintWriter writer = resp.getWriter();
16:   Method idMethod = null;
17:   try {
18:    Class k = Class.forName("Motivating");
19:    Method methods[] = k.getMethods();
20:    for (int i = 0; i < methods.length; i++) {
21:     Method method = methods[i];
22:     if (method.getName().equals("id")) {
23:      idMethod = method;
24:      break;
25:     }
26:    }
27:    Map m = new HashMap();
28:    m.put("fName", t1);
29:    m.put("lName", t2);
30:    m.put("date", new String(Date.getDate()));
31:    String s1 = (String) idMethod.invoke(this, new
32:      Object[] {m.get("fName")});
33:    String s2 = (String) idMethod.invoke(this, new
34:      Object[] {URLEncoder.encode(m.get("lName"))});
35:    String s3 = (String) idMethod.invoke(this, new
36:      Object[] {m.get("date")});
37:    Internal i1 = new Internal(s1);
38:    Internal i2 = new Internal(s2);
39:    Internal i3 = new Internal(s3);
40:    writer.println(i1); // BAD
41:    writer.println(i2); // OK
42:    writer.println(i3); // OK
43:   } catch(Exception e) {
44:    e.printStackTrace();
45:   }
46:  }
47:  public String id(String string) {
48:   return string;
49:  }
50: }
```

Figure 1. `Motivating` Program

3. Core Taint Analysis

TAJ takes a Web application and its supporting libraries, and checks it with respect to a set of "security rules". Each *security rule* is of the form (S_1, S_2, S_3), where S_1 is a set of "sources", S_2 is a set of

"sanitizers", and S_3 is a set of "sinks". A *source* is a method whose return value is considered *tainted*, or untrusted.[2] A *sanitizer* is a method that manipulates its input to produce taint-free output. A *sink* is a pair (m, P), where m is a method that performs security-sensitive computations and P contains those parameters of m that are vulnerable to attack *via* tainted data. TAJ statically checks that no value derived from a source is passed as an input to a sink unless it first undergoes appropriate sanitization.

TAJ consists of two stages. The first phase performs pointer analysis and builds a call graph. The second phase runs a novel slicing algorithm to track tainted data.

3.1 Pointer Analysis and Call-graph Construction

The TAJ architecture supports any preliminary pointer analysis and call graph construction algorithm. The current implementation relies on a context-sensitive variant of Andersen's analysis [1] with on-the-fly call graph construction.

TAJ employs a custom context-sensitivity policy tuned to address precision and performance issues that arise when analyzing real codes. Most methods are analyzed with one level of object sensitivity [18; 22], in which the context of a method invocation consists of the invoked method and the object abstraction representing the receiver. The policy also includes careful treatment of collections and security-related Application Programming Interfaces (APIs). In particular:

- Java collection classes are treated with unlimited-depth (up to recursion) object-sensitivity. This means that all internal objects of a collection are cloned for each collection instance. As a result, the contents of Java collections from different allocation sites are fully disambiguated, eliminating a major source of pointer-analysis pollution.
- The pointer analysis adds one level of call-string context to calls to library factory methods. These methods tend to pollute pointer-flow precision if handled without context sensitivity, because all the objects created by a factory method share the same allocation site.
- Taint-specific APIs, such as sources and sinks, are also analyzed with one level of call-string context. This is necessary due to the special role these APIs play in taint propagation. In the example given in Figure 1, this context allows TAJ to disambiguate the two calls to source method `getParameter` at lines 13 and 14, even though they are performed on the same receiver object.

As for other dimensions of precision, the pointer analysis of TAJ is field-sensitive [29]. Furthermore, it relies on a Static-Single Assignment (SSA) register-transfer language representation of each method [6], which gives a measure of flow sensitivity for points-to sets of local variables [14].

3.2 Hybrid Thin Slicing

Using the preliminary pointer analysis and call graph, the second phase of TAJ tracks data flow from tainted sources using *hybrid thin slicing*, a novel thin-slicing algorithm [33]. Hybrid thin slicing combines flow-insensitive reasoning about flow through the heap with flow- and context-sensitive tracking of flow through local variables.

Thin slicing [33] is a good basis for taint analysis since a thin slice typically captures the statements most relevant to a tainted flow. A *forward thin slice* from a statement t consists of those statements that are data-dependent on t [16], *excluding base-pointer dependencies*: for a store statement `x.f = y`, dependencies due to uses of the base pointer `x` are ignored; loads are handled similarly. Thin slices are typically much smaller and more understandable than program slices. Note that in [33], the term "thin slice" refers to a *backward thin slice*, in which data dependencies are considered in the opposite direction, while here we use this term to mean a forward thin slice.

Thin slices do not include control dependencies, and hence TAJ does not track the corresponding indirect information flow. Experience shows that attacks based on control dependence are rare and complex, and thus less important than direct vulnerabilities.

Hybrid thin slicing combines aspects of the previously proposed context-sensitive (CS) and context-insensitive (CI) thin slicing algorithms [33], achieving a better tradeoff between scalability and precision for taint analysis. Like CS thin slicing, hybrid thin slicing tracks flow through local variables with flow and context sensitivity. However, unlike CS thin slicing, the hybrid technique does not track heap data dependencies *via* additional method parameters and return values, as this treatment is a scalability bottleneck [33]. This handling of heap dependencies by CS thin slicing is also unsound for multi-threaded programs since it is partially flow-sensitive, and many of our target Web applications are multi-threaded. Instead, hybrid thin slicing tracks heap data dependencies *via* direct edges from stores to loads. Such edges are added based on the preliminary pointer analysis, as in CI thin slicing. As we shall show in Section 7, the hybrid approach yields better scalability than CS thin slicing and better precision than CI thin slicing (with better performance than CI in some cases).

Hybrid thin slicing performs a demand-driven traversal over a special System Dependence Graph (SDG) [16] called the *Hybrid SDG* (HSDG). Nodes in an HSDG correspond to load and store statements in the program, as well as call statements representing source and sink methods.

An HSDG has two types of edges representing data dependence: "direct edges" and "summary edges". A *direct edge* connects a store to a load and represents a data dependence computed by a preliminary pointer analysis (as in CI thin slicing [33]). A *summary edge* can connect s to t if t is transitively data-dependent on s purely *via* flow through local variables; flow through the heap is excluded. Summary edges are obtained on demand by computing context-sensitive reachability over a *no-heap SDG*—an SDG that elides all control- and data-dependence edges reflecting flow through heap locations. Note that the no-heap SDG includes no successor edges for sanitizer return and sink call statements, since we need not track flow beyond these statements.

TAJ computes the successors of a statement x in the HSDG on demand, as follows:

- If $x = st$ is a store statement, then precomputed points-to information is used to connect st to all load statements l such that the base pointers of st and l are may-aliased.
- Otherwise, a context-sensitive slice is computed from program point x on the no-heap SDG using the Reps-Horwitz-Sagiv (RHS) tabulation algorithm [28]. All the statements in the slice corresponding to store instructions and sink invocations are registered as the successors of x.

Figure 2 shows an example, which displays the slice computed on the no-heap SDG corresponding to a load-to-store summary edge in the HSDG.

To find tainted flows, we compute reachability in the HSDG from each source-call statement s, adding the necessary direct and summary edges on demand. The nodes reachable from s represent the load, store, and sink statements *directly* data-dependent on s (ignoring base-pointer data dependencies). Our final output reconstructs thin slices from s to sensitive sinks *via* the HSDG and relevant no-heap SDGs.

[2] Some methods, such as `RandomAccessFile.readFully` in package `java.io`, receive parameters by reference and taint their internal state. TAJ also supports the specification of such methods as sources.

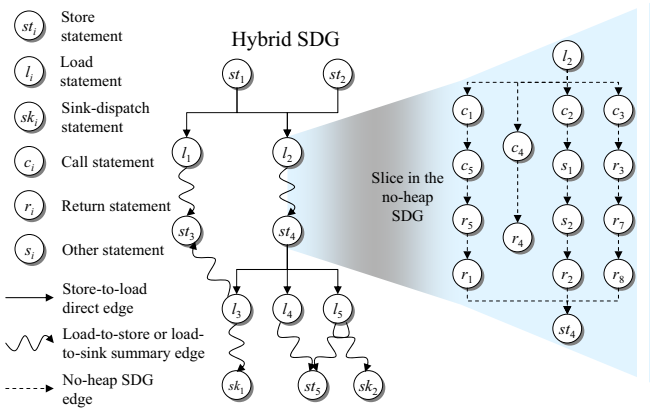

Figure 2. Fragment of the HSDG

Comparison to Refinement-Based Pointer Analysis The hybrid thin-slicing algorithm can be described in terms of Refinement-Based Pointer Analysis (RBPA) [32]. Our direct edges from stores to loads correspond to *match edges* in RBPA. However, whereas the initial match edges in RBPA are based solely on field types, our algorithm computes initial match edges more precisely with a preliminary pointer analysis. Thus, while RBPA starts with a relatively imprecise solution and tries to recover precision through refinement, our approach starts with a relatively precise solution and does not refine.

A second difference between hybrid thin-slicing and RBPA is that the latter is forced to collapse strongly-connected call-graph components discovered during analysis. In contrast, since hybrid thin slicing does not refine match edges, it can also handle method recursion on match-edge-free subpaths precisely.

We studied the refinement-based approach and found that it introduces additional false positives compared to our approach, which we wished to avoid.

4. Code-modeling Techniques

In this section, we detail modeling techniques used to deal with code that TAJ either (1) cannot analyze (for example, native code) or (2) does not analyze directly for efficiency. In Section 4.1, we describe modeling specific to checking security properties, and in Section 4.2 we discuss more generally-applicable modeling techniques.

4.1 Security-specific Modeling

This section presents the code-modeling techniques employed by TAJ to effectively capture taint flows due to objects that carry tainted data as part of their internal states. It also shows how code modeling can be used to detect information leakage and improper error handling, a vulnerability defined in Section 1.

4.1.1 Taint Carriers

TAJ must handle cases where tainted data is passed to a sensitive sink *via* the internal state of a parameter, rather than the parameter itself. For example, in line 40 of Figure 1, a tainted `String` is passed to the `println` sink method indirectly, wrapped in an `Internal` object. This constitutes a security vulnerability which must be flagged by TAJ.

We use the term *taint carrier* to refer to an object whose internal state contains tainted data. TAJ handles taint carriers by reporting an issue whenever a taint carrier is passed as a sensitive parameter to a sink. Ideally, specifications would indicate precisely which access paths of sink parameters must not hold sensitive data, in which case the analysis could check these access paths directly. However, in many cases such specifications would be difficult or impossible to write, *e.g.*, when a sink uses native code or when it takes a parameter of interface type. To avoid missing issues, we treat any passing of a taint carrier to a sink as a possible bug.

Our implementation handles taint carriers by using the preliminary pointer analysis to augment the HSDG with additional edges. We consider a pointer-analysis solution as a *heap graph* [10]—a bipartite graph whose nodes represent abstract objects (henceforth, *instance keys*) and abstract pointers (henceforth, *pointer keys*). Given pointer key P and instance key I, an edge $P \rightarrow I$ indicates that P may point to I. An edge $I \rightarrow P$ indicates that P represents either a field of an object instance modeled by I, or the array contents of array instance I. Based on the heap graph, our algorithm employs the following logic to supplement the HSDG with data-flow edges from store statements to sink calls:

1. For any store statement st, let pk be the pointer key corresponding to the base pointer for the store. Let I_{st} be the points-to set of pk. Similarly, for any sink invocation sk, let I_{sk} denote the union of the points-to sets for the sensitive formal parameters of sk.
2. For each sink invocation sk, let I^*_{sk} be the set of instance keys reachable in the heap graph from the instance keys in I_{sk}.
3. Add edge $st \rightarrow sk$ to the HSDG if and only if $I_{st} \cap I^*_{sk} \neq \varnothing$.

For example, on the code of Figure 1, the taint-carrier-detection algorithm synthesizes the HSDG edge from the store at line 5 (in the clone of the `Internal` constructor corresponding to the allocation at line 37) to the sink call at line 40. With this edge, TAJ discovers the tainted flow that from line 13 reaches line 40 going through lines 37 and 5.

Step 2 of this algorithm may introduce false positives due to pointer analysis imprecision; we address this concern in Section 6.2.

4.1.2 Handling Exceptions

TAJ employs special-case modelling for vulnerabilities from exceptions. Consider, for example, the following block of code, which exemplifies the (unfortunately) common practice of rendering caught exceptions to the screen:

```
protected void doGet(HttpServletRequest req,
    HttpServletResponse resp) throws IOException {
  try {
   ...
  } catch (Exception e) {
    resp.getWriter().println(e);
  }
}
```

This code may leak sensitive information concerning the internal makeup of the application, as the default implementation of `Exception.toString` includes the result of the call to `Exception.getMessage` in its return value. As mentioned in Section 1, exposing this information to unauthorized observers currently constitutes the sixth most common security vulnerability in today's Web applications. To account for this type of flow—where it is not clear which source should be defined—TAJ synthesizes code that calls `getMessage` on a caught `Exception` object, and marks this synthetic code as a source of tainted data. Similar techniques model taint flow in certain JSP tags.

4.2 General Models

A basic approach to static analysis would model all available program and library code directly. However, to improve performance and precision, we tune the analysis with various higher-level models. These models are general purpose and may be useful for other static analyses.

Our models serve several functions in the analysis. First, code that cannot affect the flow of taint through the program can be ignored. Second, certain library methods can be summarized, creating models that provide a succinct yet sound description of the relevant behavior. Finally, models can compensate for cases in Web-framework clients where data flow is not clear from the code alone. We next discuss specific examples of how TAJ employs synthetic models of code.

4.2.1 Code-reduction Techniques

A simple, yet effective, code-reduction optimization is to exclude benign library classes, packages, and subpackages based on a whitelist generated by hand.

A more interesting type of modeling can simplify data-flow propagation by substituting simpler models for library methods, where the simpler model encodes the behavior with respect to flow of taint. For example, taint analysis does not need to analyze the complex manipulations in the implementation of `URLEncoder.encode`; it suffices to observe that this method returns some string that is sanitized according to the relevant rules.

Using this insight, TAJ gives special treatment to `String` operations, which arise frequently in tainted flows, have relatively simple semantics, but are often difficult to analyze precisely. We define the family of *string carriers* to include classes `String`, `StringBuffer` and `StringBuilder` in package `java.lang`. String-carrier instances are handled as if they were primitive values by inserting appropriate operations into the SSA representation for each method that manipulates string carriers and modeling their public APIs. With this treatment, the analysis need not track the contents of string carriers through the heap during pointer analysis, and loses no precision.

TAJ also applies special treatment to dictionaries from the standard libraries. Clearly, the general problem of tracking data flow through a hash set is difficult. However, we observe that many Web applications access hash structures with keys that can be resolved as constants. Exploiting this observation, TAJ applies special logic to track data flow between statements making read or write access to hash structures, whenever the key can be statically resolved as a constant. For example, in the following code, TAJ will determine that `o1` cannot flow to `o2` based on the keys used to access `s`:

```
HttpSession s;
Object o1;
... // Initialization code here.
s.setAttribute("a", o1);
Object o2 = s.getAttribute("b");
```

Precise handling of this special case dramatically improves precision, as this idiom appears often in Web applications.

4.2.2 Approximating the Behavior of Web Frameworks

In many Web applications, precise analysis requires information that is external to the program's code (*e.g.* configuration files). In these cases, TAJ often uses models that provide a conservative approximation of possible behavior.

For example, TAJ uses this type of modeling to support the Apache Struts framework. Struts is an implementation of the Model View Controller (MVC) pattern, where the *controller* is configured based on an eXtensible Markup Language (XML) file, the *model* consists of the business logic and the model state (which are represented by the `Action` and `ActionForm` classes respectively), and the *view* is a JSP file. The dispatch logic coded into the XML guides the framework when invoking business-logic elements, in the form of `Action` classes, by invoking the `execute` method on them. This method takes as a parameter an `ActionForm` instance, whose fields are populated by the framework based on user input (and should thus be considered tainted).

The fact that `Action` classes are dispatched by the Struts framework is modeled by treating them as entrypoints; the analysis begins at their implementation of `execute`, and synthesizes an appropriate program state. When `ActionForm`s are passed as arguments in calls to `Action.execute`, the analysis first checks which constraints the concrete implementation of `execute` places on its `ActionForm` parameter—in the form of cast operations—and then simulates the passing of all compatible subtypes of `ActionForm` as parameters to `execute`. For each of these subtypes, the system generates a synthetic constructor which assigns tainted values to all its fields (this is done recursively, as fields may be of compound types).

Another crucial challenge for Java EE applications concerns accounting for EJB calls. Consider an enterprise bean having remote interface `EB2`, home interface `EB2Home` and bean class `EB2Bean`. Assume that a method `m2` is declared in `EB2` and implemented in `EB2Bean`. To call `m2` remotely, a method `m1` in bean `EB1Bean` would perform the following:

```
Context initial = new InitialContext();
Object objRef = initial.lookup("java:comp/env/ejb/EB2");
EB2Home eb2Home = (EB2Home) PortableRemoteObject.narrow(
    objRef, EB2Home.class);
EB2 eb2Obj = eb2Home.create();
eb2Obj.m2();
```

At the bytecode level, the call to `eb2Obj.m2` delegates to an implementation generated automatically by the Java EE deployment tool. The Java EE container consults run-time registries and a backing persistence manager (usually a database) to map the remote `m2` method call to the actual method implementation, and then passes a message to the process hosting the home container for `EB2Bean` using Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP). The receiving process unmarshalls the RMI-IIOP message, activates the relevant component through the bean lifecycle implementation, and finally delegates to a reflective call to complete the remote invocation.

Analyzing bytecode alone, it would be impossible to resolve this remote method invocation, since the relevant dispatch tables are encoded in the XML deployment descriptor, and read by the container at run-time. To guarantee soundness, it would then be necessary to analyze the container code, but given its complexity and size, this would limit the scalability and precision of the analysis. TAJ does not analyze the thousands of methods in the container implementation that perform the remote invocation of `m2`; instead, it bypasses the container and recognizes special semantics for the call to `m2`. To achieve this result, TAJ consults the deployment descriptor, generates an analyzable artifact representing the semantics of a call to `m2`, and models the call as dispatching to this artifact.[3] The simple semantics there suffice to construct a correct call graph incorporating the call to `m2`, independent of the container implementation. Effectively, TAJ ignores the generated deployed code, and models the application-level semantics of EJB calls directly, based on direct analysis of the deployment descriptor. This functionality is essential for accurate analysis of EJB calls in Java EE, and—to our knowledge—is not supported by any other static-analysis implementation.

[3] Details on the construction of this analyzable artifact are given in [9].

Our choice to model EJB calls in this way has three advantages. First, it allows scalability, as the body of code to analyze is dramatically reduced. Second, it enhances precision, as mapping an EJB remote method to its actual implementation in the corresponding EJB class without analyzing a container's RMI-IIOP implementation minimizes the risk of data-flow pollution. Finally, portability is accomplished by virtue of the fact that the analysis results are not dependent on the container implementation.

4.2.3 Reflection APIs and Native Methods

TAJ includes significant machinery to approximate the behavior of Java reflection APIs, such as `Class.forName` and `Method.invoke`. When the value of an argument to a reflection API can be inferred (for example, when it is constant), the system synthesizes a relevant abstraction in place of the reflective call.

Finally, the system relies on hand-coded synthetic models for native methods in the standard Java library. This is essential not only for tracking data- and control-flow information, but also because native calls figure prominently in security-related operations. For example, `Thread.start` and the four overloads of `AccessController.doPrivileged` are all based on native APIs. Failure to analyze these methods would render the analysis useless for many real-world Web vulnerabilities.

5. Eliminating Redundant Reports

Some tainted flows reported by the analysis may be redundant to a user. The analysis can compute all flows of the form $sr \rightsquigarrow sk$, where sr is a source, sk is a sink, and no sanitizer is present along the path from sr to sk. However, from a user's perspective, this may be too much information, since many of these flows might redundantly expose a single logical flaw.

We now describe an approach to address this potential redundancy. Considering the insertion of a sanitizer invocation into the path as a *remediation action*, we propose an approach whereby flows are grouped together according to the remediation actions they map to. TAJ reports one representative per group, rather than all the flows.

Formally, we define a *library call point* (LCP) to be the last statement along a flow from a source to a sink where data flows from application code (*i.e.*, the project's source code) to library code (*i.e.*, libraries referenced by the project). Data can flow from application to library in one of three ways: (1) a library method is invoked from application code, (2) a library memory location is written from application code, or (3) an application memory location is read from library code.

With this definition at our disposal, we can introduce an equivalence relation $\sim$, as follows: Let U and V be two flows. Then $U \sim V$, if and only if (1) $U|_{LCP} \equiv V|_{LCP}$ (where $X|_{LCP}$ is the part of flow X extending from the source to the LCP inclusive), and (2) U and V require the same remediation action. The equivalence classes induced by $\sim$ are the sets of flows into which flows are classified.

For example, considering the call graph illustrated in Figure 3, we can define U as the flow along path p_1 and V as the flow through p_2. We note that nodes n_{10} and n_{11} are both sinks with the same issue type (for example, XSS or SQLi), and so they both require the same remediation action, or sanitation logic. Since U and V both transition from application code to library code at the same point (n_4), it follows that $U \sim V$.

If we instead define U as the flow along p_3 and V as the flow through p_4, then U and V do *not* share the same LCP, and thus belong in different equivalence classes, despite flowing from the same source to the same sink. The justification for this is that, potentially, the remediation action introduced for U will not remove the security threat exposed in V (*e.g.*, if a sanitizer is called from node n_3). Similarly, the flows through p_4 and p_5 are both reported, although they originate from the same source and pass through the same LCP, since they end at sinks corresponding to different issue types, and may therefore require different remediation actions.

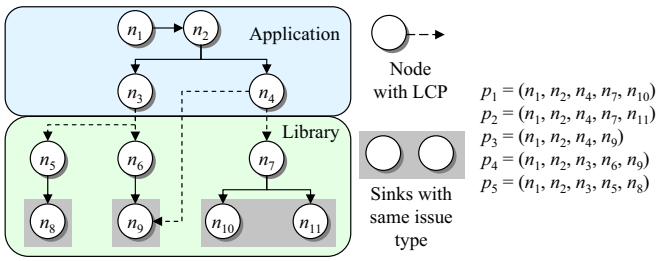

Figure 3. Call Graph Illustrating the LCP Concept

Our analysis of the example in Figure 3 demonstrates the twofold advantage of the LCP-based method of classification: (1) the part of flow X that is under the developer's control is precisely $X|_{LCP}$,[4] and (2) if flow X is a representative of equivalence class $[X]_\sim$, then, once the vulnerability exposed in X is remedied, all the other flows in $[X]_\sim$ will be remedied as well. This compact, action-oriented report greatly improves user experience.

TAJ uses the following algorithm to generate minimal reports according to this property:

1. After hybrid thin slicing, a view of the HSDG restricted to statements in the slice is produced. This view is then traversed backwards starting from sinks, and LCPs are identified as transitions from library to application code. This step outputs a relation mapping sinks to LCPs.
2. For each LCP lcp, the associated sinks are grouped into equivalence classes according to the indicated remediation logic. Let S_{lcp} be the set of sink-equivalence-class representatives for an LCP lcp.
3. We identify every source/LCP pair (sr, lcp) for which there is a data-flow path from sr to lcp, traversing the hybrid thin slice. For each such pair (sr, lcp), for each $sk \in S_{lcp}$, report $sr \rightsquigarrow lcp \rightsquigarrow sk$ as a potential vulnerability, if so indicated by the security rules.

6. Bounded Analysis Techniques

When analyzing large applications, a user may decide to constrain the analysis to a specific time and memory budget. This section presents a set of techniques for customizing the analysis to produce satisfactory results within a fixed budget.

6.1 Priority-driven Call-graph Construction

Under a fixed time and memory budget, TAJ may terminate pointer analysis and call-graph construction early, yielding an *underapproximate* result. The result is underapproximate since the points-to relation computed by Andersen's analysis grows monotonically. While the underapproximate call graph and points-to relation may not be sound, they are often sufficient for finding many bugs.

When terminated early, the order in which the pointer analysis adds and solves constraints can have a strong impact on the number of bugs discovered in taint analysis. TAJ uses *priority-driven call-graph construction* to heuristically improve pointer analysis quality

[4] Note, however, that $X|_{LCP}$ may transition between application and library code multiple times, which implies that parts of this sub-flow may not be accessible to the developer.

within a fixed budget. The priority heuristic favors the analysis of methods that are more likely to generate and propagate taint. Our experiments show that it enables the detection of a significantly larger number of taint vulnerabilities than chaotic iteration when TAJ runs in a constrained time or memory budget.

Priority-driven call-graph construction forces the pointer analysis to add constraints first from higher-priority methods—in this case, those methods likely to be more relevant to taint analysis. Our pointer analysis iterates between two key phases: (1) constraint solving, which computes points-to relationships and notes newly discovered targets for virtual calls, and (2) constraint adding, which adds constraints for new methods found to be reachable by constraint solving.[5] Priority-driven call-graph construction changes phase (2); it assigns a priority to pending methods and ensures that in each pass, constraints are only added for the highest-priority method. In this manner, methods more relevant to taint analysis are processed earlier by the pointer analysis.

More formally, let $G = (N, E)$ be the call graph under construction. We assume that the budget takes the form of a bound on the number of call graph nodes, *i.e.*, a number $maxNodes$ such that $|N| \leq maxNodes$.

The construction of G is governed by a priority policy $\Pi : N \rightarrow \mathbb{N}$, where smaller numbers mean higher priority. Upon creation of a new call-graph node n,[6] Π uses the following *initial-assignment rule* to assign n a priority: if n is a source node, then $\Pi(n) := 0$; otherwise, $\Pi(n) := maxNodes$. The idea behind this rule is that source nodes represent taint generation and so should be given the lowest priority value (corresponding to the highest importance) in the construction of G.

Pointer analysis and call-graph construction begins by instantiating call-graph nodes that represent invocations of application entrypoints and adding them to a priority queue Q. Then—while $Q \neq \varnothing$ and the analysis budget has not been met—the following steps are run in a loop:

1. A node n with the lowest priority value in Q is dequeued.
2. A set $T_n \subseteq N$ is constructed as follows: T_n contains (1) all the predecessors and successors of n in G, and (2) all the existing nodes that represent methods containing a load statement that, according to the pointer analysis, matches a store statement in the method represented by n (in which case, there will be a direct edge from the store to the load in the HSDG). Any node $t \in T_n$ with no priority assigned yet is given priority $\Pi(t)$ according to the initial-assignment rule.
3. The priorities of all nodes $t \in T_n$ are updated according to the following *update rule*: $\Pi(t) := \min\{\Pi(t), \Pi(n) + 1\}$.
4. Nodes in T_n are added to Q if necessary.
5. Any node $t \in T_n$ whose priority changed in Step 3 propagates its priority to all the nodes in T_t. This propagation process runs to a fixed point.
6. Constraints for n are added to the pointer-analysis-constraint system, and constraint solving runs to a fixed point.

The reasoning behind steps 2 and 3 above is based on the *locality-of-taint principle*, which is an observation to the effect that if taint flows through a particular code location, then nearby code locations (those in the T_n set above) are also likely to be relevant to taint propagation. In Step 2, methods containing load statements matching the stores in n are considered "near" n due to the possible corresponding flow through the heap. Step 3 ensures that all nearby methods for n have a priority close to that of n—since n has been deemed relevant to taint, nearby methods are also likely to be relevant.

Another important aspect of the technique is the role played by Π. Since Π assigns the maximal possible priority as the initial priority for source nodes, the algorithm favors nodes closer to the sources of taint. This bias leads to the discovery of more taint-related bugs in practice.

6.2 Useful Bounds on Analysis Dimensions

The call-graph construction process is not the only aspect of our algorithm that can be bounded. Limits can also be set on the slicing process, the algorithm searching for nested taint under object abstractions, and many other components of the framework. The choice of which dimensions to bound, and how to bound them, is crucial for an analysis to yield satisfying results under constraints. In what follows, we discuss bounds we found to be useful when running TAJ in environments with limited resources.

6.2.1 Slice Size

There are two ways to constrain the size of a slice, when computed through hybrid thin slicing. One is to limit the number of heap store-to-load transitions, and another is to cast constraints on the slice sizes through the no-heap SDG. Our experience suggests that limiting the number of heap transitions yields better overall results. The main reason for this is the loss of precision entailed by data flow through the heap, which relies on a flow-insensitive pointer analysis. It is straightforward to constrain the size of the slice in this manner during hybrid thin slicing by keeping track of store-to-load expansions of the slice.

6.2.2 Flow Length

The loss in precision entailed by long series of heap transitions, along with other over-approximations that are required for sound analysis (such as the static resolution of reflective and virtual calls), account for a strong correlation between the length of a reported flow and its classification as a true positive. Our empirical studies suggest that the longer a flow is, the less likely it is to be a true positive. The theoretical justification for this is that the longer a flow is, the more opportunities the analysis has to err on the conservative side. Section 7 presents further confirmation for this claim, in the form of empirical evidence.

6.2.3 Nested-taint Depth

Using the algorithm described in Section 4.1.1 to compute the set of objects reachable from a given object abstraction, without placing any restriction on the length of field-dereference sequences, can lead to an overly conservative analysis. This can happen, for example, if a data-structure reference is stored as the field of an object, thereby bridging between that object and the transitive fields of all the objects stored in the data structure. While dismissal of long field-dereference sequences from consideration (by introducing a bound) is theoretically unsound, our experience suggests that it is highly unlikely for a sink to consume data that is stored "deep" inside the state of its arguments. Empirically, we found 2 levels of field dereference to be sufficient, as will be discussed in Section 7.

7. Experimental Results

Since TAJ was designed to support a commercial product [17], it has undergone extensive evaluation on a large set of large industrial benchmarks. In this section, we present and discuss results from the main experiments, run on 22 benchmarks.

[5] For context-sensitive analysis, constraints are added separately for each method clone.

[6] A call-graph node represents a method in some calling context, as determined by the context-sensitivity policy.

7.1 Experimental Setup

TAJ is a WALA client [35] written in Java and implemented as an Eclipse V3.4 plug-in. Its testing environment comprised an IBM desktop running Microsoft Windows XP Service Pack 3, with a 1.86 GHz Intel Core 2 processor and 3GB of RAM. TAJ was run on top of Sun Microsystems Java Standard Edition Runtime Environment (JRE) V1.6.0_06, using 1 GB of heap space.

Algorithm		Nested Taint	LCPs	Modeling	Priorities	Other Bounds
Hybrid	*Unbounded*			✓		
	Prioritized			✓	✓	
	Optimized	✓	✓	✓	✓	✓
CS				✓		
CI				✓		

Table 1. Settings Used for the Evaluated Algorithms

Five different algorithms were evaluated: CS thin slicing [33], CI thin slicing [33], and three variants of hybrid thin slicing—an *unbounded* version running to completion, a *prioritized* version running under a call-graph size bound with the priority-driven scheme, and a *fully optimized* version running with all the optimizations and bounds described above. Details concerning the settings we used are provided in Table 1. A call-graph bound of 20,000 nodes was used for the prioritized and fully optimized versions of the hybrid algorithm. The fully optimized variant also restricts heap transitions (during slicing) to 20,000, filters out flows whose length is greater than 14, and allows no more than 2 field dereferences when running the taint-carrier detection algorithm.

Table 2 lists information about the applications used for the evaluation, including supporting libraries. The identifiers `A`, `B`, `I`, `S` and `ST` are used instead of the actual names of the corresponding applications for anonymity. Together, the 22 benchmarks included in our experiments capture all the challenges discussed in earlier sections. Most of the benchmarks make heavy use of Web frameworks and reflection, and—as confirmed by manual inspection—all of the selected benchmarks expose non-trivial taint flows. The larger benchmarks also pose scalability challenges. Some of the applications in the list have been included in previous studies on taint analysis [20].

7.2 Discussion

Results concerning the performance of each of the algorithms are detailed in Table 3. For nine of the applications, we manually classified the reported issues into true and false positives. The distribution of reports between these two categories appears in Figure 4.[7]

The following trends are apparent from the gathered data:

The unbounded hybrid algorithm offers a compelling tradeoff between performance and accuracy, when compared to the CI and CS configurations.

Table 3 presents the running times for all configurations on all benchmarks. The average running time for the unbounded hybrid configuration was 1051 seconds, which is a factor of 2.65X slower than the CI configuration. The CS configuration only completed on six of the smaller benchmarks. (On the remaining benchmarks, the CS analysis ran out of memory.) On these six benchmarks, the average running time for the unbounded hybrid configuration was 19.3 seconds, which is a factor of 29X faster than the CS configuration. Note from Table 1 that all configurations use synthetic models, which are key to good performance.

[7] Empty entries in Table 3, as well as missing columns in Figure 4, represent instances where the relevant algorithm failed to complete its run on the input program. The only algorithm for which such cases were registered is CS thin slicing.

Turning to accuracy, we examine the data breaking down false positives, as reported in Figure 4. We define the *accuracy score* for an analysis as the ratio between the number of true positives and the number of true and false positives combined, which corresponds to the total number of reported issues; a higher accuracy score indicates better accuracy. The respective accuracy scores of the unbounded hybrid, CS and CI algorithms are 0.35, 0.54 and 0.22. Note that the CS algorithm completed only on four of the benchmarks for which we manually evaluated accuracy. On these four benchmarks, the unbounded hybrid and CI algorithms had accuracy scores of 0.54 and 0.34, respectively. In the worst case (`A`), the unbounded hybrid algorithm is only 1.05 times less accurate than CS.

An interesting observation is that the unbounded hybrid and CI algorithms agree on the number of true positives for all the nine benchmarks for which we manually evaluated accuracy, which is expected given that both these algorithms are sound, while the CS algorithm has false negatives on `BlueBlog`, `I` and `SBM`; the numbers of false negatives are 2, 1 and 2, respectively. This reflects the fact that the CS algorithm is unsound with respect to multi-threaded applications, as we observed in Section 3.

We conclude that the unbounded hybrid algorithm represents an attractive tradeoff between performance and accuracy.

The prioritized hybrid algorithm offers superior accuracy and performance tradeoffs than the CI and unbounded hybrid configurations.

On the 9 benchmarks evaluated manually, the unbounded hybrid algorithm reports 556 false positives, while the prioritized version reports only 146 false positives. The 20,000-node call-graph bound did not lead to any missed true positives on 8 of the 9 benchmarks, with `Webgoat` as the only exception. Compared to CI (which is the most conservative algorithm), the prioritized hybrid algorithm introduces only 1 more false negative on `A` and 3 more false negatives on `BlueBlog`. The prioritized hybrid algorithm runs on average in 215 seconds, a factor of 1.8X faster than CI.

We conclude that the prioritized hybrid algorithm offers a compelling tradeoff between performance gain and accuracy loss compared to the unbounded alternatives.

The fully optimized version of the hybrid algorithm is more accurate than the prioritized variant and more efficient than the CI algorithm.

Compared to the prioritized hybrid algorithm, the fully optimized variant introduces only 1 new false negative (on `BlueBlog`). It finds, however, 14 more true positives (on `Webgoat`), and reports an overall of 74 false positives, compared to 146 false positives reported by the unoptimized prioritized hybrid configuration.

Intuitively, the fully optimized version recovers issues lost by the prioritized algorithm since the optimizations lead to a more efficient use of the limited analysis budget. The fully optimized version generates fewer false positives due mainly to restrictions on flow lengths and heap transitions in tainted flows.

The average running time of the fully optimized algorithm is 325 seconds, 21% faster than the CI algorithm. Overall, the fully optimized version is 1.5X slower than the unoptimized hybrid variant. On 13 out of the 22 benchmarks, the fully optimized algorithm is faster than the prioritized algorithm; its higher average running time is mainly due to degraded performance on `GridSphere`.

Taken together, these trends point to the fully optimized hybrid algorithm as the most attractive compromise between accuracy and scalability, among the configurations evaluated.

8. Related Work

In this section, we compare TAJ with other work in the area of static taint analysis. Related work on dynamic taint analysis is discussed

Application	Version	File Count	Line Count	Class Count		Method Count	
				App.	Total	App.	Total
A	1.0	121	746	43	2057	4272	150339
B	-	314	1680	246	9252	14552	328941
Blojsom	3.1	225	19984	254	7216	10688	354114
BlueBlog	1.0	32	650	38	1044	7628	269056
Dlog	3.0-BETA-2	240	17229	268	12957	7790	284808
Friki	2.1.1-58	40	2339	35	1133	3848	116480
GestCV	1.0	159	107494	124	5139	13673	473574
Ginp	1.0	121	387	73	2941	8076	277680
GridSphere	2.2.10	698	44767	676	32134	10671	385609
I	1.0	30	281	25	996	4254	149278
JSPWiki	2.6	724	27000	429	13087	9863	335828
Lutece	1.0	1039	3065	467	12398	7606	237137
MVNForum	1.0.2	969	8860	608	19722	8979	315527
PersonalBlog	1.2.6	135	47007	38	1644	4951	157794
Roller	0.9.9	325	4865	251	9786	7200	246390
S	-	168	2064	100	10965	6219	393204
SBM	1.08	125	5165	143	6506	8047	283069
SnipSnap	1.0-BETA-1	828	85325	571	17960	12493	455410
SPLC	1.0	106	12447	69	3526	6538	229417
ST	-	1451	594	5956	31309	24221	822362
VQWiki	1.0	280	31325	185	6164	4803	152341
Webgoat	5.1-20080213	245	17656	192	14309	6663	254726

Table 2. Statistics on the Applications Used in the Experiments

Application	Hybrid						CS		CI	
	Unbounded		Prioritized		Fully Optimized					
	Issues	Time(s)	Issues	Time(s)	Issues	Time(s)	Issues	Time(s)	Issues	Time(s)
A	54	43	33	54	37	23	51	554	73	88
B	25	1160	7	242	1	217	-	-	67	564
Blojsom	238	783	162	222	123	207	-	-	504	275
BlueBlog	19	5	19	5	12	6	14	376	30	7
Dlog	21	873	11	243	6	221	-	-	168	602
Friki	60	11	60	10	7	9	14	1392	125	11
GestCV	21	2461	20	182	7	209	-	-	255	760
Ginp	67	40	67	45	49	28	43	1028	309	75
GridSphere	803	6505	116	735	261	2467	-	-	853	1281
I	3	8	3	8	3	8	2	16	17	15
JSPWiki	68	159	67	270	26	118	-	-	381	192
Lutece	3	824	2	28	4	59	-	-	41	99
MVNForum	260	313	100	228	293	205	-	-	374	213
PersonalBlog	454	3708	108	386	48	740	-	-	1854	604
Roller	650	1495	87	175	230	268	-	-	3171	794
S	395	602	25	398	24	263	-	-	697	729
SBM	154	9	154	7	159	6	125	26	161	10
SnipSnap	91	279	89	167	94	153	-	-	397	291
SPLC	40	188	37	279	36	116	-	-	103	272
ST	731	933	369	207	347	277	-	-	1830	565
VQWiki	888	2450	303	383	545	565	-	-	2284	784
Webgoat	48	276	27	180	39	193	-	-	102	485

Table 3. Experimental Results Comparing between Hybrid Variants and Other Algorithms

in [4]. Works related to our contributions in the area of program slicing are surveyed in [33] and references therein. Of notable importance is the fact that while program slicing has been applied to taint analysis in the past, none of the proposed algorithms has been shown to scale to applications whose size is comparable to that of the large benchmarks against which we evaluate TAJ.

The notion of *tainted variables* as variables in security-sensitive code where untrusted values can flow became known with the Perl language. In Perl, using the `-T` option allows detecting tainted variables [37]. Typically, the data manipulated by a program can be tagged with security levels [8], which naturally assume the structure of a partially ordered set. Under certain conditions, this partially ordered set is a lattice [7]. In the simplest example, this lattice only contains two elements, indicated by *high* and *low*. Given a program, the principle of *non-interference* dictates that low-security behavior of the program be not affected by any high-security data, unless that high-security data has been previously verified and downgraded. [12]. The taint analysis problem described in this paper is an information-flow problem in which high data is the untrusted output of a source, low-security operations are those performed by sinks, and untrusted data is downgraded by sanitizers.

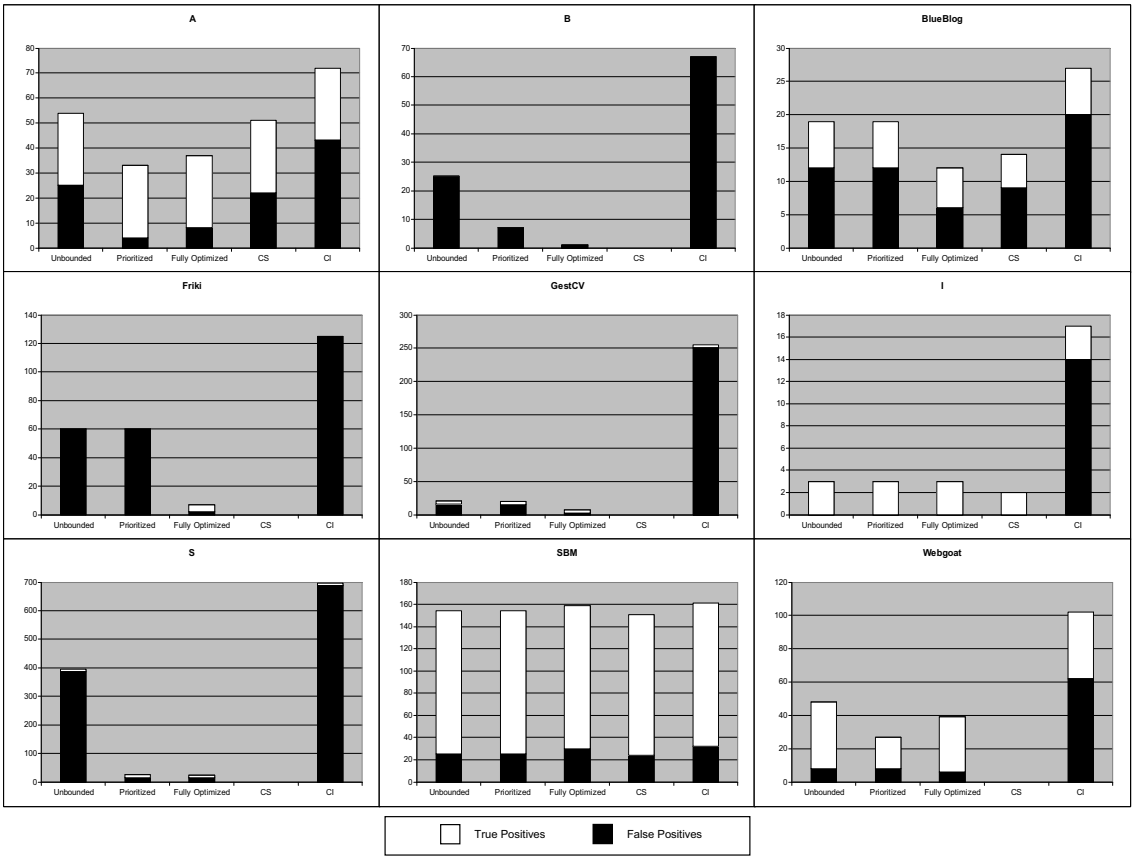

Figure 4. Classification of Reported Issues into True and False Positives on Key Benchmarks

Volpano, *et al.* [36] have shown a type-based algorithm that certifies implicit and explicit flows and also guarantees non-interference. Shankar, *et al.* present a taint analysis for C using a constraint-based type-inference engine based on `cqual` [30]. To find format string bugs, `cqual` uses a type-qualifier system [11] with two qualifiers: *tainted* and *untainted*. The types of values that can be controlled by an untrusted adversary are qualified as being tainted, and the rest of the variables are qualified as untainted. A constraint graph is constructed for a `cqual` program. If there is a path from a tainted node to an untainted node in the graph, an error is flagged. Myers' Java Information Flow (Jif) [24] uses type-based static analysis to track information flow. Jif is based on the Decentralized Label Model [25], and considers all memory as a channel of information, which requires that every variable, field, and parameter used in the program be statically labeled. Labels can either be declared or inferred.

Ashcraft and Engler [2] also use taint analysis to detect software attacks due to tainted variables. Their approach provides user-defined sanity checks to untaint potentially tainted variables. Pistoia, *et al.* [27] present a static analysis explicitly designed to detect tainted variables in privilege-asserting code in access-control systems based on stack inspection. They also perform a backward call-graph traversal starting at security-sensitive calls until a *boundary edge* between application code and library code is encountered. Boundary edges indicate to the user the optimal code locations where calls to privilege-asserting APIs should be inserted. This is similar to the LCP algorithm discussed in Section 5.

Snelting, *et al.* [31] make the observation that Program Dependence Graphs (PDGs) and non-interference are related in the following manner. Consider statements s_1 and s_2. If $dom(s_1) \not\rightsquigarrow dom(s_2)$ then, in a security-correct program, it must be the case that $s_1 \notin backslice(s_2)$. Here, *backslice* is the function that maps each statement s to its *static backwards slice*, consisting of all the (transitive) predecessors of s along control- and data-dependence edges in the PDG. Based on this observation, Hammer, *et al.* [13] present an algorithm for checking for non-interference: for any output statement s, it must be the case that $backslice(s)$ contains only statements that have a lower security label than s. Though promising, their approach has not been shown to scale.

Livshits and Lam [20] present an elegant approach for taint analysis for Java EE applications that is engineered to track taint flowing through heap-allocated objects. Their analysis requires prior computation of Whaley and Lam's flow-insensitive, context-sensitive may-points-to analysis, based on Binary Decision Diagrams (BDDs) [40]. The points-to relation is the same for the entire program, ignoring the program's control flow. By contrast, the PDG-based algorithm in [13] handles heap objects in a flow-sensitive manner, albeit at a much higher cost. Livshits and Lam's

taint analysis requires the presence of programmer-supplied descriptors for sources and sinks, as well as for library methods that handle objects through which taint may flow. This is reminiscent of our source and sink specification, as well as of our special treatment of string carriers. Their approach does not cover all the attack vectors addressed by TAJ, and does not deal with challenges such as flow through containers, nested taint and accurate handling of Web frameworks, which are essential for precise and comprehensive analysis of industrial applications. Furthermore, it is unclear whether BDD-based static analysis can scale to large applications when using object sensitivity [18]. Customization of the BDD-based approach (*e.g.* by fixing analysis budget and enhancing the analysis with a priority-driven scheme) also appears to be problematic.

Wassermann and Su extend Minamide's string-analysis algorithm [23] to syntactically isolate tainted substrings from untainted substrings in PHP applications. They label non-terminals in a Context-Free Grammar (CFG) with annotations reflecting taintedness and untaintedness. Their expensive, yet elegant, mechanism is applied to detect both SQLi [38] and XSS [39] vulnerabilities.

McCamant and Ernst [21] take a quantitative approach in information flow: instead of using taint analysis, they cast information-flow security to a network-flow-capacity problem, and describe a dynamic technique for measuring the amount of secret data that leaks to public observers.

9. Conclusion and Future Work

We have presented TAJ, an approach to taint analysis suitable for industrial applications. An experimental evaluation indicates that the hybrid algorithm TAJ uses for slice construction is an attractive compromise between context-sensitive and context-insensitive thin slicing. We also demonstrated priority heuristics to perform effective taint analysis in a limited budget, improving performance without significantly degrading accuracy.

In future research, we intend to introduce modular as well as incremental analysis capabilities into TAJ [5]. Also, we plan to investigate techniques from demand-driven pointer analysis [15] as an alternative to the priority heuristics presented here. Since string carriers are the most common taint mediators in Web applications, we are currently in the process of enhancing our analysis with string-specific taint-detection capabilities, in the spirit of [23]. Finally, we plan to extend our coverage of security rules, by investigating ways for statically identifying cross site-request forgery and client-side vulnerabilities [26].

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, Denmark, 1994.

[2] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *S&P 2002*.

[3] R. Bodík, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *PLDI 2000*.

[4] W. Chang, B. Streiff, and C. Lin. Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis. In *CCS 2008*.

[5] P. Cousot and R. Cousot. Modular Static Program Analysis. In *CC 2002*.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *TOPLAS*, 13(4), 1991.

[7] D. E. Denning. A Lattice Model of Secure Information Flow. *CACM*, 19(5), 1976.

[8] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *CACM*, 20(7), 1977.

[9] S. Fink, J. Dolby, and L. Colby. Semi-Automatic J2EE Transaction Configuration. IBM Research Report RC23326, 2004.

[10] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective Typestate Verification in the Presence of Aliasing. In *ISSTA 2006*.

[11] J. S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *PLDI 2002*.

[12] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *S&P 1982*.

[13] C. Hammer, J. Krinke, and G. Snelting. Information Flow Control for Java Based on Path Conditions in Dependence Graphs. In *ISSSE 2006*.

[14] R. Hasti and S. Horwitz. Using Static Single Assignment Form to Improve Flow-insensitive Pointer Analysis. In *PLDI 1998*.

[15] N. Heintze and O. Tardieu. Demand-Driven Pointer Analysis. In *PLDI 2001*.

[16] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *PLDI 1988*.

[17] IBM Rational AppScan Developer Edition (AppScan DE), `http://www.ibm.com/software/awdtools/appscan/developer`

[18] O. Lhoták and L. J. Hendren. Context-Sensitive Points-to Analysis: Is It Worth It? In *CC 2006*.

[19] B. Livshits, J. Whaley, and M. S. Lam. Reflection Analysis for Java. In *ASPLAS 2005*.

[20] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security 2005*.

[21] S. McCamant and M. D. Ernst. Quantitative Information Flow as Network Flow Capacity. In *PLDI 2008*.

[22] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *TOSEM*, 14(1), 2005.

[23] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *WWW 2005*.

[24] A. C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *POPL 1999*.

[25] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *SOSP 1997*.

[26] OWASP, `http://www.owasp.org`.

[27] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. In *ECOOP 2005*.

[28] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL 1995*.

[29] B. G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Languages. In *CC 2003*. Invited Paper.

[30] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *USENIX Security 2001*.

[31] G. Snelting, T. Robschink, and J. Krinke. Efficent Path Conditions in Dependence Graphs for Software Safety Analysis. *TOSEM*, 15(4), 2006.

[32] M. Sridharan and R. Bodík. Refinement-based Context-sensitive Points-to Analysis for Java. In *PLDI 2006*.

[33] M. Sridharan, S. J. Fink, and R. Bodík. Thin Slicing. In *PLDI 2007*.

[34] Stanford SecuriBench Micro, `http://suif.stanford.edu/~livshits/work/securibench-micro`.

[35] T. J. Watson Libraries for Analysis (WALA), `http://wala.sf.net`.

[36] D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *JCS*, 4(2-3), 1996.

[37] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly & Associates, Inc., 3^{rd} edition, 2000.

[38] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *PLDI 2007*.

[39] G. Wassermann and Z. Su. Static Detection of Cross-site Scripting Vulnerabilities. In *ICSE 2008*.

[40] J. Whaley and M. S. Lam. Cloning Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI 2004*.