Jaint: A Framework for User-Defined Dynamic Taint-Analyses based on Dynamic Symbolic Execution of Java Programs

Malte Mues[1], Till Schallau[2], and Falk Howar[2]

[1] Dortmund University of Technology, Germany,
malte.mues@tu-dortmund.de
[2] Dortmund University of Technology, Germany

Abstract. We present JAINT, a generic security analysis for JAVA Web-applications that combines concolic execution and dynamic taint analysis in a modular way. JAINT executes user-defined taint analyses that are formally specified in a domain-specific language for expressing taint-flow analyses. We demonstrate how dynamic taint analysis can be integrated into JDART, a dynamic symbolic execution engine for the JAVA virtual machine in JAVA PathFinder. The integration of the two methods is modular in the sense that it traces taint independently of symbolic annotations. Therefore, JAINT is capable of sanitizing taint information (if specified by a taint analysis) and using multi-colored taint for running multiple taint analyses in parallel. We design a domain-specific language that enables users to define specific taint-based security analyses for JAVA Web-applications. Specifications in this domain-specific language serve as a basis for the automated generation of corresponding taint injectors, sanitization points and taint-flow monitors that implement taint analyses in JAINT. We demonstrate the generality and effectiveness of the approach by analyzing the OWASP benchmark set, using generated taint analyses for all 11 classes of CVEs in the benchmark set.

1 Introduction

Web-based enterprise applications are ubiquitous today and many of these applications are developed in JVM-based languages. The Tiobe index tracks the relevance of programming languages. JAVA leads this ranking consistently (with short periods of being ranked second) for the past 15 years[3]. Moreover, Apache Tomcat is running Web-applications for over 5,000 international companies with a yearly revenue greater than one billion US Dollar each, according to data collected by HG Insights[4]. Therefore, security of Java Web-applications is of critical importance and attacks on them are reported every single day[5]. Though there is no publicly available data on the exact distribution of breaches across different

[3] https://www.tiobe.com/tiobe-index/
[4] https://discovery.hgdata.com/product/apache-tomcat
[5] https://www.cvedetails.com/vulnerabilities-by-types.php

programming languages. Based on the market share of Java in the realm of enterprise applications, one can assume that a significant fraction of the reported breaches exploits vulnerabilities of JVM-based Web-applications.

Many of the vulnerabilities tracked in the Common Vulnerability and Exposures (CVE)[6] list pertain to the flow of information through a program from a (potentially) malicious *source* to a protected *sink*. In modern Web-applications, such flows almost universally exist as these applications receive inputs from (untrusted) users and, e.g., store these inputs in (protected) databases. These inputs should pass *sanitizing* methods, e.g., for escaping of specific characters in a textual input or prepared and safe statements. Otherwise, attackers might use these inputs maliciously to inject commands into SQL statements in an attack.

Taint analysis is a well-established technique for analyzing the data flow through applications: Inputs are tainted and taint is then propagated along the data flow. Critical sinks (i.e. databases) are monitored by taint guards ensuring that no tainted data values reach the sink (c.f. [1,3,14,23,24,25]). Otherwise a security vulnerability is detected. Classically, taint analysis is either implemented as a static analysis, over-approximating flow of taint (c.f. [27]), or as a dynamic analysis, under-approximating taint flow by observing concrete program executions (c.f. [12]). The literature distinguishes data-flow taint, i.e., taint that propagates from right to left sides of assignments, and control-flow taint, i.e., taint is propagated through branching conditions to assignments in executed branches (c.f. [24]). One can observe a close similarity to symbolic execution [25]: (Data-flow) taint propagates like symbolic values, and (control-flow) taint captures path constraints of execution paths. However, this close similarity has not yet been fully leveraged as the basis for an integrated analysis for Java.

In this paper, we present Jaint, a framework for finding security weaknesses in Java Web-applications. The framework combines dynamic symbolic execution and dynamic taint analysis into a powerful analysis engine. This analysis engine is paired with a domain-specific language (DSL) that describes the concrete taint analysis tasks, Jaint executes during one analysis run. It is the first framework exploiting concolic execution for the dynamic but exhaustive exploration of execution paths in Java Web-servlets while maintaining explicit multi color taint marks on data values. This multi color taint allows the specification of multiple taint analyses run in parallel tracking data flow from malicious sources to protected sinks and monitoring potential security vulnerabilities. Moreover, as taint marks and symbolic values are separate annotations, the framework supports sanitization definitions on a taint color base making it more precise than previous work using symbolic annotations as taint marks [11]. The combination of dynamic symbolic execution and dynamic taint analysis results in greater precision than can be achieved with classic static taint analysis methods that are insensitive to most conditions on control flow. Moreover, for many of the identified vulnerabilities, our analysis can produce request parameters that exhibit a found vulnerability in a servlet. In contrast to purely dynamic taint analysis techniques, our approach is exhaustive given that dynamic symbolic ex-

[6] https://cve.mitre.org

ecution terminates [18]: it generates a set of request parameters for every feasible execution path.

We have implemented JAINT as an extension of JDART [18], a dynamic symbolic execution engine for JAVA, and on top of JAVA PATHFINDER [13], a software model checker for JAVA that is based on a custom implementation of the JAVA virtual machine (JPF-VM). JAINT's DSL for defining concrete taint analyses (i.e., sources, sanitization methods and sinks) is designed on the basis of the Meta Programming System (MPS). JAINT's implementation is publicly available [20]. We evaluate JAINT on the OWASP benchmark suite[7], the current industrial standard for comparing analysis approaches for JAVA Web-applications. All 11 CWEs in the OWASP benchmark suite can be specified in our domain-specific language and JAINT analyzes the OWASP benchmark suite with a false negative rate of 0% and a false positive rate of 0%, identifying all security vulnerabilities.

Related Work. Schwartz et al. [25] describe a formal theory for dynamic taint propagation and discuss challenges in the implementation of an analysis combining dynamic symbolic execution and dynamic taint analysis. Their focus is mostly on memory representation problems for running the symbolic analysis in a programming language that allows pointer arithmetic. Due to the design of the JAVA virtual machine, these concerns are not relevant when analyzing JAVA byte code. The formalization of taint analysis by Schoepe et al. [24] stresses the importance of a clear division of data-flow and control-flow based taint propagation. From our point of view, this observation supports a separation of analysis methods: dynamic taint analysis and dynamic symbolic execution: Dynamic tainting tracks information following the data flow path, e.g., through instrumentation (c.f. [7,8,12,16,17,21,26,28]). Dynamic symbolic execution can be used for controlling the program execution path with external inputs ensuring exhaustive exploration of all paths.

Haldar et al. [12] presented a dynamic tainting mechanism for JAVA propagating the dynamic taint along a single path. JAINT's advantage over the approach of Haldar, is the integration of single path propagation with dynamic symbolic execution [2,6] for exhaustive path enumeration.

For C programs, Corin and Manzano [10] describe the integration of taint analysis into KLEE [5]. Their work is limited to propagation of single color taint and do not show, how different analyses can be run on top of the taint propagation, while we demonstrate how multi color taint can be used to analyze the OWASP benchmark. It seems some work has been started on KLEE-TAINT[8] for a more sophisticated taint analysis in KLEE combining symbolic execution with taint, but the approach requires to rewrite the C program to inject taint assigning methods and taint checks for the analysis. JAINT integrates the taint analysis without any modifications of the bytecode as taint injection and taint monitoring is computed in the virtual machine and not as part of the binary. Both approaches require a driver for the dynamic symbolic execution part.

[7] https://github.com/OWASP/Benchmark
[8] https://github.com/feliam/klee-taint

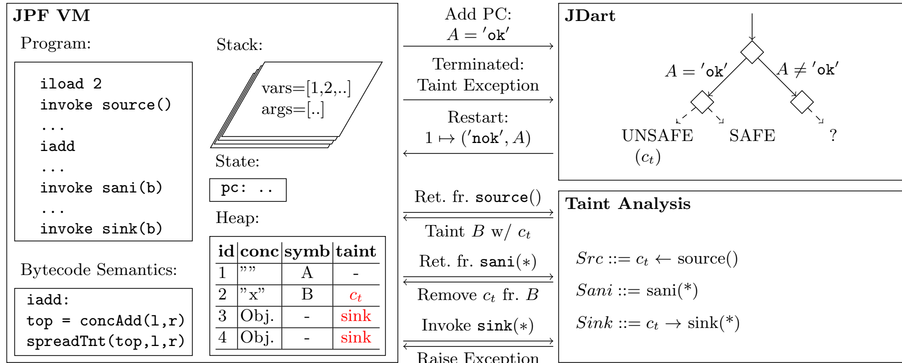

Fig. 1. Software Architecture of the implemented Vulnerability Analysis.

Several strategies for the implementation of taint models and taint propagation have been proposed: They range from integrating the taint check into the interpreter [22] to a complete taint propagation DSL integrating the taint analysis into the program [9]. In between are the flavors of integrating the taint check into the compiler [16,28] or into an execution environment [3,7,8,15,21]. We consider binary instrumentation as part of execution environment modification. The advantage of a DSL integrated into the program is that the execution environment and tool chain stay untouched. JAINT mixes two of those proposals. For the taint propagation, we modified the program interpreter, in our case the JPF-VM. In addition, we defined a DSL that allows to describe in which places taint should be injected, sanitized and checked during execution. As a consequence, the concrete taint injection does not require a modification of the program. Our DSL only describes the analysis and not the taint propagation. Hence, it is a different style of DSL than the one proposed by Conti and Russo [9].

Outline. The paper is structured as follows: We present our analysis framework JAINT in Section 2 and discuss the proposed domain-specific language for expressing concrete taint analyses in Section 3. Section 4 details results from the evaluation of the integrated analysis on the OWASP benchmark suite. We present conclusions and discuss directions for future research in Section 5.

2 Taint Analysis with Jaint

JAINT integrates dynamic symbolic execution and dynamic tainting in a single analysis framework. It is built on top of the JPF-VM. Figure 1 illustrates the interplay between the dynamic symbolic execution handled by JDART [18], the taint analysis and the JPF-VM.

The virtual machine of JAVA PATHFINDER provides several extension mechanisms that JAINT uses for the implementation of the analysis: *VM events*, *bytecodes*, *peers*, and *heap annotations*. *Heap annotations* are a mechanism for annotating objects on the heap with meta-information. *VM events* are hooks an

analysis can use to collect or modify meta-information during execution. The JPF-VM allows to replace *bytecode instructions* or extend them to collect information or trace meta-information during symbolic execution. *Peers* can replace implementations of (native) library functions.

The dynamic symbolic execution uses *bytecode semantics* and *peers* for recording symbolic path constraints as shown on the top-most arrow from the JPF-VM box to the JDart box in Figure 1. Bytecode semantics for symbolic execution and taint analysis are similar: while in the one case the result of operations is computed and maintained symbolically based on the symbolic annotations on operands, in the other case operations propagate existing taint annotations on operands to results of operations. E.g., the implementation of the `iadd` bytecode pops two integers including potential symbolic and taint annotations from the stack, performs a concrete addition, computes a symbolic term representing the result (only in case one of the integers was annotated symbolically), propagates taint from the integers to the result, and pushes the result and annotations back onto the stack. Symbolic values are used in path constraints (recorded on branching bytecode instructions) that accumulate in the constraints tree (upper right corner of the figure). The JDart concolic execution engine interacts with the virtual machine by placing concolic values (concrete values with symbolic annotations as shown in the heap table of the JPF-VM in Figure 1) on the heap to drive execution down previously unexplored paths in the constraints tree. In the current version of Jaint, bytecode implementations do not remove symbolic annotations or taint (e.g., on multiplication with constant 0). Such behavior could, however, be implemented easily. JDart has the same limitations that symbolic execution has in general: recursion and loops are only analyzed up to a (configurable) fixed number of invocations (iterations, respectively).

The dynamic taint analysis is built around *VM events*. Listening on VM events (returns from methods), the taint analysis injects or removes taint from objects on the heap. E.g., a sanitization interaction between the analysis and the heap is shown in the lower part of Figure 1. The method exit event for the sani method interacts with the taint analysis and removes the c_t taint mark from the heap object with id 2 in this analysis. The interaction is represented by the Ret. fr. sani(*) arrow. In addition, the taint analysis will check taint annotations on heap objects and primitive values before entering methods. Those checks are used for monitoring tainting of protected sinks. In combination with the taint propagation in the *bytecodes*, the *VM events* implement the complete multi-colored taint analysis. In the remainder of this section, we discuss the central ideas of the interplay of the internal components of the implementation along a small example and provide a high-level overview of the analysis.

2.1 Integration of Symbolic Execution and Taint Analysis.

Let us assume, we want to analyze method `foo(String a, String b)` from the code snippet shown in Listing 1.1. In particular, we want to check that no data flows from malicious method call `source()` to the protected method `sink()` unless the data is sanitized by passing through method `sani()`. This specifies

```
1  static void foo(String a,
       String b){
2    if (a.equals("ok"))
3      b = sani(b);
4    sink(b);}
5  public static void main
6      (...){
7    String a =
8      Verifier.
9        symbString("","A");
10   String b = source();
11   foo(a,b);}
12 public String source(){
13   return Verifier.
14     symbString("x","B");}
```

Listing 1.1. Code Example: Parameter `b` of method `foo` is only sanitized if parameter `a` has value `"ok"`.

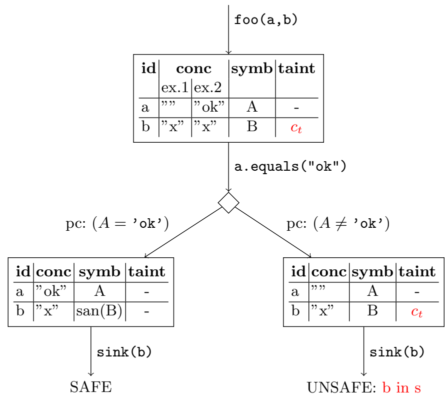

Fig. 2. Tree of concolic executions of method `foo`. Nodes show snapshots of the heap with annotations of taint and symbolic values.

the taint property denoted with taint color c_t in this example and confirms the configurable part we have to write down in JAINT's taint DSL. It is visualized in the lower right part of Figure 1. We will first describe how dynamic symbolic execution is applied to the example followed by the taint integration.

Dynamic Symbolic Execution. Dynamic symbolic execution (DSE) is a dynamic analysis technique in which a program is executed with concrete data values while constraints that are imposed on these values along a single execution are recorded as path constraints. Recorded symbolic constraints can then be used as a basis for finding new concrete values that drive execution along previously unexplored program paths. The program execution is restarted with the new concrete values. This is represented in the top right corner of Figure 1.

As DSE is a dynamic technique, a driver for the method under analysis is required. For our example, this can be seen in Listing 1.1. The `main(...)` method is used as a test driver for analyzing method `foo(...)`: in the listing, we create two variables of type String with values `""` and `"x"` and instruct the analysis to annotate these Strings with symbolic values A and B. These annotations are tracked, modified, and propagated by the symbolic part of the underlying execution engine. The state of the analysis is visualized in Figure 2: the tree represents executions of `foo(...)` with different sets of concrete values. The nodes of the tree visualize the state of the heap, including annotations to heap cells that keep track of symbolic values and taint.

Let us first focus on symbolic values. Initially, variables `a` and `b` are marked symbolically and contain the original concrete values (column *ex.1* in the root node of Figure 2). Execution with these values proceeds down the right path in the tree as `a` does not equal `"ok"`. Path constraint $A \neq 'ok'$ is recorded. After execution terminates, the analysis uses the recorded constraint for generating a new value for `a` (`"ok"` in this case) that drives execution down the unexplored

path, represented by the left leaf of the execution tree. On this path, statement `b=sani(b)` is executed and the symbolic value of `b` is updated accordingly to symbolic value $san(B)$. After execution of the path the tree is complete, i.e., all feasible method paths through `foo(...)` have been explored and concolic execution halts. Next, we will briefly discuss, how we integrate the taint analysis along the paths discovered by dynamic symbolic execution.

Dynamic Taint Analysis. We check if the defined property is violated on some execution path by tainting relevant data values and tracking propagation of taint (visualized in the last column of the tables that represent the state of the heap in Figure 2). The taint specification interacts with the JPF-VM using JPF's listener concept for VM events. The taint analysis subscribes to VM events, such as method invocations and method exits. If such an event is triggered, e.g., a method invocation, the generated listener checks whether the invoked method is part of the taint specification. If this is the case, code for injecting taint, removing taint or checking taint gets integrated into the execution. The JPF-VM allows to extend objects with annotations directly on the heap. This is used for adding the taint marks to the objects on the heap. The JPF-VM takes care to track those annotations. If JAVA bytecodes operates on none heap objects as primitive data types, the implemented *bytecode semantics* for symbolic execution get extended with taint propagation semantics.

As the `main` method used as a driver for running `foo` only initializes `b` with a call of `source`, only `b` will be tainted as *malicious source* with the c_t taint color. As there is no call to `source` in any of the assignments to `a`, `a` never becomes marked with the c_t taint color. Object s gets annotated as a *protected sink*. During the first execution (along the right path in the tree), the taint marker on `b` is not removed and a connection from source to sink is established upon invocation of `sink(b)`. The analysis reports that on this path the property that "no data is allowed to flow from the malicious source to the protected sink" is violated. Those taint exceptions directly abort the DSE along a path and trigger the start of the next path. The second execution of `foo(...)` proceeds along the left path in the figure. In this case, statement `b=sani(b)` is executed and the taint marker is removed from `b`. The analysis concludes that the path is safe to execute and exits without any error along this path. After the combined taint analysis and DSE terminates, akin to other dynamic analysis methods, we can produce a concrete witness that exhibits the security vulnerability on the first execution path and triggers our taint monitor. At the same time, we are confident that all feasible program paths (within the cone of influence of the symbolic variables) were analyzed.

In the context of taint analysis, control-flow dependent taint is often discussed as a problem for precise taint analysis. In the scope of the method `foo`, both parameters are external parameters, but only `b` becomes tainted. In contrast `a` is the parameter influencing the control-flow in line 2 of Listing 1.1. As both are external parameters, we model both of them symbolically, and as the parameter `a` influences the if condition, this example further demonstrates, how symbolic execution ensures the control-flow dependent value propagation even if

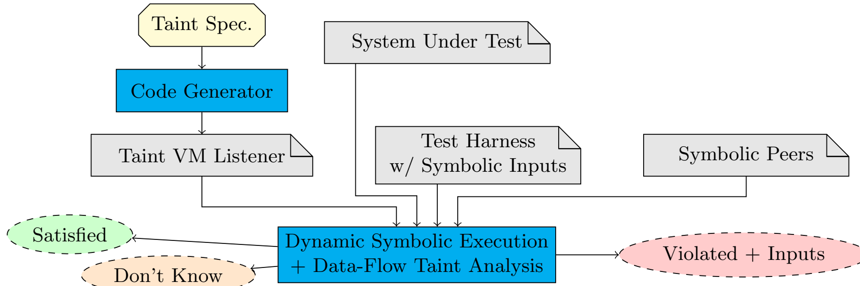

Fig. 3. JAINT combines user-defined dynamic taint-analyses with dynamic symbolic execution of Java programs. Taint DSL Specification (hexagon) and source code artifacts (documents) are compiled and analyzed (rectangles); verdicts shown as ellipses.

the parameter influencing the control-flow is not part of the taint specification. In contrast to pure dynamic tainting approaches, JAINT does this without any over-approximation. Due to the DSE, the analysis keeps track of the different branches and reports precisely which branches eventually violate a property and on which branches the property holds. This way, JAINT integrates DSE and dynamic taint analysis in a single framework splitting the tasks of data-flow tainting and control-flow tainting between the symbolic model in the DSE and the dynamic taint analysis. An appropriate symbolic model takes care of eventual effects from external parameters on the control-flow. The dynamic taint analysis only propagates the different taint colors along the current execution path, checks monitors and eventually removes taint marks wherever required. We will summarize this workflow below.

2.2 User-defined Taint Analyses with Jaint

As shown in the previous subsection, the JAINT framework combines dynamic taint analysis with DSE. Figure 3 shows the analysis workflow. The cyan boxes are tools built to establish the JAINT workflow. In the center of the lower part is the analysis engine running dynamic symbolic execution and the data-flow taint analysis. An analysis will lead to one of three verdicts: no exploitable vulnerability exists (*Satisfied*, green ellipse), a vulnerability and an exploit are found (*Violated + Inputs*, red ellipse), the instance is undecidable due to an intractable symbolic constraint or due to exhausted resources (*Don't Know*, orange ellipse). In the upper half, the required inputs are represented: For some system under test, a test harness that defines the scope of symbolic analysis, and a set of symbolic peers are provided for the dynamic symbolic execution from the center to the right side. On the left, the required taint inputs are represented. The user provides a taint specification in JAINT's taint DSL. The DSL code generator part of the framework generates the required VM listeners for the taint analysis, which are passed along to the main tool.

JAINT describes taint flow properties in its own domain specific language (DSL), which is described in detail in Section 3. The DSL allows to specify different taint analyses which are all executed in parallel during the execution

of the program as long as they all use a different taint color. The DSL is built on top of MPS and Jaint runs a code generator (the upper left cyan box) to synthesize the required VM listeners working as taint monitors for each of the specified taint analyses.

A test harness defines the symbolic parts of the system under test and therefore the analysis scope. In the test harness certain inputs are modeled symbolically, while others might remain concrete values. For analyzing Java Web-applications, we constructed symbolic String values as part of JDart along with a symbolic peer for String operations as an example for such peers. The symbolic peer models String operations on the basis of symbolic byte arrays. Those byte arrays are logical encoded in the bit-vector theory for constraint solving. The String model is robust enough for the evaluation of the OWASP benchmark and performed well in the Java track of SV-Comp 2020 (c.f. [19]). We released it open source as part of the JDart version[9] used for SV-Comp. Balancing symbolic and concrete parts of the system state space is the key factor for analysis performance. Unnecessary large state spaces waste resources, while a too small state space might harm the analysis verdict by cutting away relevant paths.

The analysis environment can be modeled using symbolic peers in JDart. Apart from the symbolic peer modeling the symbolic operations of Strings, we can use such peers as well to mock the behavior of an interface or model symbolically the execution of an external resource. As an example, in the case of SQL injection analyses, a model for `java.sql.Statement` is required to describe the taint flow appropriate. Similar, we defined symbolic peers for other system resources as the file system or the LDAP API. This allows us to analyze the OWASP benchmark. In the same way, a test harness might skip relevant parts of the execution, a symbolic peer might threat the analysis, if the environment model is an under approximation.

Jaint allows to split the task of establishing an effective security analysis in two domains. A program analysis engineer might model the relevant resources for dynamic symbolic execution, while a security engineer can define the security properties. Next, we will explain the DSL Jaint offers for the security engineer.

3 A DSL for Defining Taint Analyses

In Jaint, concrete taint analyses are specified by means of a domain-specific language (DSL). Taint generators, sanitizers, and monitors are generated from specifications. While code generation is currently tailored towards JPF/JDart, it could easily be adapted to generate code for other verification frameworks. The triggers and conditions for generating and removing taint as well as for raising alarms that can be specified in the language are generic (for Java programs). Concrete analyses are however particular to the libraries and frameworks used by a program under analysis: these libraries have APIs and methods in these APIs may be sources or sinks for taint flow. In this section we present this domain-

[9] https://github.com/tudo-aqua/jdart

specific language along with some examples of concrete taint analyses motivated by CWEs in the OWASP benchmark suite.

Specification of Taint Analyses. Our DSL enables the definition of custom taint analyses. An analysis is specified by a tuple $\langle \mathit{Src}, \mathit{Sani}, \mathit{Sink}\rangle$, consisting of malicious sources (Src), sanitization methods (Sani), and protected sinks (Sink). Each of these elements specifies signatures of methods that, upon invocation or return, should trigger either marking a return attribute, removing the mark from an object, or checking for marked parameters, respectively. The syntax of the DSL is defined in (1) as BNF. Constant syntax elements are highlighted with gray boxes.

$$
\begin{aligned}
\mathit{Generation} &::= \mathit{Analysis}(,\ \mathit{Analysis})^* \\
\mathit{Analysis} &::= (\mathit{Src})^*, (\mathit{Sani})^*, \mathit{Sink} \\
\mathit{Src} &::= \texttt{Src::=}\ (\mathbf{id}|\mathbf{id}^+)\ \leftarrow\ \mathit{Signatures} \\
\mathit{Sani} &::= \texttt{Sani::=}\ \mathit{Signatures} \\
\mathit{Sink} &::= \texttt{Sink::=}\ (\mathbf{id}|\mathbf{id}^+)\ \rightarrow\ \mathit{Signatures} \\
\mathit{Signatures} &::= \mathit{ExtSignature}\ (,\ \mathit{ExtSignature})^* \\
\mathit{ExtSignature} &::= \mathit{Signature}\ (\texttt{.<}\mathbf{class}\texttt{>}\mathit{Method}\texttt{(}\mathit{Parameter}\texttt{)})^* \\
\mathit{Signature} &::= \texttt{(_:}\mathbf{class}\texttt{).}\mathit{Method}\texttt{(}\mathit{Parameter}\texttt{)} \\
\mathit{Method} &::= (\mathbf{method}|\texttt{<init>}) \\
\mathit{Parameter} &::= (\mathbf{param}|\mathbf{param}^+|\mathit{ValueCheckExp}) \\
\mathit{ValueCheckExp} &::= (\mathit{ValueCheck}\ ((\texttt{and}|\texttt{or})\ \mathit{ValueCheck})^* \\
\mathit{ValueCheck} &::= (\mathit{ParamValue}\ \texttt{has}\ (\texttt{not})^*\ \texttt{value}\ \mathbf{value}) \\
\mathit{ParamValue} &::= (\mathbf{type\ param}|\mathbf{class\ id}\ \texttt{:}\ \mathbf{id}\ \texttt{.}\ (\mathit{Method}\texttt{()}|\mathbf{param}))
\end{aligned} \tag{1}
$$

To allow multiple parallel taint analyses the top-level *Generation* allows the containment of multiple *Analysis* elements. Each analysis is based on the tuple $\langle \mathit{Src}, \mathit{Sani}, \mathit{Sink}\rangle$ of which the first two are optional. For some weaknesses sanitization methods are not available and therefore neglectable. Taint analyses which depend on specific argument values of protected sinks and not on taint flow (c.f. example in (5)), do not contain source definitions. Each weakness has its unique identifier (or color) declared by **id** in the *Src* and *Sink* declaration. We use $\mathbf{id}^+$ in *Src* to indicate that fields and nested objects of the returned object are tainted additionally. With the usage of $\mathbf{id}^+$ in *Sink* we indicate that not only immediate taint of some parameter value has to be checked when invoking a sink but also taint of reachable objects from the parameter. The expression **class** matches fully qualified class names and **method** is an expression for matching method names. We allow $*$ as a wildcard for an arbitrary sequence of symbols. With <init> we restrict the method check to only consider constructors of the declared class. The expression **param** matches names of parameters. We use the empty String for methods without parameters and $*$ for arbitrary parameters. With $\mathbf{param}^+$ we define that, instead of the return attribute, the declared

parameter will be tainted. To also conveniently describe trigger conditions on the concrete values passed into sink methods: **param** may contain expressions like `(int p has value 5)` for specific parameter values or `(Object o : o.var has value 5)` for field accesses. It indicates that a taint alarm should be raised in case of a method invocation with a field value of `5` for the field `var` of parameter `o` which is of type `Object`. For building complex expressions we allow composite boolean expressions with the keywords `and` and `or` e.g., `(param has value a) or (param has not value b)`.

To express a sequence of method calls that constitute a protected sink, additional information has to be provided (c.f. *ExtSignature*). For that, **<class>** specifies the type of the returned variable on which taint should be checked.

Example. To clarify this behavior and give an example, we further describe parts of the *Cross Site Scripting* weakness analysis with a code snippet in Listing 1.2 and corresponding DSL snippet in (2). Cross site scripting (CWE 79[10]) occurs when data (e.g., JavaScript code) from an untrusted source is added to the Web-page and served to other users without proper sanitization.

$$\begin{aligned}
Src &::= \; xss^{+} \leftarrow (_ : *\texttt{HttpServletRequest}).\texttt{get}*(*) \\
Sani &::= \; (_ : \texttt{org.apache.commons.lang.StringEscapeUtils}) \\
&\qquad .\texttt{escapeHtml}(*), \\
&\quad\; (_ : \texttt{org.owasp.esapi.ESAPI}).\texttt{encodeForHTML}(*), \\
&\quad\; (_ : \texttt{org.springframework.web.util.HtmlUtils}) \\
&\qquad .\texttt{htmlEscape}(*) \\
Sink &::= \; xss^{+} \rightarrow (_ : \texttt{javax.servlet.http.HttpServletResponse}) \\
&\qquad\qquad .\texttt{getWriter}().\texttt{<java.io.PrintWriter>}*(*)
\end{aligned} \tag{2}$$

```
1  public void doPost(HttpServletRequest request,
2  HttpServletResponse response) {
3    ...
4    String param = "";
5    java.util.Enumeration<String> headers = request.
         getHeaders("Referer");
6    if(headers != null && headers.hasMoreElements()){
7      param = headers.nextElement();
8    }
9    ...
10   response.getWriter().format(..., param, ...);}
```

Listing 1.2. Code Example: Cross side scripting vulnerability in servlet `BenchmarkTest00013` of the OWASP benchmark suite (omissions for improved readability).

[10] https://cwe.mitre.org/data/definitions/79.html

In line 5 data is read from the `HttpServletRequest` object. According to the specification in (2) this classifies as reading from a malicious source. Therefore, the returned value is annotated with a taint marker of type xss during concolic execution. At the same time, all elements contained in the returned `Enumeration<String>` are tainted as well, as the non-immediate taint flag is set (c.f. xss^+ in *Src* of (2)). This is necessary as the `param` variable is set by getting the next element with the `nextElement()` method in line 7. Without implicit taint propagation the taint information would be lost at this point. From line 7 code is executed that eventually manifests a protected sink for taint of type xss: In line 10 the condition for the protected source is matched. The `PrintWriter` object returned by the `getWriter()` method is flagged to signalize possible future taint violations (c.f. xss^+ in *Sink* of (2)). Calling the `format(...)` method in the same line first checks the called object if it is flagged. Here, this is the case, so the real taint check on the parameter `param` can be executed. Since the variable is marked as xss-tainted, the analysis will correctly raise an alarm.

4 Evaluation

We evaluate Jaint by applying the framework on the OWASP benchmark. The OWASP benchmark suite (version 1.2) consists of 2,740 servlets that are categorized into 11 CWE classes. We aim to answer the following three research questions during the evaluation:

RQ1: Is Jaint's Taint-DSL expressive enough for specifying security analyses? We approach this question by specifying analyses for the 11 CWEs in the OWASP benchmark and by discussing briefly comparing the expressiveness to the specifications provided by other tools.

RQ2: Does the combination of dynamic symbolic execution and dynamic tainting improve precision over the state of the art in security analysis? We approach this question by comparing Jaint's precision to industrial tools.

RQ3: How expensive is the application of Jaint, especially compared to existing tools? We approach this question by analyzing Jaint's runtime.

We begin by detailing some taint analyses (**RQ1**), before presenting results from a series of experiments (**RQ2** and **RQ3**).

4.1 Taint Analyses for OWASP CWEs

The CWEs included in the OWASP benchmark suite, broadly fall into two classes of properties: *Source-to-Sink-Flow* and *Condition-on-Sink* properties. The first class is the main domain of taint analysis and requires the flow of taint marks from a source to a sink. The second group checks a concrete value for a concrete assignment at a certain point of time in the execution flow. While this is not the typical strength of dynamic tainting, we can still check those properties easily with Jaint, using only sink conditions. The *Source-to-Sink-Flow* group comprises 8 CWEs: Path Traversal Injection (CWE 22), Cross Site Scripting

(CWE 79), SQL Injection (CWE 89), Command Injection (CWE 78), LDAP Injection (CWE 90), Weak randomness (CWE 330), Trust Bound Violation (CWE 501) and XPath Injection (CWE643). The *Condition-on-Sink* group comprises 3 CWEs: Weak Crypto (CWE 327), Weak Hashing (CWE 330) and Secure Cookie (CWE614). In the remainder of this subsection, we detail the specifications for three of the CWEs.

SQL Injection. The structured query language (SQL) is a fourth-generation language and SQL queries are constructed as Strings in Java programs. When this is done manually in a servlet, parameters of the HTTP request are typically integrated into the SQL query through String concatenation. Without proper String sanitization before the concatenation, this allows for a so-called SQL injection (CWE 89[11]), i.e., the resulting SQL query can be manipulated by injecting additional SQL statements into the query String.

It is well known that proper sanitization of parameters is hard and SQL injection vulnerabilities are best prevented by using prepared statements instead of building queries manually. Consequently, the OWASP benchmark assumes that there are no adequate sanitization methods for this weakness. The specification of the corresponding taint analysis is shown in (3). We consider the `sql` parameter of any method as a protected sink in some of the interfaces from the `java.sql` and `org.springframework.jdbc` packages.

$$\begin{aligned} Src ::= \;& sqli \leftarrow (_ : *\texttt{HttpServletRequest}).\texttt{get}*() \\ Sink ::= \;& sqli \rightarrow (_ : \texttt{java.sql.Statement}).*(\texttt{sql}), \\ & \qquad\; (_ : \texttt{java.sql.Connection}).*(\texttt{sql}), \\ & \qquad\; (_ : \texttt{org.springframework.jdbc.core.JdbcTemplate}).*(\texttt{sql}) \end{aligned} \tag{3}$$

Command Injection. Command injection (CWE 78[12]) attacks are similar to the injection attacks discussed above. However, instead of injecting statements into some query language, these attacks aim at injecting commands into a shell, i.e., into a command that is executed as a new process. (4) specifies the corresponding taint analysis. Methods that match patterns `Runtime.exec(*)` and `ProcessBuilder.*(command)` are considered protected sinks.

$$\begin{aligned} Src ::= \;& cmdi^{+} \leftarrow (_ : *\texttt{HttpServletRequest}).\texttt{get}*() \\ Sink ::= \;& cmdi \;\; \rightarrow (_ : \texttt{java.lang.Runtime}).\texttt{exec}(*), \\ & \qquad\quad (_ : \texttt{java.lang.ProcessBuilder}).*(\texttt{command}) \end{aligned} \tag{4}$$

Secure Cookie Flag. A secure cookie flag (CWE 614[13]) weakness exists in a servlet when a cookie with sensitive data is added to the response object without setting the secure cookie flag (setting the flag forces Web-containers to use HTTPS communication). The corresponding taint analysis is specified in

[11] https://cwe.mitre.org/data/definitions/89.html
[12] https://cwe.mitre.org/data/definitions/78.html
[13] https://cwe.mitre.org/data/definitions/614.html

(5). When a cookie is added to the request, the analysis checks that the secure flag is set.

$$\mathit{Sink} ::= \ * \rightarrow (_ : \texttt{javax.servlet.http.Response}) \\ \texttt{.addCookie(cookie c : c.getSecure() has value false)} \quad (5)$$

Please note that the specification of the trigger condition in (5) is more complex as in the case of SQL injection as we have to express a condition on a field of an object.

Summarizing, the expressiveness of JAINT's taint DSL was sufficient for expressing the CWEs in the OWASP benchmarks.

Comparing the expressiveness to other tools that provide performance data for the OWASP benchmark suite, at least `SBwFindSecBugs` (cf. next subsection) uses an approach similar to JAINT: Method signatures and parameter positions are used for specifying taint sources and sinks. JAINT's taint DSL is more precise and more expressive than `SBwFindSecBugs` by allowing custom sources and sinks per analysis, by allowing to express that an object obtained from a sink becomes a sink as well, and by allowing to specify constraints on parameter values.

Together, these two results provide some confidence in the expressiveness of JAINT's taint DSL (**RQ1**). Of course, there is effort associated with specifying custom sources and sinks for analyses and for analyzed APIs but developers of tools have to spent effort on definition of taint sources and sinks anyway and (in the long run) all tools can profit from more detailed specifications.

4.2 Experimental Performance Analysis

In this subsection we describe the setup used to evaluate our framework on the OWASP benchmark and compare JAINT with the other tools based on precision (**RQ2**). We will show that JAINT successfully beats existing research approaches in precision and discusses JAINT's runtime performance compared with other noncommercial tools (**RQ3**).

Setup. JAINT's taint DSL and a corresponding code generator are implemented in the Meta Programming System (MPS) [14]. We used the implementation to generate monitors and taint injectors together with sanitization points for the 11 CWEs in the OWASP benchmark. We have written a generic *HttpServlet* driver for executing each of the servlets. For the DSE, we modeled all data read from a request object symbolically as it is the untrusted input read from the web. This ensures that we explore all paths across a *HttpServlet* that might be influenced through a request by a malicious attacker, as the OWASP benchmark does not contain another untrusted source. In addition, we provided suitable symbolic peers for the used libraries that require environment interaction. For example, the analysis of a test case related to a potential SQL injection weakness (CWE 89) requires a suitable abstraction for the database interaction involved in the test case. In the same way, we provided abstractions for file system access,

[14] https://www.jetbrains.com/mps/

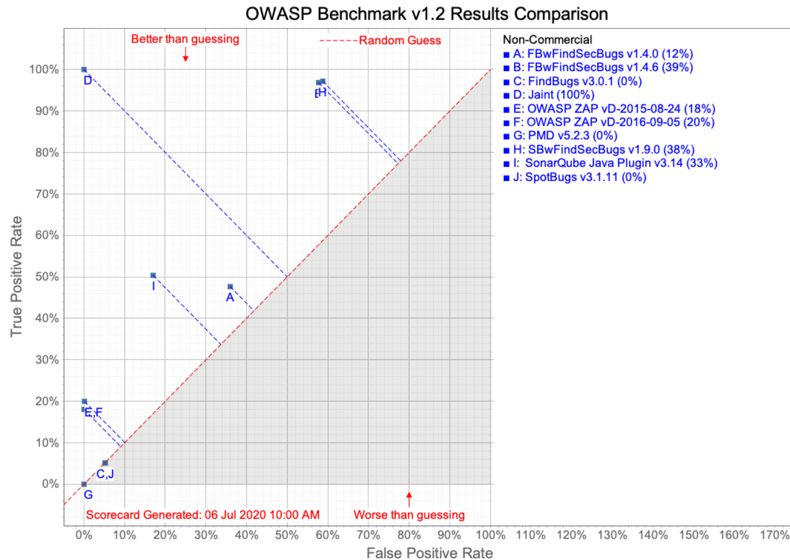

Fig. 4. Comparing our results of the generated DSL specification with results from related work. The percentag is computed as follows: $TruePositiveRate - FalsePositiveRate$.

LDAP related implementations and XPath libraries. Those libraries are required to enable the DSE to process the OWASP benchmark and are not related to the taint analysis. Using the mentioned driver together with the peers, we analyzed every servlet with JAINT and 11 taint colors enabled. All experiments were conducted on an Intel(R) Core(TM) i9-7960X machine with 128 GB RAM and an SSD hard drive, running Ubuntu with kernel 5.4.0-33 (x86_64).

Precision. Over all categories of CWEs, JAINT achieves the maximum possible precision of 100% true verdicts and 0% false verdicts, i.e., it finds all vulnerabilities in the benchmarks and does not raise a single false alarm. It outperforms the other tools for which performance is reported to OWASP by a big margin: the scorecard[15] that is provided by the OWASP benchmark suite is shown in Figure 4 (JAINT is marked D). The other tools in the card that perform better than random guessing fall into three groups: over-approximating tools with (close to) no false negatives but a high rate of false positives (B, H), under-approximating tools with no false positives but high numbers of false negatives (E, F), and a third group (A, I) with high rates of false positives and false negatives.

For some commercial tools, performance data is not included in the OWASP scorecard, and hence not included in our evaluation, but can be found in promotional statements on the web pages of tool vendors. Most notably, Hdiv's and Contrast's IAST tools also report 100% true verdicts and 0% false positives on the OWASP benchmark suite. It seems, however, that IAST is a dynamic analy-

[15] Score computation: https://owasp.org/www-project-benchmark/#div-scoring

sis and — in contrast to JAINT— cannot guarantee complete exploration. Julia, a commercial static analyzer using abstract interpretation, is reported to achieve a 90% score in the benchmark, which is a very good score but still includes 116 false positive results [4]. So far, JAINT is the only tool that can provide completeness guarantees (within the limits of symbolic execution), while performing precise security analysis (**RQ2**).

Performance. We compare the runtime of JAINT to the static code analysis FindSecBugs (H) as performance data for the commercial IAST tools and for Juliet could not be obtained. FindSecBugs needs 62 seconds (average over 3 runs with no significant variance) for analyzing the OWASP benchmark suite, averaging $23\,ms$ per task. JAINT, in comparison, needs 1 879 seconds (average over 3 runs with 5 seconds std. deviation), i.e., an average of $686\,ms$ per task. While this constitutes a thirtyfold increase in runtime, the absolute runtime still allows to run JAINT as part of a CI pipeline, especially since the reported runtime is obtained through single-threaded and sequential task processing, leaving space for runtime optimization through parallelization (**RQ3**).

5 Conclusion

In this paper, we have presented JAINT, a framework for analyzing JAVA Web-Applications. JAINT is the first working proof-of-concept for combining dynamic symbolic execution and dynamic multi-colored taint analysis in JAVA. Our approach strictly separates symbolic annotations and colored taint markers used for a security analysis. This enables analysis of arbitrary sanitization operations while dynamic symbolic execution is still capable of exploring the symbolic state space. JAINT uses JDART and the JPF-VM, as the dynamic symbolic execution engine of JAVA byte code.

We extended JDART with environment models that represent parts of the JAVA standard library and provide symbolic summaries and model taint propagation for some of the interfaces in the JAVA library, e.g., classes from the *java.sql* package. For the specification of security properties that JAINT should check, we provide a domain-specific language (DSL) based on the Meta Programming System (MPS). Custom components for checking of specified properties are generated from specifications (i.e., VM event listeners that can be plugged into the JPF-VM for taint injection, taint sanitization, and taint monitoring).

The evaluation of the approach on the OWASP benchmark shows promising results: the implementation achieves a 100% score and 0% false positive results, outperforming all other research tools for which performance data on the OWASP benchmark suite is available. Basis for the evaluation was the specification of taint analyses for the 11 classes of CWEs in the OWASP benchmark suite using the proposed taint DSL. Specifications were derived by researching CWEs and by inspection of the code of the OWASP benchmark suite and the JAVA class library. As these taint analyses are specified using our DSL we could demonstrate successfully, that our domain-specific language is expressive enough for specifying taint analyses for a relevant set of CWEs.

Being based on the synthetic OWASP benchmark suite, the conducted experiments only provide initial insights into the applicability and challenges of combining dynamic symbolic execution and taint analysis for the analysis of Web-Applications. The scalability of JAINT depends on the performance of the underlying dynamic symbolic execution engine. Here, the manually developed environment models may hamper application in industrial contexts. One direction of future work is thus the automation of environment modeling, e.g., using domain-specific languages.

References

1. Jon Allen. Perl version 5.8.8 documentation - perlsec. http://perldoc.perl.org/5.8.8/perlsec.pdf, May 2016.
2. Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):50, 2018.
3. Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A taint based approach for smart fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 818–825. IEEE, 2012.
4. Elisa Burato, Pietro Ferrara, and Fausto Spoto. Security analysis of the OWASP benchmark with Julia. *Proceedings of ITASEC*, 17, 2017.
5. Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
6. Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013. `doi:10.1145/2408776.2408795`.
7. Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *11th IEEE Symposium on Computers and Communications (ISCC'06)*, pages 749–754. IEEE, 2006.
8. James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007.
9. Juan José Conti and Alejandro Russo. A taint mode for Python via a library. In *Nordic Conference on Secure IT Systems*, pages 210–222. Springer, 2010.
10. Ricardo Corin and Felipe Andrés Manzano. Taint analysis of security code in the KLEE symbolic execution engine. In Tat Wing Chim and Tsz Hon Yuen, editors, *Information and Communications Security*, pages 264–275, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
11. Ehsan Edalat, Babak Sadeghiyan, and Fatemeh Ghassemi. Considroid: A concolic-based tool for detecting SQL injection vulnerability in android apps. *CoRR*, abs/1811.10448, 2018. `arXiv:1811.10448`.
12. Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 9–pp. IEEE, 2005.
13. Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

14. Kangkook Jee, Georgios Portokalidis, Vasileios P Kemerlis, Soumyadeep Ghosh, David I August, and Angelos D Keromytis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *NDSS*, 2012.
15. Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011. URL: http://www.isoc.org/isoc/conferences/ndss/11/pdf/5_4.pdf.
16. Lap Chung Lam and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 463–472. IEEE, 2006.
17. V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.
18. Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. Jdart: A dynamic symbolic analysis framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 442–459. Springer, 2016.
19. Malte Mues and Falk Howar. JDart: Dynamic symbolic execution for Java bytecode (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 398–402. Springer, 2020.
20. Malte Mues, Till Schallau, and Falk Howar. Artifact for 'Jaint: A Framework for User-Defined Dynamic Taint-Analyses based on Dynamic Symbolic Execution of Java Programs', September 2020. `doi:10.5281/zenodo.4060244`.
21. James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.
22. Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. *Automatically hardening web applications using precise tainting*. Springer, 2005.
23. Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.
24. Daniel Schoepe, Musard Balliu, Benjamin C Pierce, and Andrei Sabelfeld. Explicit secrecy: A policy for taint tracking. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 15–30. IEEE, 2016.
25. Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.
26. Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In R. Sekar and Arun K. Pujari, editors, *Information Systems Security*, pages 1–25, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
27. Fausto Spoto. The julia static analyzer for java. In Xavier Rival, editor, *Static Analysis*, pages 39–57, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
28. Wei Xu, Sandeep Bhatkar, and Ramachandran Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, pages 121–136, 2006.