Revealing Injection Vulnerabilities by Leveraging Existing Tests

Katherine Hough[1], Gebrehiwet Welearegai[2], Christian Hammer[2] and Jonathan Bell[1]
[1]George Mason University, Fairfax, VA, USA
[2]University of Potsdam, Potsdam, Germany
khough2@gmu.edu,welearegai@uni-potsdam.de,hammer@cs.uni-potsdam.de,bellj@gmu.edu

Abstract

Code injection attacks, like the one used in the high-profile 2017 Equifax breach, have become increasingly common, now ranking #1 on OWASP's list of critical web application vulnerabilities. Static analyses for detecting these vulnerabilities can overwhelm developers with false positive reports. Meanwhile, most dynamic analyses rely on detecting vulnerabilities as they occur in the field, which can introduce a high performance overhead in production code. This paper describes a new approach for detecting injection vulnerabilities in applications by harnessing the combined power of human developers' test suites and automated dynamic analysis. Our new approach, RIVULET, monitors the execution of developer-written functional tests in order to detect information flows that may be vulnerable to attack. Then, RIVULET uses a white-box test generation technique to repurpose those functional tests to check if any vulnerable flow could be exploited. When applied to the version of Apache Struts exploited in the 2017 Equifax attack, RIVULET quickly identifies the vulnerability, leveraging only the tests that existed in Struts at that time. We compared RIVULET to the state-of-the-art static vulnerability detector *Julia* on benchmarks, finding that RIVULET outperformed *Julia* in both false positives and false negatives. We also used RIVULET to detect new vulnerabilities.

CCS Concepts

• Security and privacy → Vulnerability management; **Web application security**; **• Software and its engineering → Software testing and debugging**.

Keywords

injection attacks, vulnerability testing, taint tracking

ACM Reference Format:
Katherine Hough, Gebrehiwet Welearegai, Christian Hammer and Jonathan Bell. 2020. Revealing Injection Vulnerabilities by Leveraging Existing Tests. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380326

1 Introduction

In the high-profile 2017 *Equifax* attack, millions of individuals' private data was stolen, costing the firm nearly one and a half *billion* dollars in remediation efforts [55]. This attack leveraged a *code injection* exploit in Apache Struts (CVE-2017-5638) and is just one of over 8,200 similar code injection exploits discovered in recent years in popular software [44]. Code injection vulnerabilities have been exploited in repeated attacks on US election systems [10, 18, 39, 61], in the theft of sensitive financial data [56], and in the theft of millions of credit card numbers [33]. In the past several years, code injection attacks have persistently ranked at the top of the Open Web Application Security Project (OWASP) top ten most dangerous web flaws [46]. Injection attacks can be damaging even for applications that are *not* traditionally considered critical targets, such as personal websites, because attackers can use them as footholds to launch more complicated attacks.

In a code injection attack, an adversary crafts a malicious input that gets interpreted by the application as code rather than data. These weaknesses, "injection flaws," are so difficult to detect that rather than suggesting testing as a defense, OWASP suggests that developers try to avoid using APIs that might be targeted by attackers altogether or enforce site-wide input filtering. Consider again the Equifax hack: the underlying weakness that was exploited was originally introduced in 2011 and sat undetected in production around the world (not just at Equifax) for *six years* [4, 43]. While some experts blame Equifax for the successful attack — a patch had been released two months prior to the attack, but was not applied — one really has to ask: how is it possible that critical vulnerabilities go unnoticed in production software for so long?

With the exception of safety-critical and similar "high-assurance" software, general best practices call for developers to extensively test their applications, to perform code reviews, and perhaps to run static analyzers to detect potentially weak parts of their software. Unfortunately, testing is a never-ending process: how do developers know that they've truly tested all input scenarios? To catch code injection exploits just-in-time, researchers have proposed deploying dynamic *taint tracking* frameworks, which track information flows, ensuring that untrusted inputs do not flow into sensitive parts of applications, *e.g.,* interpreters [8, 21, 38, 54, 58, 63]. However, these approaches have prohibitive runtime overheads: even the most performant can impose a slowdown of at least 10–20% and often far more [8, 12, 15, 31]. Although black-box fuzzers can be connected with taint tracking to detect vulnerabilities in the lab, it is difficult to use these approaches on stateful applications or those that require structured inputs [23, 32]. While some static analysis tools have seen recent developer adoption [9, 19, 45], statically detecting code injection vulnerabilities is challenging since static analysis tools must perform interprocedural data flow analysis [7, 59, 60, 71].

Our key idea is to use dynamic taint tracking *before deployment* to amplify developer-written tests to check for injection vulnerabilities. These integration tests typically perform functional checks. Our approach re-executes these existing test cases, *mutating* values that are controlled by users (*e.g.,* parts of each of the test's HTTP requests) and detecting when these mutated values result in real

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380326

attacks. To our knowledge, this is the *first* approach that combines dynamic analysis with existing tests to detect injection attacks.

Key to our test amplification approach is a white-box *context-sensitive* input generation strategy. For each user-controlled value, state-of-the-art testing tools generate hundreds of attack strings to test the application [32, 49, 50]. By leveraging the context of *how* that user-controlled value is used in security-sensitive parts of the application, we can trivially rule out most of the candidate attack strings for any given value, reducing the number of values to check by orders of magnitude. Our testing-based approach borrows ideas from both fuzzing and regression testing, and is language agnostic.

We implemented this approach in the JVM, creating a tool that we call Rivulet. Rivulet Reveals Injection VUlnerabilities by Leveraging Existing Tests, and does not require access to application source code, and runs in commodity, off-the-shelf JVMs, integrating directly with the popular build automation platform *Maven*.

Like any testing-based approach, Rivulet is not guaranteed to detect all vulnerabilities. However, Rivulet guarantees that every vulnerability that it reports meets strict criteria for demonstrating an attack. We found that Rivulet performed as well as or better than a state-of-the-art static vulnerability detection tool [59] on several benchmarks. Rivulet discovers the Apache Struts vulnerability exploited in 2017 Equifax hack within minutes. When we ran Rivulet with the open-source project *Jenkins*, Rivulet found a previously unknown cross-site scripting vulnerability, which was confirmed by the developers. On the educational project *iTrust* [22], Rivulet found 5 previously unknown vulnerabilities. Unlike the state-of-the-art static analysis tool that we used, Julia [59], Rivulet did not show *any* false positives.

Using dynamic analysis to detect injection vulnerabilities before deployment is hard, and we have identified two key challenges that have limited past attempts: (1) Unlike static analysis, dynamic analysis requires a representative workload to execute the application under analysis; and (2) For each potential attack vector, there may be hundreds of input strings that should be checked. Rivulet addresses these challenges, making the following key contributions:

- A technique for re-using functional test cases to detect security vulnerabilities by modifying their inputs and oracles
- Context-sensitive mutational input generators for SQL, OGNL, and XSS that handle complex, stateful applications
- Embedded attack detectors to verify whether rerunning a test with new inputs leads to valid attacks

Rivulet is publicly available under the MIT license [24, 25].

2 Background and Motivating Example

Injection vulnerabilities come in a variety of flavors, as attackers may be able to insert different kinds of code into the target application. Perhaps the most classic type of injection attack is *SQL Injection (SQLI)*, where attackers can control the contents of an SQL statement. For instance, consider this Java-like code snippet that is intended to select and then display the details of a user from a database: `execQuery("SELECT * from Users where name = '" + name + "'");`. If an attacker can arbitrarily control the value of the `name` variable, then they may perform a SQL injection attack. For instance, the attacker could supply the value `name = "Bob' OR '1'='1"` which, when joined to the query string will produce `where name = 'Bob' OR '1'='1'`, which would result in *all* rows in this user table being selected. SQLI attacks may result in data breaches, denial of service attacks, and privilege escalations.

Remote Code Execution (RCE) vulnerabilities are a form of injection vulnerabilities where an attacker can execute arbitrary code on an application server using the same system-level privileges as the application itself. Command injection attacks are a particularly dangerous form of RCE where an attacker may directly execute shell commands on the server. Other RCE attacks may target features of the application runtime that parse and execute code in other languages such as J2EE EL [47] or OGNL [66, 67].

Cross-site Scripting (XSS) vulnerabilities are similar to RCE, but result in code being executed by a user's browser, rather than on the server. XSS attacks occur when a user can craft a request that inserts arbitrary HTML, Javascript code, or both into the response returned by the server-side application. Such an attack might hijack a user's session (allowing the attacker to impersonate the user on that website), steal sensitive data, or inject key loggers. Server-side XSS attacks may be *reflected* or *persistent*. Reflected XSS attacks are typically used in the context of a phishing scheme, where a user is sent a link to a trusted website with the attack embedded in the link. Persistent XSS attacks occur when a payload is stored in the host system (*e.g.,* in a database) and is presented to users who visit the compromised site.

Developers defend their software from injection attacks through input validation and sanitization. Broadly, validation is a set of whitelisting techniques, such as: "only accept inputs that match a limited set of characters," while sanitization is a set of transformations that render attacks harmless, such as: "escape all quotation marks in user input." Ideally, each user-controlled input (also referred to as a "tainted source") that can reach critical methods that may result in code execution (also referred to as a "sensitive sink") will be properly sanitized, validated, or both. Reaching such an ideal state is non-trivial [37]. Hence, the key challenge in detecting these vulnerabilities is to detect flows from tainted sources to sensitive sinks that have not been properly sanitized.

Listing 1 shows a simplified example of two genuine cross-site scripting vulnerabilities. Lines 9 and 10 show a parameter provided by the user flowing into the response sent back to the browser *without* proper sanitization. In the first case (line 9), the vulnerability occurs despite an attempt to sanitize the user's input (using the Apache Commons-Language library function `escapeHtml4`), and in the second case (line 10), there is no sanitization at all.

```
1 @Override
2 public void doGet(HttpServletRequest request,
      HttpServletResponse response) throws IOException {
3   String name = request.getParameter("name");
4   response.setContentType("text/html");
5   String escaped = StringEscapeUtils.escapeHtml4(name);
6   String content = "<a href=\"%s\">hello</a>";
7   try(PrintWriter pw = response.getWriter()) {
8     pw.println("<html><body>");
9     pw.println(String.format(content, escaped));
10    pw.println(String.format(content, name));
11    pw.println("</body></html>");
12  }
13 }
```

Listing 1: Two example XSS vulnerabilities. An untrusted user input from an HTTP request flows into the response to the browser on lines 9 and 10.

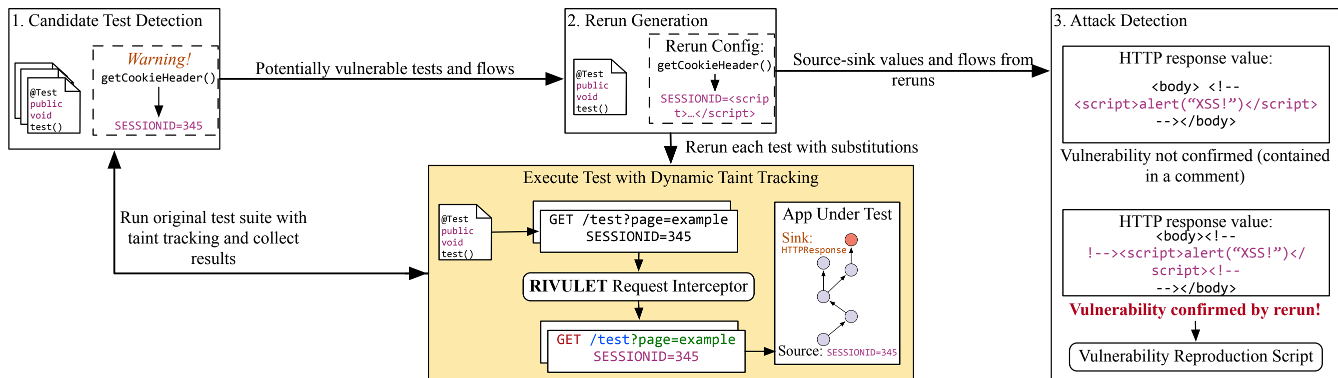

Figure 1: High-Level Overview of Rivulet. Rivulet detects vulnerabilities in three phases. Key to our approach is the repeated execution of developer-provided test cases with dynamic taint tracking. First, each developer-provided test is executed using taint tracking to detect which tests expose potentially vulnerable data flows. HTTP requests made during a test are intercepted and parsed into their syntactic elements which are then tainted with identifying information. Then, source-sink flows observed during test execution are recorded and passed with contextual information to a rerun generator. The rerun generator creates rerun configurations using the supplied flow and contextual information, and executes these reruns, swapping out developer-provided inputs for malicious payloads. Source-sink flows observed during test re-execution are passed to an attack detector which verifies source-sink flows that demonstrate genuine vulnerabilities.

In either case, providing the input string `javascript:alert('XSS');` for the parameter `"name"` will result in JavaScript code executing in the client's browser if they click on the link. The chosen sanitizer escapes any HTML characters in the input string (*i.e.,* preventing an injection of a `<script>` tag), but is insufficient for this case, as an attacker need only pass the prefix `javascript:` in their payload to cause code to execute when the user clicks on this link (many XSS attack payloads do not include brackets or quotes for this reason [50]).

To fix this vulnerability, the developer needs to apply a sanitizing function that prevents the insertion of JavaScript code. Static analysis tools, such as the state-of-the-art Julia platform [59], typically assume that library methods pre-defined as sanitizers for a class of attack (*e.g.,* XSS sanitizers) eliminate vulnerabilities for the data flows that they are applied to. In our testing-based approach, sanitizer methods do not need to be annotated by users. Instead we test whether a flow is adequately sanitized by attempting to generate a counterexample (*i.e.,* a malicious payload that produces a successful injection attack).

3 Approach Overview

Generating tests that expose the rich behavior of complicated, stateful web applications can be quite difficult. For instance, consider a vulnerability in a health records application that can only be discovered by logging in to a system, submitting some health data, and sending a message to a healthcare provider. Fuzzers have long struggled to generate inputs that follow a multi-step workflow like this example [23, 32]. Instead, Rivulet begins by executing the existing, ordinary test suite that developers have written, which does *not* need to have any security checks included in it: in this healthcare messaging example, an existing test might simply check that the workflow completes without an error. As we show in our evaluation (§5), even small test suites can be used by Rivulet to detect vulnerabilities.

Figure 1 shows a high-level overview of Rivulet's three-step process to detect injection vulnerabilities in web applications. First, Rivulet uses dynamic taint tracking while running each test to observe data flows from "sources," untrusted system inputs controlled by a potentially malicious actor, to "sinks," sensitive parts of an application that may be vulnerable to injection attacks. These source-sink flows do not necessarily represent vulnerabilities: it is possible that a sanitizer function correctly protects the application. Hence, when a source-sink flow is observed, Rivulet generates malicious payloads based on contextual information of the sink method. Then, tests are re-executed and those untrusted source values are replaced with generated payloads, probing for weak or missing sanitizers. Lastly, specialized logic based on the type of vulnerability, *e.g.,* XSS, is used as an oracle to determine whether a re-execution demonstrates a successful attack, thereby transforming a functional test into a security test.

In this way, source-sink flows are verified as vulnerable only if a successful attack can be demonstrated using a concrete exploit. This standard produces few false positives. Test reruns enable our technique to consider input sanitization and validation without requiring sanitization and validation methods to be explicitly specified or modeled. Verifying whether a sanitizer or validator is correct in all cases is a hard problem and beyond the scope of this work. However, if a system sanitizes or validates input improperly before it flows into a sink method, then one of the malicious payloads may be able to demonstrate a successful attack, causing the flow to be verified. Our implementation of Rivulet (described in §4) automatically detects SQL injection, remote code execution, and cross-site scripting vulnerabilities: developers do not need to specify any additional sources or sinks in order to find these kinds of vulnerabilities. Section 4 describes, in detail, the specific strategy that Rivulet uses to find these kinds of vulnerabilities.

3.1 Detecting Candidate Tests

Rivulet co-opts existing, functional test cases to test for security properties by mutating user-controlled inputs and adding security-based oracles to detect code that is vulnerable to injection attacks. We assume that developers write tests that demonstrate typical application behavior, and our approach relies on automated testing to detect weak or missing sanitization. This assumption is grounded in

best practices for software development: we assume that developers will implement some form of automated functional testing before scrutinizing their application for security vulnerabilities. Rivulet detects candidate tests by executing each test using dynamic taint tracking, identifying tests that expose potentially vulnerable source-sink flows, each of which we refer to as a *violation*. By leveraging developer tests, our approach can detect vulnerabilities that can only be revealed through a complex sequence of actions. These vulnerabilities can be difficult for test generation approaches to detect, but are critical when dealing with stateful applications [32].

In this model, developers do *not* need to write test cases that demonstrate an attack — instead, they need only write test cases that expose an information flow that is vulnerable to an attack. For instance, consider a recent Apache Struts vulnerability (CVE-2017-9791) that allowed user-provided web form input to flow directly into the Object-Graph Navigation Language (OGNL) engine. Struts includes a sample application for keeping track of the names of different people, this application can be used to demonstrate this vulnerability by placing an attack string in the "save person" form. To detect this vulnerability, we do *not* need to observe a test case that uses an attack string in the input, instead, we need only observe *any* test that saves *any* string through this form in order to observe the insecure information flow. Once this is detected, Rivulet, can then re-execute and perturb the test case, mutating the value of the form field, eventually demonstrating the exploit.

3.2 Rerun Generation and Execution

The next phase in Rivulet's vulnerability detection process is to re-execute each test, perturbing the inputs that the server received from the test case in order to add malicious payloads. A significant challenge to our approach is in the potentially enormous number of reruns that Rivulet needs to perform in order to test each potentially vulnerable source-sink flow. If an application has *thousands* of tests, each of which may have *dozens* of potentially vulnerable flows, it is crucial to limit the number of times that each test needs to be perturbed and re-executed. Unfortunately, it is typical to consider over *100* different malicious XSS payloads for *each* potentially vulnerable input [32, 50], and other attacks may still call for dozens of malicious payloads.

Instead, Rivulet uses a white-box, in situ approach to payload generation in order to drastically reduce the number of reruns needed to evaluate a source-sink flow. Successful injection attacks often need to modify the syntactic structure of a query, document or command from what was intended by the developer [62]. By looking at the placement of taint tags (representing each source) within structured values that reach sink methods, *i.e.,* the syntactic context into which untrusted values flow, the number of payloads needed to test a flow can be limited to only those capable of disrupting that structure from the tainted portions of the value.

For instance, when an untrusted value reaches a sink method vulnerable to SQL injection attacks, developers usually intend for the value to be treated as a string or numeric literal. Consider the following SQL query: `SELECT * FROM animals WHERE name = '%Tiger%';` where the word `Tiger` is found to be tainted. In order to modify the structure of the query, a payload must be able to end the single-quoted string literal containing the tainted portion of the query. Payloads which do not contain a single-quote would be ineffective in this context, *e.g.,* payloads that aim to end double-quoted string literals, and do not need to be tested when evaluating this flow. Rivulet uses a similar approach for generating payloads for other kinds of attacks, as we will describe in § 4.2.

3.3 Attack Detection

The attack detector component provides the oracle for each modified test (removing any existing assertions), determining if the new input resulted in a successful attack on the system under test. There is a natural interdependence between payload generation and attack detection. Attack detection logic must be able to determine the success of an attack using any of the payloads that could be generated by Rivulet. Likewise, generated payloads should aim to trigger a successful determination from the detection logic. This relationship can be used not only to guide payload generation, but also to enable stricter (and simpler to implement) criteria for determining what constitutes a successful attack. Specifically, it is not necessary to recognize all possible successful attacks, but instead, only those generated by the system. Furthermore, this reduces the difficulty of formulating an appropriate detection procedure, particularly for certain types of attacks. Rivulet's attack detectors inspect both the taint tags and concrete values of data that reaches sensitive sinks.

4 Implementation

Our implementation of Rivulet for Java is built using the Phosphor taint tracking framework [8], and automatically configures the popular build and test management platform *Maven* to perform dynamic taint tracking during the execution of developer-written tests, generate malicious payloads based on source-sink flows observed during test execution, and execute test reruns. Developers can use Rivulet by simply adding a single maven extension to their build configuration file: Rivulet and Maven automatically configure the rest. Out of the box, Rivulet detects cross-site scripting, SQL injection, and OGNL injection vulnerabilities without any additional configuration. Phosphor propagates taint tags by rewriting Java bytecode using the ASM bytecode instrumentation and analysis framework [51], and does not require access to application or library source code. We chose Phosphor since it is capable of performing taint tracking on all Java data types, ensuring that Rivulet is not limited in its selection of source and sink methods to only methods that operate on strings.

4.1 Executing Tests with Dynamic Tainting

Rivulet's approach for dynamic taint tracking within test cases is key to its success. Taint tracking allows data to be annotated with labels (or "taint tags"), which are propagated through data flows as the application runs. It is particularly critical to determine *where* these tags are applied (the "source methods") and how they correspond to the actual input that could come from a user, since it is at these same source methods that Rivulet injects malicious values when rerunning tests.

Many approaches to applying taint tracking to HTTP requests in the JVM use high-level Java API methods as taint sources, such as `ServletRequest.getParameter()` for parameters or, for cookies, `HttpServletRequest.getCookies()` [13, 20, 40, 59]. However, these approaches can be brittle: if a single source is missed or a new version of the application engine is used (which adds new sources),

there may be false negatives. Moreover, since application middleware (between the user's socket request and these methods) performs parsing and validation, mutating these values directly could result in false positives when replaying and mutating requests. If Rivulet performed its injection *after* the middleware parses the HTTP request from the socket (*i.e.,* as a user application reads a value from the server middleware), Rivulet might generate something that could never have passed the middleware's validation. For instance, if performing a replacement on the method `getCookies()`, Rivulet might try to generate a replacement value `NAME=alert(String.fromCharCode(88,88,83))`, which could *never* be a valid return value from this method source, since HTTP cookies may not contain commas [41].

Instead of using existing Java methods as taint sources, Rivulet uses bytecode instrumentation to intercept the bytes of HTTP requests directly as they are read from sockets. Intercepted bytes are then buffered until a full request is read from the socket. Requests read from the socket are parsed into their syntactic elements, *e.g.,* query string, entity-body, and headers. Each element then passes through a taint source method which taints the characters of the element with the name of the source method, the index of the character in the element, and a number assigned to the request that was parsed. The original request is then reconstructed from the tainted elements and broken down back into bytes which are passed to the object that originally read from the socket. This technique allows a tainted value to be traced back to a range of indices in a syntactic element of a specific request. Thus, this tainting approach enables precise replacements to be made during test re-executions.

We have integrated Rivulet with the two most popular Java HTTP servers, Tomcat [5] and Jetty [68], using bytecode manipulation. Rivulet modifies components in Tomcat and Jetty which make method calls to read bytes from a network socket to instead pass the receiver object (*i.e.,* the socket) and arguments of the call to the request interceptor. The interceptor reads bytes from any socket passed to it, parses the bytes into a request and taints the bytes based on their semantic location within the parsed request. It would be easy to add similar support to other Java web servers, however, Tomcat and Jetty are the most popular platforms by far.

4.2 Rerun Generation

Rivulet uses an easy-to-reconfigure, predefined set of sink methods, which we enumerate by vulnerability type below. When a sink method is called, the arguments passed to the call are recursively checked for taint tags, *i.e.,* arguments are checked, the fields of the arguments are checked, the fields of the fields of arguments checked, and so on until to a fixed maximum checking depth is reached. If a tainted value is found during the checking process, a source-sink flow is recorded. When Rivulet finishes checking the arguments of the call, it passes contextual information and flow information to a generator that handles the type of vulnerability associated with the sink method that was called. The contextual information consists of the receiver object of the sink method call and the arguments of the call. The flow information consists of the source information contained in the labels of the tainted values that were found and a description of the sink method that was called.

Rerun generators create rerun configurations identifying the test case that should be rerun, the detector that should be used to determine whether a successful attack was demonstrated by the rerun, the original source-sink flow that the rerun is trying to verify, and at least one replacement. Replacements define a replacement value, information used to identify the source value that should be replaced (target information), and possibly a "strategy" for how the source value should be replaced. A replacement can either be built as a "payload" replacement or a "non-payload" replacement.

Payload replacements are automatically assigned target information and sometimes a strategy based on flow information. For example, the labels on a tainted value that reached some sink might show that the value came from indices $6 - 10$ of the second call to the source `getQueryString()`. One payload replacement built off of that flow information would direct that the second time `getQueryString()` is called that its return value should be replaced using a strategy that replaces only indices six through ten with a replacement value. Payload replacements are how malicious payloads are normally specified, thus every rerun is required to have at least one of them. Non-payload replacements are useful for specifying secondary conditions that an attack may need in order to succeed, such as changing the "Content-Type" header of a request.

SQL Injection. The rerun generator for SQL injection uses all `java.sql.Statement` and `java.sql.Connection` methods that accept SQL code as sinks, and considers three primary SQL query contexts in which a tainted value may appear: literals, comments, LIKE clauses. Tainted values appearing in other parts of the query are treated similarly to unquoted literals. Tainted values appearing in LIKE clauses are also considered to be in literals, thus cause both the payloads for tainted literals and tainted LIKE clauses to be generated. If a tainted value appears in a literal, the generator first determines the "quoting" for the literal. A literal can be either unquoted (like a numeric literal might be), single-quoted, double-quoted, or backtick-quoted (used for table and column identifiers in MySQL). Payloads for tainted literals are prefixed by a string that is based on the quoting of the literal in order to attempt to end the literal. The quoting can also be used to determine an appropriate ending for payloads. If a tainted value appears in a comment, the generator first determines the characters used to end and start the type of comment the value appears in. Payloads for tainted comments are prefixed by the end characters for the comment and ended with the start characters for the comment. If a tainted value appears in a LIKE clause, the generator creates payloads containing SQL wildcard characters.

Rivulet generates 2–5 SQL injection payloads for a tainted value in a particular context out of 20 unique payloads that could be generated for the same tainted value across all of the contexts considered by the SQL injection rerun generator. If wildcard payloads for LIKE clause are not generated then only 2–3 payloads are generated per context. This is a reduction from Kieżun *et al.*'s *Ardilla*, which uses 6 SQL injection patterns and does not consider tainted backtick-quoted values, comments, or LIKE clauses [32].

Cross-Site Scripting. Rivulet uses special sink checking logic for XSS, checking data as it is sent over-the-wire to the browser. The overloaded variants of `SocketChannel.write()` are used as sink methods for XSS attacks. In order to give the XSS generator all of the HTML content for a single response at once, Rivulet stores the bytes written to a socket until a full response can be parsed from the bytes. If the parsed response contains HTML content and the

HTML in the response's entity-body contains a tainted value, then that HTML is passed to the XSS rerun generator.

The XSS rerun generator parses HTML content into an HTML document model using the Jsoup library [29]. This model is traversed, generating payloads for each tainted value encountered. The XSS rerun generator considers 5 primary HTML document contexts in which a tainted value may appear: tag names, attribute names, attribute values, text or data content, and comments. Different payloads are capable of introducing a script-triggering mechanism into the document's structure depending on the context. Rivulet also addresses context-specific issues like the quoting of attribute values or whether content is contained in an element which causes the tokenizer to leave the data state during parsing [74].

The XSS generator also considers whether a tainted value was placed in a context that would already be classified as an embedded script or the address of an external script. Furthermore, if a tainted value appears in a context that would be classified as an embedded script then the generator also determines whether the tainted value is contained within a string literal, template literal, or comment.

Rivulet generates 3–7 XSS payloads for a tainted value in a particular context out of over 100 unique payloads that could be generated for the same tainted value across all of the contexts considered by the XSS rerun generator. By comparison, OWASP's "XSS Filter Evasion Cheat Sheet" features 152 unique payloads for cross-site scripting attacks [50] and Ardilla uses 106 patterns for creating cross-site scripting attacks [32].

Command and OGNL Injection. The command injection rerun generator creates payloads with common UNIX commands like `ls`, considering `java.lang.ProcessBuilder` and `java.lang.Runtime` methods as sinks.

The OGNL injection rerun generator creates payloads that facilitate attack detection. It can be difficult to specify generic criteria for detecting any OGNL injection attack because the language is designed to allow users to execute "non-malicious" code. OGNL expressions can modify Java objects' properties, access Java objects' properties and make method calls [66]. Applications using OGNL can limit the code that user specified expressions can execute by whitelisting or blacklisting certain patterns [67]. The evaluation of improperly validated OGNL expressions can enable a user to execute arbitrary code. The OGNL rerun generator uses payloads that we collected from the Exploit Database [16] and simplified to integrate more tightly with Rivulet's attack detection mechanism.

Rerun Execution. Rerun configurations created by the rerun generators specify test cases that should be re-executed. Values are replaced when they are assigned a label at a source method and the information on the label being assigned to the value meets the criteria specified by one of the current rerun configuration's replacements. Replacements may dictate a strategy for replacing the original value; strategies can specify ways of combining an original value with a replacement value, a way of modifying the replacement value, or both. For example, a strategy could specify that only a certain range of indices in the original value should be replaced, that the replacement value should be percent encoded, or both. Rivulet automatically converts values to ensure that the type of a replacement value is appropriate (*e.g.,* converting between a string and a character array).

4.3 Attack Detection

Rerun configurations specify which vulnerability-specific attack detector should be used to check flows during a test re-execution.

SQL Injection. Our approach for detecting SQL injection attack builds on Halfond *et al.*'s "syntax-aware evaluation" model, which calls for checking that all parts of SQL queries except for string and numeric literals come from trusted sources [21]. We determine a SQL injection attack to be successful if a tainted SQL keyword not contained in a literal or comment is found within a query that reached a sink vulnerable to SQL injection. Alternatively, an attack is deemed successful if a sink-reaching query contains a `LIKE` clause with an unescaped tainted wildcard character (*i.e.,* % or _) as the system could be vulnerable to a SQL wildcard denial-of-service attack [49]. The attack detector for SQL injection uses ANTLR, a parser generation tool [65] and JSqlParser, a SQL statement parser that supports multiple SQL dialects [30], to parse SQL statements that reach sink methods vulnerable to SQL injection attacks.

Cross-Site Scripting. The World Wide Web Consortium's (W3C's) Recommendation for HTML 5.2 specifies mechanisms which can trigger the execution of embedded or external scripts: "processing of script elements," "navigating to javascript: URLs," "event handlers," "processing of technologies like SVG that have their own scripting features" [73]. Only the syntactic components of an HTML document that are capable of activating a script-triggering mechanism are vulnerable to script injections. As such, we determine the success of an XSS attack by checking these vulnerable components.

Rivulet intercepts and buffers the bytes of HTTP responses until a full response can be parsed from the bytes. Then, the parsed document is checked for components that could activate a script-triggering mechanism. Depending on the mechanism potentially activated by the component, a portion of the component is then classified as either an embedded script or the address of an external script. The following rules are used to identify embedded and external scripts in the response: (1) The inner content of every "script" tag is classified as an embedded script. (2) The HTML entity decoded value of every "src" attribute specified for a "script" tag is classified as an external script's address. (3) The HTML entity decoded value of every "href" attribute specified for a "base" tag is classified as an external script's address because of its potential impact on elements in the document using relative URLs. (4) The HTML entity decoded value of every event handler attribute, *e.g.,* "onload," specified for any tag is classified as an embedded script. (5) The HTML entity decoded value of every attribute listed as having a URL value in W3C's Recommendation for HTML 5.2 [73], *e.g.,* the "href" attribute, is examined. If the decoded value starts with "javascript:", then the portion of the decoded value after "javascript:" is classified as an embedded script.

Embedded scripts are checked for values successfully injected into non-literal, non-commented portions of the script. To do so, the portions of the script that are not contained in JavaScript string literals, template literals, or comments are checked for a predefined target string. This target string is based on the malicious payload being used in the current test re-execution, *e.g.,* `alert` is an appropriate target string for the payload `<script>alert(1)</script>`, but other payloads may have more complicated target strings. If the target string is found in the non-literal, non-commented portions

of the script and it is tainted, then the attack is deemed successful. Since the target string must be tainted to be deemed a successful attack, a vulnerability will be reported only if an attacker could inject that target string into the application.

External script addresses are checked for successfully injected URLs that could potentially be controlled by a malicious actor. The start of each address is checked for a predefined target URL. The target URL is based on the malicious payload being used in the current test re-execution. If the target URL is found at the start of an address and is tainted, then the attack is deemed successful.

The XSS attack detector stores bytes written to a socket by calls to `SocketChannel.write()` until a full response can be parsed (using Jsoup [29]) from the bytes stored for a particular socket. The rules described above are then applied to the document model parsed from the entity-body. The embedded script checks are also performed using ANTLR [65] and a simplified grammar for JavaScript to identify string literals, template literals, and comments.

Command and OGNL Injection. A command injection attack is determined to be successful if any tainted value flows into a sink vulnerable to command injection (such as `ProcessBuilder.command()` and `Runtime.exec()`). Additionally, if a call is made to `ProcessBuilder.start()`, the detector will deem an attack successful if the "command" field of the receiver object for the call is tainted. This relatively relaxed standard is a product of a lack of legitimate reasons for allowing untrusted data to flow into these sinks and the severity of the security risk that doing so presents. This approach could be fine-tuned to perform more complicated argument parsing (similar to the XSS detector), however, in practice, we found it sufficient, producing no false positives on our evaluation benchmarks. We use a similar tactic to test for successful OGNL injection attacks since the OGNL payloads generated by Rivulet are crafted to perform command injection attacks.

4.4 Limitations

Our approach is not intended to be complete; it is only capable of detecting vulnerabilities from source-sink flows that are exposed by a test case. Hence, Rivulet requires applications to have existing test cases, although we believe that this is a fair assumption to make, and in our evaluation, show that Rivulet can detect a real vulnerability even when presented with a very small test suite (for Apache Struts). This limitation could be mitigated by integrating our approach with an automatic test generation technique. Vulnerabilities caused by a nondeterministic flow are hard for Rivulet to detect, even if the flow occurs during the original test run, because the flow may fail to occur during the re-execution of the test. Rivulet does not detect XSS attacks which rely on an open redirection vulnerability [69]. More generally, Rivulet can only detect attacks that we have constructed generators and detectors for, but this is primarily a limitation of Rivulet's implementation, and not its approach. We note that even static analysis tools can only claim soundness to the extent that their model holds in the code under analysis: in our empirical evaluation of a sound static-analysis tool, we found that the static analyzer missed several vulnerabilities (§5.1).

Since Phosphor is unable to track taint tags through code outside of the JVM, Rivulet is also unable to do so. As a result, Rivulet cannot detect persistent XSS vulnerabilities caused by a value stored in an external database, but it can detect one caused by a value stored in Java heap memory. We plan to propose extensions to Phosphor to overcome this limitation, building off of work demonstrating the feasibility of persisting taint tags in databases in the Android-based TaintDroid system [64]. At present, Rivulet can only detect vulnerabilities that result from explicit (data) flow, and not through implicit (control) flows, or side-channels such as timing [53], a limitation shared by most other tools, including Julia [59]. Experimental support for implicit flow tracking in Phosphor may lift this limitation in the future. Despite these limitations, we have found Rivulet to be effective at detecting injection vulnerabilities.

5 Evaluation

We performed an empirical evaluation of Rivulet, with the goal of answering several research questions:

- **RQ1:** How does Rivulet perform in comparison to a state-of-the-art static analysis tool?
- **RQ2:** Does Rivulet scale to large projects and their test suites?
- **RQ3:** How significantly does Rivulet's contextual payload generation reduce the number of reruns needed?

To answer these questions, we applied both Rivulet and the state-of-the-art static analysis tool Julia [59] to several suites of vulnerability detection benchmarks. These curated benchmarks are intentionally seeded with vulnerabilities, allowing us to compare Rivulet and Julia in terms of both precision and recall. We were also able to use one of these benchmarks to compare Rivulet against six commercial vulnerability detection tools. These benchmarks allow us to evaluate the efficacy of Rivulet's attack generators and detectors, but since they are micro-benchmarks, they do not provide much insight into how Rivulet performs when applied to real, developer-provided test suites. To this end, we also applied Rivulet to three larger applications and their test suites.

We conducted all of our experiments on Amazon's EC2 infrastructure, using a single "c5d.4xlarge" instance with 16 3.0Ghz Intel Xeon 8000-series CPUs and 32 of RAM, running Ubuntu 16.04 "xenial" and OpenJDK 1.8.0_222. We evaluated Julia by using the JuliaCloud web portal, using the most recent version publicly available as of August 16, 2019. When available (for Juliet-SQLI, Juliet-XSS and all of OWASP), we re-use results reported by the Julia authors [59]. When we executed it ourselves, we confirmed our usage of Julia through personal communication with a representative of JuliaSoft, and greatly thank them for their assistance.

5.1 RQ1: Evaluating Rivulet on Benchmarks

In order to evaluate the precision and recall of Rivulet and Julia, we turn to third-party vulnerability detection benchmarks, specifically NIST's Juliet Benchmark version 1.3 [42], OWASP's Benchmark version 1.2 [48], Livshits' securibench-micro [36], and the Application Vulnerability Scanner Evaluation Project's WAVSEP version 1.5 [11]. Each of these benchmarks contains test cases with vulnerabilities that are representative of real vulnerabilities found in various applications. From these tests, we can collect the number of true positives and false negatives reported by each tool. The benchmarks also contain test cases with variants of those vulnerabilities that are *not* vulnerable, allowing us to also collect the number of false positives and true negatives reported by each tool.

Each benchmark consists of a series of web servlets (and in some cases, also non-servlet applications) that are tests well-suited

Table 1: Comparison of Rivulet and Julia [59] on third-party benchmarks. For each vulnerability type in each benchmark suite we show the total number of test cases (for both true and false alarm tests). For Rivulet and Julia, we report the number of true positives, false positives, true negatives, false negatives, and analysis time in minutes. Times are aggregate for the whole benchmark suite.

		# Test Cases		Rivulet					Julia				
Suite	Type	True Alarm	False Alarm	TP	FP	TN	FN	Time	TP	FP	TN	FN	Time
Juliet	RCE	444	444	444	0	444	0	25	444	0	444	0	33
	SQL	2,220	2,220	2,220	0	2,220	0		2,220	0	2,220	0	
	XSS	1,332	1,332	1,332	0	1,332	0		1,332	0	1,332	0	
OWASP	RCE	126	125	126	0	125	0	3	126	20	105	0	15
	SQL	272	232	272	0	232	0		272	36	196	0	
	XSS	246	209	246	0	209	0		246	19	190	0	
Securibench-Micro	SQL	3	0	3	0	0	0	1	3	0	0	0	1
	XSS	86	21	85	0	21	1		77	14	7	9	
WavSep	SQL	132	10	132	0	10	0	2	132	0	10	0	2
	XSS	79	7	79	0	7	0		79	6	1	0	

for analysis by a static analyzer like Julia. However, Rivulet requires executable, JUnit-style test cases to perform its analysis. Each servlet is designed to be its own standalone application to analyze, and they are not stateful. Hence, for each benchmark, we generated JUnit test cases that requested each servlet over HTTP, passing along some default, non-malicious parameters as needed. Where necessary, we modified benchmarks to resolve runtime errors, mostly related to invalid SQL syntax in the benchmark. We ignored several tests from securibench-micro that were not at all suitable to dynamic analysis (some had infinite loops, which would not result in a page being returned to the user), and otherwise included only tests for the vulnerabilities targeted by Rivulet (RCE, SQLI and XSS). Most of these benchmarks have only been analyzed by static tools, and not executed, and hence, such issues may not have been noticed by prior researchers. For transparency and reproducibility, all benchmark code is included in this paper's accompanying artifact [24].

Table 1 presents our findings from applying both Rivulet and Julia to these benchmarks. Rivulet had near perfect recall and precision, identifying every true alarm test case as a true positive but one, and every false alarm test case as a true negative. In three interesting Securibench-Micro test cases, the test case was non-deterministically vulnerable: with some random probability the test could be vulnerable or not. In two of these cases, Rivulet eventually detected the vulnerability after repeated trials (the vulnerability was exposed with a 50% probability and was revealed after just several repeated trials). However, in the case that we report a false negative (simplified and presented in Listing 2), the probability of any attack succeeding on the test was just $1/2^{32}$, and Rivulet could not detect the vulnerability within a reasonable time bound. We note that this particularly difficult case does not likely represent a significant security flaw, since just like Rivulet, an attacker can not control the probability that their attack would succeed. This test case likely represents the worst-case pathological application that Rivulet could encounter.

In comparison, Julia demonstrated both false positives and false negatives. Many of the false positives were due to Julia's lack of sensitivity for multiple elements in a collection, resulting in overtainting all elements in a collection. We confirmed with JuliaSoft that the tool's false negatives were not bugs, and instead generally due to limitations in recovering exact dynamic targets of method calls when the receiver of a method call was retrieved from the heap, causing it to (incorrectly) assume a method call to not be a sink. Listing 3 shows an example of one such case, where Julia reports a vulnerability on Line 3 but not on Line 6 since it is unable to precisely determine the dynamic target of the second `println`. Unlike the very tricky non-deterministic case that Rivulet struggled to detect, we note that this form of data flow is not uncommon, and this limitation may significantly impact Julia's ability to detect XSS vulnerabilities in applications that pass the servlet's `PrintWriter` between various application methods.

We also collected execution times to analyze each entire benchmark for both tools. For Rivulet, we report the total time needed to execute each benchmark (including any necessary setup, such as starting a MySQL server), and for Julia, we report the execution time from the cloud service. Despite its need to execute thousands of JUnit tests, Rivulet ran as fast or faster than Julia in all cases.

```
void doGet(HttpServletRequest req, HttpServletResponse
    resp) {
  Random r = new Random();
  if(r.nextInt() == 3)
    resp.getWriter().println(req.getParameter("name"));
}
```

Listing 2: Simplified code of the vulnerability Rivulet misses. `r.nextInt()` returns one of the 2^{32} integers randomly.

```
1private PrintWriter writer;
2void doGet(HttpServletRequest req, HttpServletResponse
    resp) {
3  resp.getWriter().println(req.getParameter("dummy"));
4  //XSS reported on line above
5  this.writer = resp.getWriter();
6  this.writer.println(req.getParameter("other"));
7  //No XSS reported on line above
8}
```

Listing 3: Example of a false negative reported by Julia

Table 2: Comparison between Rivulet and different vulnerability detection tools on the OWASP benchmark. For each vulnerability type, we report the true positive rate and false positive rate for the tool. Each *SAST-0** tool is one of: Checkmarx CxSAST, Coverity Code Advisor, HP Fortify, IBM AppScan, Parasoft Jtest, and Veracode SAST.

	RCE		SQL		XSS	
Tool	TPR	FPR	TPR	FPR	TPR	FPR
SAST-01	35%	18%	37%	13%	34%	25%
SAST-02	67%	42%	94%	62%	67%	42%
SAST-03	59%	35%	82%	47%	49%	22%
SAST-04	72%	42%	83%	51%	66%	40%
SAST-05	62%	57%	77%	62%	41%	25%
SAST-06	100%	100%	100%	90%	85%	45%
Rivulet	100%	0%	100%	0%	100%	0%

Most of Rivulet's time on these benchmarks was spent on the false positive tests, which act as a "worst case scenario" for its execution time: if Rivulet can confirm a flow is vulnerable based on a single attack payload, then it need not try other re-run configurations for that flow. However, on the false positive cases, Rivulet must try every possible payload (in the case of XSS, this is up to 7, although it may also try different encoding strategies for each payload, depending on the source).

Unfortunately, it is not possible to report a direct comparison between Rivulet and any commercial tools (except for Julia) due to licensing restrictions. However, the OWASP benchmark is distributed with anonymized results from applying six proprietary tools (Checkmarx CxSAST, Coverity Code Advisor, HP Fortify, IBM AppScan, Parasoft Jtest, and Veracode SAST) to the benchmark, and we report these results in comparison to Rivulet. Table 2 presents these results (each commercial tool is anonymized), showing the true positive rate and false positive rate for each tool. Rivulet outperforms each of these commercial static analysis tools in both true positive and false positive detection rates.

5.2 RQ2: Rivulet on Large Applications

While the benchmarks evaluated in § 5.1 are useful for evaluating the potential to detect vulnerabilities, they are limited in that they are micro-benchmarks. They help us make general claims about how Rivulet might perform when applied to an arbitrary application. However, since each micro-benchmark is designed to be easily executed (and indeed, we automatically generated tests to execute them), it is not possible to judge how Rivulet performs when using existing, *developer-written*, test cases on real applications.

To provide more detailed results on how Rivulet performs on larger, real applications, we applied it to three different open-source Java web applications and their existing JUnit test suites. *iTrust* is an electronic health record system iteratively developed over 25 semesters by students at North Carolina State University [22, 26]. We evaluated iTrust version 1.23, the most recent version of iTrust1 — a newer "iTrust2" is under development, but has far less functionality than iTrust1 [22]. A prior version of iTrust was also used in the evaluation of Mohammadi *et al.*'s XSS attack testing tool, although the authors were unable to provide a detailed list of the vulnerabilities that they detected or the specific version of iTrust used [40]. We also assessed a recent revision, `8349cebb`, of *Jenkins*, a popular open-source continuous integration server [28], using its test suite. *Struts* is an open-source web application framework library which is used to build enterprise software [3]. Struts is distributed with sample applications that use the framework, as well as JUnit tests for those applications. We evaluated Rivulet with one such sample application (*rest-showcase*), using Struts version 2.3.20_1, which is known to have a serious RCE vulnerability.

Table 3 presents the results of this experiment, showing for each project the number of tests, and then for each injection category the number of vulnerable flows, reruns executed, reruns that succeeded in finding a vulnerability, and the number of unique vulnerabilities found. Rivulet reported no false positives. We briefly discuss the vulnerabilities that Rivulet detected in each application below.

In *iTrust*, Rivulet detected five pages with XSS vulnerabilities, where a user's submitted form values were reflected back in the page. While these values were in only five pages, each page had multiple form inputs that were vulnerable, and hence, Rivulet reported a total of 289 different rerun configurations that demonstrate these true vulnerabilities. There were no flows into SQL queries in iTrust: while iTrust uses a MySQL database, it exclusively accesses it through correct use of the `preparedStatement` API, which is designed to properly escape all parameters. We reported all five vulnerabilities to the *iTrust* developers and submitted a patch.

We also submitted iTrust to the Julia cloud platform for analysis, which produced 278 XSS injection warnings. We did not have adequate resources to confirm how many of these warnings are false positives, but did check to ensure that Julia included all of the XSS vulnerabilities that Rivulet reported. We describe one example that we closely investigated and found to be a false positive reported by Julia. The vulnerability consists of a page with a form that allows the user to filter a list of hospital rooms and their occupants by filtering on three criteria. After submitting the form, the criteria submitted by the user are echoed back on the page without passing through any standard sanitizer, hence Julia raises an alert. While Rivulet did not alert that there was a vulnerability on this page, it did observe the same potentially vulnerable data flow, and generated and executed rerun configurations to test it (not finding it to be vulnerable). We carefully inspected this code to confirm that Rivulet's assessment of these flows was correct, and found that the filter criteria would only be displayed on the page if there were any rooms that matched those criteria. The only circumstances that an exploit could succeed here would be if an administrator had defined a hospital or ward named with a malicious string — in that case, that same malicious string could be used in the filter. While perhaps not a best practice, this does not represent a serious risk — an untrustworthy administrator could easily do even more nefarious actions than create the scenario to enable this exploit.

In *Jenkins*, Rivulet detected a single XSS vulnerability, but that vulnerability was exposed by multiple test cases, and hence, Rivulet created 9 distinct valid test rerun configurations that demonstrated the vulnerability. We contacted the developers of Jenkins who confirmed the vulnerability, assigned it the identifier CVE-2019-10406, and patched it. Jenkins does not use a database, and hence, had no SQL-related flows. We did not observe flows from user-controlled inputs to command execution APIs. Jenkins'

Table 3: Results of executing Rivulet on open-source applications. For each application we show the number of lines of Java code (as measured by `cloc` [14]) the number of test methods, and the time it takes to run those tests with and without Rivulet. For each vulnerability type, we show the number of potentially vulnerable flows detected by Rivulet (**Flows**), the naive number of reruns that would be performed without Rivulet's contextual payload generators (**Reruns$_n$**), the actual number of reruns (**Reruns**), the number of reruns succeeding in exposing a vulnerability (**Crit**), and the number of unique vulnerabilities discovered (**Vuln**). There were no SQL-related flows.

			Time (Minutes)		RCE					XSS				
Application	**LOC**	**Tests**	**Baseline**	**Rivulet**	**Flows**	**Reruns$_n$**	**Reruns**	**Crit**	**Vuln**	**Flows**	**Reruns$_n$**	**Reruns**	**Crit**	**Vuln**
iTrust	80,002	1,253	6	239	0	0	0	0	0	124	117,778	5,424	289	5
Jenkins	185,852	9,330	85	1,140	0	0	0	0	0	534	294,489	13,562	9	1
Struts Rest-Showcase	152,582	15	0.3	5	53	2,609	2,609	4	1	9	6,254	228	0	0

slower performance was caused primarily by its test execution configuration, which calls for every single JUnit test class to execute in its own JVM, with its own Tomcat server running Jenkins. Hence, for each test, a web server must be started, and Jenkins must be deployed on that server. This process is greatly slowed by load-time dynamic bytecode instrumentation performed by Rivulet's underlying taint tracking engine (Phosphor), and could be reduced by hand-tuning Phosphor for this project.

In *Struts*, Rivulet detected a command injection vulnerability, CVE-2017-5638, the same used in the Equifax attack (this vulnerability was known to exist in this revision). Again, multiple tests exposed the vulnerability, and hence Rivulet generated multiple rerun configurations that demonstrate the vulnerabilities. In this revision of struts, a request with an invalid HTTP Content-Type header can trigger remote code execution, since that header flows into the OGNL expression evaluation engine (CVE-2017-5638), and Rivulet demonstrates this vulnerability by modifying headers to include OGNL attack payloads. The struts application doesn't use a database, and hence, had no SQL-related flows.

The runtime for Rivulet varied from 5 minutes to about 19 hours. It is not unusual for automated testing tools (*i.e.,* fuzzers) to run for a full day, or even several weeks [34], and hence, we believe that even in the case of Jenkins, Rivulet's performance is acceptable. Moreover, Rivulet's test reruns could occur in parallel, dramatically reducing the wall-clock time needed to execute it.

5.3 RQ3: Reduction in Reruns

This research question evaluates Rivulet's reduction in the number of reruns needed to test whether a given source-sink flow is vulnerable to an attack compared to a naive approach. To do so, we considered the number of payloads that a more naive attack generator such as Ardilla [32] or Navex [2] might create for each class of vulnerability, and then estimate the number of reruns needed. To estimate the number of payloads used for XSS testing, we referred to the OWASP XSS testing cheat sheet, which has 152 distinct payloads [50]. We assume that for RCE testing, the naive generator would generate the same 12 payloads that Rivulet uses (Rivulet does not use context in these payloads). We assume that the naive generator will also consider multiple encoding schemes for each payload (as Rivulet does). Hence, to estimate the number of reruns created by this naive generator, we divide the number of reruns actually executed by the total number of payloads that Rivulet could create, and then multiply this by the number of payloads that the naive generator would create (*e.g., Reruns*$/7 * 152$ for XSS).

Table 3 shows the number of reruns generated by this naive generator as Reruns$_n$. As expected, Rivulet generates far fewer reruns, particularly with its XSS generator, where it generated $22x$ fewer reruns for Jenkins than the naive generator would have. Furthermore, given that Rivulet took 19 hours to complete on Jenkins, prior approaches that do not use Rivulet's in situ rerun generation would be infeasible for the project. Hence, we conclude that Rivulet's context-sensitive payload generators are quite effective at reducing the number of inputs needed to test if a source-sink flow is vulnerable to attack.

5.4 Threats to Validity

Perhaps the greatest threat to the validity of our experimental results comes from our selection of evaluation subjects. Researchers and practitioners alike have long struggled to establish a reproducible benchmark for security vulnerabilities that is representative of real-world flaws to enable a fair comparison of different tools [34]. Thankfully, in the context of injection vulnerabilities, there are several well-regarded benchmarks. To further reduce the threat of benchmark selection, we used four such benchmarks (Juliet, OWASP, Securibench-Micro and WavSep). Nonetheless, it is possible that these benchmarks are not truly representative of real defects — perhaps we overfit to the benchmarks. However, we are further encouraged because these benchmarks include test cases that expose the known limitations of both Rivulet and Julia: for Rivulet, the benchmark suite contains vulnerabilities that are exposed only non-deterministically, and for Julia, the benchmark suite contains tests that are negatively impacted by the imprecision of the static analysis. To aid reproducibility of our results, we have made Rivulet (and scripts to run the benchmarks) available under the MIT open source license [24, 25].

To demonstrate Rivulet's ability to find vulnerabilities using developer-written tests, we were unable to find any appropriate benchmarks, and instead evaluate Rivulet on several open-source projects. It is possible that these projects are not representative of the wider population of web-based Java applications or their tests. However, the projects that we selected demonstrate a wide range of testing practices: Jenkins topping in with 9,330 tests, and Struts with only 15, showing that Rivulet can successfully find vulnerabilities even in projects with very few tests. We are quite interested in finding industrial collaborators so that we can apply Rivulet to proprietary applications as well, however, we do not have any such collaborators at this time.

6 Related Work

Dynamic taint tracking has been proposed as a runtime approach to detect code injection attacks in production applications, as a sort of last line of defense [8, 21, 38, 54, 58, 63]. However, these approaches are generally not adopted due to prohibitive runtime overhead: even the most performant can impose a slowdown of at least 10–20% and often far more [8, 12, 15, 31]. Although prior work has used the term test *amplification* to refer to techniques that automatically inject exceptions or system callbacks in existing tests [1, 75, 76], we believe that Rivulet is the first to use dynamic taint tracking to amplify test cases.

A variety of automated testing tools have been proposed to detect injection vulnerabilities before software is deployed. These tools differ from black-box testing tools in that they assume that the tester has access to the application server, allowing the tool to gather more precise feedback about the success of any given attack. Kieżun *et al.*'s Ardilla detects SQL injection and XSS vulnerabilities in PHP-based web applications through a white-box testing approach [32]. Ardilla uses symbolic execution to explore different application states, then for each state, uses dynamic taint tracking to identify which user-controlled inputs flow to sensitive sinks, generating attack payloads for those inputs from a dictionary of over 100 attack strings. Similar to Ardilla, Alhuzali *et al.*'s *Navex* automatically detects injection vulnerabilities in PHP code using concolic execution to generate sequences of HTTP requests that reach vulnerable sinks [2]. Rivulet improves on these approaches by leveraging the context of the complete value flowing into each vulnerable sink, allowing it to focus its payload generation to exclude infeasible attack strings. The naive rerun generator that we used as a comparison in our experiments roughly represents the number of attack strings that Ardilla would have tested, showing that Rivulet provides a significant reduction inputs tested. Unlike these systems' automated input generators, Rivulet uses developer-provided functional tests to perform its initial exploration of the application's behavior, a technique that we found to work quite well. If a more robust concolic execution tool were available for Java, it would be quite interesting to apply a similar approach to Rivulet, which could reduce our reliance on developer-provided test cases to discover application behavior.

Other tools treat the application under test as a black-box, testing for vulnerabilities by generating inputs and observing commands as they are sent to SQL servers, or HTML as it is returned to browsers. Mohammadi *et al.* used a grammar-based approach to generate over 200 XSS attack strings, however, our context-sensitive approach considers the location of taint tags within the resulting document, allowing Rivulet to select far fewer payloads for testing [40]. Simos *et al.* combined a grammar-based approach for generating SQL injection attack strings with a combinatorial testing methodology for testing applications for SQL injection vulnerabilities [57]. Thomé *et al.*'s evolutionary fuzzer generates inputs to trigger SQL injection vulnerabilities using a web crawler [70]. Others have considered mutation-based approaches to detect SQL injection [6] and XML injection vulnerabilities [27]. In contrast, Rivulet uses data flow information to target only inputs that flow to vulnerable sinks.

While our work considers injection vulnerabilities that are triggered through code that runs on a web server, other work focuses on injection vulnerabilities that exist entirely in code that runs in client browsers. Lekies *et al.* deployed a taint tracking engine inside of a web browser, traced which data sources could flow into vulnerable sinks, and then generated XSS attacks based on the HTML and JavaScript context surrounding each value at the sink [35]. Rivulet also uses taint tracking to generate attack payloads, expanding this approach to generate SQL and RCE injection attacks, and uses existing test cases to expose non-trivial application behavior.

A variety of static taint analysis approaches have also been used to detect injection vulnerabilities [7, 59, 60, 71]. The most recent and relevant is *Julia*, which uses an interprocedural, flow-sensitive and context-sensitive static analysis to detect injection vulnerabilities [59]. Compared to a dynamic approach like Rivulet, static approaches have the advantage of not needing to execute the code under analysis. However, in the presence of reflection, deep class hierarchies, and dynamic code generation (all of which are often present in large Java web applications), static tools tend to struggle to balance between false positives and false negatives. In our benchmark evaluation, we found that Rivulet outperformed Julia.

While Rivulet uses specialized input generation and attack detection to find code injection vulnerabilities, a variety of fuzzers use taint tracking to instead find program crashes. For instance, BuzzFuzz uses taint tracking to target input bytes that flow to a sink and replace those bytes with large, small, and zero-valued integers [17]. VUzzer takes a similar approach, but records values that inputs are compared to in branches and uses those same values as inputs (*e.g.,* if it sees `if(taintedData[49] == 105)...` it would try value 105 in taintedData byte 49) [52]. Similarly, TaintScope uses fuzzing to detect cases where fuzzed inputs flow through checksum-like routines and uses a symbolic representation of these checksum bytes when generating new inputs in order to pass input validation [72]. Rivulet's key novelties over existing taint-based fuzzers are its context-sensitive input generation which enables the creation of complex, relevant attacks and its attack detectors which report injection vulnerabilities rather than just program crashes.

7 Conclusion

Despite many efforts to reduce their incidence in practice, code injection attacks remain common, and are ranked as #1 on OWASP's most recent list of critical web application vulnerabilities [46]. We have presented a new approach to automatically detect these vulnerabilities before software is released, by amplifying existing application tests with dynamic taint tracking. Rivulet applies novel, context-sensitive, input generators to efficiently and effectively test applications for injection vulnerabilities. On four benchmark suites, Rivulet had near perfect precision and recall, detecting every true vulnerability (except for one pathological case) and raising no false alarms. Using developer-provided integration tests, Rivulet found six new vulnerabilities and confirmed one old vulnerability in three large open-source applications. Rivulet is publicly available under the MIT license [25], and an artifact containing RIVULET and the experiments described in this paper is also publicly available [24].

Acknowledgements

We would like to thank Pietro Ferrara for his assistance running and interpreting the *Julia* experiments. Jonathan Bell's group is funded in part by NSF CCF-1763822, NSF CNS-1844880, and the NSA under contract number H98230-18-D-008.

References

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic Execution of Android Test Suites in Adverse Conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 83–93. https://doi.org/10.1145/2771783.2771786

[2] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V. N. Venkatakrishnan. 2018. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, Berkeley, CA, USA, 377–392. http://dl.acm.org/citation.cfm?id=3277203.3277232

[3] Apache Foundation. 2019. Apache Struts. https://struts.apache.org.

[4] Apache Foundation. 2019. Apache Struts Release History. https://struts.apache.org/releases.html.

[5] Apache Foundation. 2019. Apache Tomcat. https://tomcat.apache.org.

[6] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. 2014. Automated Testing for SQL Injection Vulnerabilities: An Input Mutation Approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2610384.2610403

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[8] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 83–101. https://doi.org/10.1145/2660193.2660212

[9] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53 (February 2010), 66–75. Issue 2.

[10] Steve Bousquet. 2016. Criminal charges filed in hacking of Florida elections websites. http://www.miamiherald.com/news/politics-government/article75670177.html.

[11] Shay Chen. 2014. The Web Application Vulnerability Scanner Evaluation Project. https://code.google.com/archive/p/wavsep/.

[12] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. 2006. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *11th IEEE Symposium on Computers and Communications (ISCC '06)*. IEEE, Washington, DC, USA, 6. https://doi.org/10.1109/ISCC.2006.158

[13] Erika Chin and David Wagner. 2009. Efficient Character-level Taint Tracking for Java. In *Proceedings of the 2009 ACM Workshop on Secure Web Services (SWS '09)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/1655121.1655125

[14] Al Daniel. 2019. cloc: Count Lines of Code. https://github.com/AlDanial/cloc.

[15] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI'10*. USENIX Association, Berkeley, CA, USA, 6. http://dl.acm.org/citation.cfm?id=1924943.1924971

[16] Exploit Database. 2019. Offensive Security's Exploit Database Archive. https://www.exploit-db.com.

[17] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-Based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 474–484. https://doi.org/10.1109/ICSE.2009.5070546

[18] Jeff Goldman. 2016. Researchers Find Russian Hacker Selling Access to U.S. Election Assistance Commission. https://www.esecurityplanet.com/hackers/researchers-find-russian-hacker-selling-access-to-u.s.-election-assistance-commission.html.

[19] Google. 2019. Error-Prone: Catch Common Java Mistakes as Compile-Time Errors. https://github.com/google/error-prone.

[20] Vivek Haldar, Deepak Chandra, and Michael Franz. 2005. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC '05)*. IEEE Computer Society, Washington, DC, USA, 303–311. https://doi.org/10.1109/CSAC.2005.21

[21] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. 2006. Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks. In *SIGSOFT '06/FSE-14*. ACM, New York, NY, USA, 175–185. https://doi.org/10.1145/1181775.1181797

[22] Sarah Heckman, Kathryn T. Stolee, and Christopher Parnin. 2018. 10+ Years of Teaching Software Engineering with Itrust: The Good, the Bad, and the Ugly. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '18)*. ACM, New York, NY, USA, 1–4. https://doi.org/10.1145/3183377.3183393

[23] Matthias Höschele and Andreas Zeller. 2017. Mining input grammars with AUTOGRAM. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. 31–34. https://doi.org/10.1109/ICSE-C.2017.14

[24] Katherine Hough, Gebrehiwet Welearegai, Christian Hammer, and Jonathan Bell. 2020. Revealing Injection Vulnerabilities by Leveraging Existing Tests (Artifact). (2020). https://doi.org/10.6084/m9.figshare.11592033

[25] Katherine Hough, Gebrehiwet Welearegai, Christian Hammer, and Jonathan Bell. 2020. Revealing Injection Vulnerabilities by Leveraging Existing Tests (GitHub). https://github.com/gmu-swe/rivulet.

[26] iTrust Team. 2019. iTrust - GitHub. https://github.com/ncsu-csc326/iTrust.

[27] Sadeeq Jan, Cu D. Nguyen, and Lionel C. Briand. 2016. Automated and Effective Testing of Web Services for XML Injection Attacks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 12–23. https://doi.org/10.1145/2931037.2931042

[28] Jenkins Project Developers. 2019. Jenkins. https://jenkins.io.

[29] Jonathan Hedley. 2019. jsoup: Java HTML Parser. https://jsoup.org/.

[30] JSqlParser Project Authors. 2019. JSqlParser. http://jsqlparser.sourceforge.net/.

[31] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/2151024.2151042

[32] Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. 2009. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE 2009, Proceedings of the 31st International Conference on Software Engineering*. Vancouver, BC, Canada, 199–209.

[33] Tracy Kitten. 2013. Card Fraud Scheme: The Breached Victims. http://www.bankinfosecurity.com/card-fraud-scheme-breached-victims-a-5941.

[34] George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[35] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 Million Flows Later: Large-scale Detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 1193–1204. https://doi.org/10.1145/2508859.2516703

[36] Ben Livshits. 2005. Defining a Set of Common Benchmarks for Web Application Security. In *Proceedings of the Workshop on Defining the State of the Art in Software Security Tools*.

[37] Benjamin Livshits and Stephen Chong. 2013. Towards Fully Automatic Placement of Security Sanitizers and Declassifiers. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 385–398. https://doi.org/10.1145/2429069.2429115

[38] Wes Masri and Andy Podgurski. 2005. Using Dynamic Information Flow Analysis to Detect Attacks Against Applications. In *Proceedings of the 2005 Workshop on Software Engineering for Secure Systems&Mdash;Building Trustworthy Applications (SESS '05)*. ACM, New York, NY, USA, 1–7. https://doi.org/10.1145/1082983.1083216

[39] Rick Miller. 2016. "Foreign" hack attack on state voter registration site. http://capitolfax.com/2016/07/21/foreign-hack-attack-on-state-voter-registration-site/.

[40] M. Mohammadi, B. Chu, and H. R. Lipford. 2017. Detecting Cross-Site Scripting Vulnerabilities through Automated Unit Testing. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 364–373. https://doi.org/10.1109/QRS.2017.46

[41] Lou Montulli and David M. Kristol. 2000. HTTP State Management Mechanism. RFC 2965. https://doi.org/10.17487/RFC2965

[42] National Institute of Standards and Technology. 2017. Juliet Test Suite for Java. https://samate.nist.gov/SRD/testsuite.php.

[43] National Vulnerability Database. 2017. CVE-2017-5638 Detail. https://nvd.nist.gov/vuln/detail/CVE-2017-5638.

[44] National Vulnerability Database. 2019. National Vulnerability Database search for "execute arbitrary commands". https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&query=execute+arbitrary+commands&search_type=all.

[45] University of Maryland. 2019. FindBugs - Find Bugs in Java Programs. http://findbugs.sourceforge.net.

[46] Open Web Application Security Project. 2017. OWASP Top 10 - 2017 The Ten Most Critical Web Application Security Risks. https://www.owasp.org/index.php/Top_10-2017_Top_10.

[47] Open Web Application Security Project. 2019. Expression Language Injection. https://www.owasp.org/index.php/Expression_Language_Injection.

[48] Open Web Application Security Project. 2019. OWASP Benchmark Project. https://www.owasp.org/index.php/Benchmark.

[49] Open Web Application Security Project. 2019. Testing for SQL Wildcard Attacks (OWASP-DS-001). https://www.owasp.org/index.php/Testing_for_SQL_Wildcard_Attacks_(OWASP-DS-001).

[50] Open Web Application Security Project. 2019. XSS Filter Evasion Cheat Sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.
[51] OW2 Consortium. 2019. ASM. https://asm.ow2.io/.
[52] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*. https://www.vusec.net/download/?t=papers/vuzzer_ndss17.pdf
[53] A. Sabelfeld and A. C. Myers. 2006. Language-based Information-flow Security. *IEEE J.Sel. A. Commun.* 21, 1 (Sept. 2006), 5–19. https://doi.org/10.1109/JSAC.2002.806121
[54] Tejas Saoji, Thomas H. Austin, and Cormac Flanagan. 2017. Using Precise Taint Tracking for Auto-sanitization. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security (PLAS '17)*. ACM, New York, NY, USA, 15–24. https://doi.org/10.1145/3139337.3139341
[55] Matthew Schwartz. 2019. Equifax's Data Breach Costs Hit $1.4 Billion. https://www.bankinfosecurity.com/equifaxs-data-breach-costs-hit-14-billion-a-12473.
[56] Ashwin Seshagiri. 2015. How Hackers Made $1 Million by Stealing One News Release. https://www.nytimes.com/2015/08/12/business/dealbook/how-hackers-made-1-million-by-stealing-one-news-release.html?_r=0.
[57] Dimitris E. Simos, Jovan Zivanovic, and Manuel Leithner. 2019. Automated Combinatorial Testing for Detecting SQL Vulnerabilities in Web Applications. In *Proceedings of the 14th International Workshop on Automation of Software Test (AST '19)*. IEEE Press, Piscataway, NJ, USA, 55–61. https://doi.org/10.1109/AST.2019.00014
[58] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. 2013. Diglossia: detecting code injection attacks with precision and efficiency. In *CCS '13*. ACM, New York, NY, USA, 12. https://doi.org/10.1145/2508859.2516696
[59] Fausto Spoto, Elisa Burato, Michael D. Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, and Ciprian Spiridon. 2019. Static Identification of Injection Attacks in Java. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 18 (July 2019), 58 pages. https://doi.org/10.1145/3332371
[60] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. 2011. F4F: Taint Analysis of Framework-based Web Applications. In *OOPSLA '11*. ACM, 16. https://doi.org/10.1145/2048066.2048145
[61] Derek Staahl. 2016. Hack that targeted Arizona voter database was easy to prevent, expert says. http://www.azfamily.com/story/32945105/hack-that-targeted-arizona-voter-database-was-easy-to-prevent-expert-says.
[62] Zhendong Su and Gary Wassermann. 2006. The Essence of Command Injection Attacks in Web Applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 372–382. https://doi.org/10.1145/1111037.1111070
[63] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS XI*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1024393.1024404
[64] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. 2012. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 77–91. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/tang
[65] Terence Parr. 2019. ANTLR. https://www.antlr.org/.
[66] The Apache Software Foundation. 2019. OGNL - Apache Commons OGNL - Developer Guide. https://commons.apache.org/proper/commons-ognl/developer-guide.html.
[67] The Apache Software Foundation. 2019. Security. https://struts.apache.org/security/.
[68] The Eclipse Foundation. 2019. Jetty - Servlet Engine and Http Server. https://www.eclipse.org/jetty/.
[69] The MITRE Corporation. 2019. CWE-601: URL Redirection to Untrusted Site ('Open Redirect'). https://cwe.mitre.org/data/definitions/601.html.
[70] Julian Thomé, Alessandra Gorla, and Andreas Zeller. 2014. Search-based Security Testing of Web Applications. In *Proceedings of the 7th International Workshop on Search-Based Software Testing (SBST 2014)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/2593833.2593835
[71] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *PLDI '09*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/1542476.1542486
[72] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2011. Checksum-Aware Fuzzing Combined with Dynamic Taint Analysis and Symbolic Execution. *ACM Trans. Inf. Syst. Secur.* 14, 2, Article Article 15 (Sept. 2011), 28 pages. https://doi.org/10.1145/2019599.2019600
[73] World Wide Web Consortium. 2017. HTML 5.2. https://www.w3.org/TR/html52/.
[74] World Wide Web Consortium. 2019. Parsing HTML Documents. https://html.spec.whatwg.org/multipage/parsing.html.
[75] Pingyu Zhang and Sebastian Elbaum. 2012. Amplifying Tests to Validate Exception Handling Code. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 595–605. http://dl.acm.org/citation.cfm?id=2337223.2337293
[76] Pingyu Zhang and Sebastian Elbaum. 2014. Amplifying Tests to Validate Exception Handling Code: An Extended Study in the Mobile Application Domain. *ACM Trans. Softw. Eng. Methodol.* 23, 4, Article 32 (Sept. 2014), 28 pages. https://doi.org/10.1145/2652483