LEVERAGING DYNAMIC DATA RELATIONSHIPS TO AMPLIFY SOFTWARE TESTS

KATHERINE HOUGH

Doctor of Philosophy in Computer Science
Khoury College of Computer Sciences
Northeastern University
November, 2024

ABSTRACT

Software testing supports the production of reliable, high-quality software by facilitating the identification and correction of defects. However, writing effective tests and debugging detected failures requires substantial developer effort and expertise. This dissertation explores the use of dynamic data relationships, associations between values in a program at runtime, to enable developers to test and debug their systems more thoroughly with less effort and expertise. We present the following contributions: 1) an approach for using taint tracking to enable existing tests to detect additional classes of defects, 2) an alternative approach for propagating control-flow relationships in taint tracking that can be used to aid in developers' understanding of failures exposed by tests, 3) an approach for using relationships between values to enable parametric fuzzers to more thoroughly exercise system behaviors, and 4) an approach for automatically tracking information flows in modern Java Virtual Machines.

ACKNOWLEDGMENTS

I would to thank Jonathan Bell, Frank Tip, William Robertson, Darko Marinov, and Rohan Padhye for serving on my thesis committee. Your guidance and thoughtful questions have helped steer my research. I would also like to thank the faculty and students of the Programming Research Laboratory for their advice and encouragement. Finally, I would like to thank my family for their support without which this would not have been possible.

CONTENTS

ACRONYMS

API	application programming interface
CSV	comma-separated values
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JAR	Java archive
JCL	Java Class Library
JDK	Java Development Kit
JNDI	Java Naming and Directory Interface
JPMS	Java Platform Module System
JSON	JavaScript object notation
JVM	Java Virtual Machine
LTS	long-term support
OGNL	Object-Graph Navigation Language
OWASP	Open Worldwide Application Security Project
RCE	remote code execution
RSS	resident set size
SQL	Structured Query Language
SQLi	Structured Query Language injection
SSA	static single assignment
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language
XSS	cross-site scripting

1 | INTRODUCTION

Dynamic software testing exercises system behaviors to identify deviations from stakeholder expectations, including functional failures, performance issues, and security vulnerabilities. Testing facilitates the identification and correction of defects, thereby supporting the production of reliable, high-quality software. The Consortium for Information and Software Quality (Krasner, 2022) estimated that software quality issues in the United States cost businesses at least $2.41 trillion in 2022 due to, for example, lost revenue from unplanned outages, reputational damages, and cybersecurity failures. Increased use of third party software and recent attacks targeting software supply chains have made the detection of software defects even more critical (Krasner, 2022). For instance, the "Log4Shell" vulnerability in Apache Log4j (Apache Software Foundation, 2023e), a popular Java logging library, allowed attackers to execute arbitrary code remotely on impacted systems (National Institute of Standards and Technology, 2021). The widespread use of Log4j left many companies vulnerable; according to the Check Point Research Team (2021), at its peak, Log4Shell affected more than 48% of corporate networks globally.

A software test observes the behavior of a system when supplied with a specific input in a controlled environment (Claessen and Hughes, 2000; Barr et al., 2015; Wright et al., 2020). A test "oracle" then compares the observed behavior against the expected behavior of the system and reports any violations (Claessen and Hughes, 2000; Barr et al., 2015). Academic experiments and reports from industry have demonstrated the effectiveness of software testing techniques for identifying software defects (Michał Zalewski, 2024; OSS-Fuzz Contributors, 2024; LLVM Project, 2024; Padhye et al., 2019b; Holler et al., 2012; Pham et al., 2021; Wright et al., 2020; Williams et al., 2009). However, writing effective tests and debugging detected failures requires substantial developer time and effort. Furthermore, testing for certain classes of defects, such as injection vulnerabilities, requires specialized expertise. An ineffective or incomplete test suite may fail to exercise unexpected corner cases or lack the ability to detect specific types of defects. It is, therefore, important to develop automated approaches that improve upon existing testing techniques to enable developers to test their systems more thoroughly with less effort and expertise. To this end, we propose the following thesis:

> *K*nowledge of dynamic data relationships can be leveraged to amplify existing software tests to reduce the amount of developer effort and expertise needed to detect and correct defects.

We use the term "dynamic data relationship" to mean any association between two values in a program at runtime. One general-purpose analysis for tracing relationships between values at runtime is dynamic taint tracking (Schwartz et al., 2010; Bell and Kaiser, 2014; Chandra and Franz, 2007; Sabelfeld and Myers, 2006; McCamant and Ernst, 2008; Clause et al., 2007; Enck et al., 2010a). Taint tracking systems monitor the flow of information through a system, allowing users to determine whether one value is derived from another. Other specialized analyses can be used to track application-specific relationships between values. In this manuscript, we show that these analyses for tracing relationships between values can be used to guide test input generation, enable existing tests to precisely detect additional classes of defects, and aid in developers' understanding of failures exposed by tests.

The remainder of this manuscript is organized as follows. In Chapter 2, we present an approach for detecting injection vulnerabilities in web applications using a combination of dynamic taint tracking and test generation. This approach, Rivulet, monitors the execution of developer-written functional tests in order to detect information flows that may be vulnerable to injection attacks. Then, Rivulet uses a white-box test generation technique to repurpose those functional tests to check if a candidate flow can be exploited. Chapter 3 describes Conflux—an alternative approach for propagating control-flow relationships in taint tracking that can be used to aid in developers' understanding of failures exposed by tests. Conflux precisely tracks control-flow relationships by decreasing the scope of influence for control flows and providing a heuristic for reducing loop-related imprecise. In Chapter 4, we describe parametric fuzzing, an automated technique for generating structured test inputs, and present linked crossover, a novel approach for using relationships between values to enable parametric fuzzers to more thoroughly exercise system behaviors. Linked crossover uses dynamic execution information to identify and exchange analogous portions of inputs, allowing it to combine advantageous traits from multiple inputs. Next, in Chapter 5, we present Galette, an approach for dynamic taint tracking in the Java Virtual Machine (JVM) that provides precise, robust taint tag propagation in modern JVMs. Galette addresses limitations of existing approaches for dynamic taint tracking in the JVM, enabling tools that rely on dynamic taint tracking, like Rivulet and Conflux, to be built for the latest Java versions. Finally, we discuss related work in Chapter 6 and we conclude this manuscript in Chapter 7.

2 REVEALING INJECTION VULNERABILITIES BY LEVERAGING EXISTING TESTS

2.1 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 (Schwartz, 2019). This attack leveraged an injection vulnerability in Apache Struts (CVE-2017-5638) and is just one of thousands of similar injection vulnerabilities discovered in recent years in popular software (National Vulnerability Database, 2019). Injection vulnerabilities are a class of defects in which an application interprets an untrusted, potentially malicious input as a command or query. A successful injection attack can allow the attacker to run arbitrary code on a vulnerable system, as was the case in the 2017 Equifax attack. Therefore, injection vulnerabilities are considered a major risk, placing third in the 2021 ranking of critical security risks to web applications by the Open Worldwide Application Security Project (OWASP). 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 complex attacks.

Successful injection attacks often need to modify the syntactic structure of a query, document, or command from what was intended by the developer, changing the structure from one in which a malicious input is treated like data to one in which the malicious input is treated like code. Developers can guard against this by neutralizing malicious inputs using sanitization and validation (Livshits and Chong, 2013). However, input neutralization is challenging to implement properly (Livshits and Chong, 2013). It is therefore critical to detect when missing or improper input neutralization leaves an application vulnerable to injection attacks.

Static approaches are able to detect injection vulnerabilities without executing the system under analysis (Google, 2019; of Maryland, 2019; Bessey et al., 2010). However, these approaches are prone to reporting false positives due to imprecise inter-procedural analyses (Spoto et al., 2019; Tripp et al., 2009; Sridharan et al., 2011; Arzt et al., 2014). Runtime monitoring tools are generally more precise and stop injection attacks just-in-time by preventing untrusted values from reaching sensitive program locations (Son et al., 2013; Halfond et al., 2006; Suh et al., 2004; Saoji et al., 2017; Bell and Kaiser, 2014; Masri and Podgurski, 2005). However, these tools often have unacceptably high runtime overheads; even the most performant can impose a slowdown of at least 10–20% (Cheng et al., 2006; Bell and Kaiser, 2014; Kemerlis et al., 2012; Enck et al., 2010b). Fuzzing-based approaches allow vulnerabilities to be detected before an application is deployed. However, they may be unable to generate inputs that exercise vulnerable behaviors in complex or stateful applications (Kieżun et al., 2009; Höschele and Zeller, 2017). All of these approaches have difficulties distinguishing between proper and improper uses validation and sanitization procedures.

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

Our test amplification approach uses a 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 (Open Web Application Security Project, 2019d,c; Kieżun et al., 2009). By leveraging the context of how that user-controlled value is used in security-sensitive parts of the application, we can rule out many candidate attack strings, 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 does not require access to application's source code and runs in commodity, off-the-shelf JVMs, by 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 (Spoto et al., 2019) on several established 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 (XSS) vulnerability, which was confirmed by the developers. On the educational project iTrust (Heckman et al., 2018), Rivulet found five previously unknown vulnerabilities. Unlike the state-of-the-art static analysis tool that we used, Julia (Spoto et al., 2019), Rivulet did not report any false positives.

Prior approaches for dynamically detecting injection vulnerabilities have been limited by two key challenges. Firstly, dynamic analyses require a representative workload to exercise the application under analysis. Secondly, for each potential attack vector, there may be hundreds of input strings that should be checked. Rivulet addresses these challenges, making the following 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 Structured Query Language injection (SQLi), Object-Graph Navigation Language (OGNL) injection, and XSS that handle complex, stateful applications
- Embedded attack detectors to verify whether rerunning a test with new inputs leads to valid attacks

2.2 BACKGROUND

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 SQLi, where attackers can control the contents of a Structured Query Language (SQL) statement. For instance, consider this 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 + "'");`. The name supplied by the user is concatenated with the query string and then executed on the database server. For example, the value `Bob` will result in the clause `where name = 'Bob';` being executed on the server. However, if the `name` variable is controlled by a user, the user may modify the query more broadly. If an attacker can arbitrarily control the value of the `name` variable, then they may perform a SQLi attack. For instance, the attacker could supply the value `Bob' OR '1'='1` which will produce `where name = 'Bob' OR '1'='1'` when joined to the query string. This would result in all rows in the table being selected. SQLi attacks can result in data breaches, denial of service attacks, and privilege escalations.

Remote code execution (RCE) vulnerabilities are a form of injection vulnerability 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 Expression Language (Open Web Application Security Project, 2019a) or OGNL (Apache Software Foundation, 2019c,d).

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 Hypertext Markup Language (HTML), JavaScript code, or both into the response returned by the server-side application. A successful XSS attack results in malicious code being included as part of a web page in a browser. Such an attack might hijack a user's session allowing the attacker to impersonate the user on that website, steal sensitive data, or inject keyloggers. 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. A user may recognize the link as a trusted site, but upon clicking the link, the attacker's payload will be loaded as part of the server's response and executed by the client's browser. 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, even without a specially crafted attack link.

Developers can guard against injection attacks by neutralizing malicious inputs using sanitization and validation (Livshits and Chong, 2013). Validation checks whether an input is well-formed. Sanitization is a sequence of transformations that aim to render attacks harmless by removing or altering unwanted values. All potentially malicious data should be properly neutralized before it reaches sensitive program locations, known as "sinks", that may interrupt that data as code. However, ensuring this is non-trivial (Livshits and Chong, 2013). Hence, the key challenge in detecting injection vulnerabilities is to detect information flows from untrusted sources to sensitive sinks that have not been properly neutralized.

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

Listing 2.1 shows an example of two XSS vulnerabilities. On Lines 9 and 10, a user-controlled parameter flows into the response sent back to the browser without proper neutralization. In the first case (line 9), a vulnerability occurs despite an attempt to sanitize the user's input using the function `escapeHtml4`. In the second case (line 10), no attempt is made to neutralize the input. In both cases, providing the input

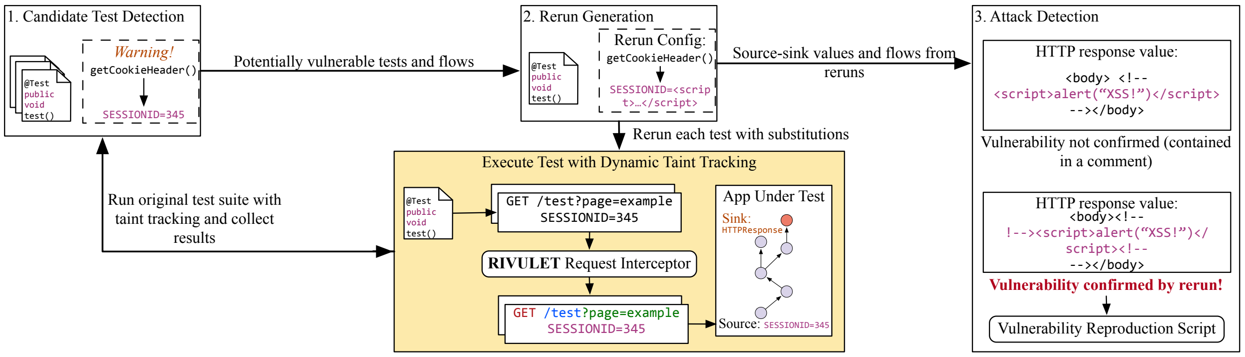

1. Candidate Test Detection
Warning!
getCookieHeader()
SESSIONID=345
@Test public void test()
Potentially vulnerable tests and flows
2. Rerun Generation
Rerun Config:
getCookieHeader()
SESSIONID=<script>…</script>
Source-sink values and flows from reruns
3. Attack Detection
HTTP response value:
<body> <!--<script>alert("XSS!")</script>--></body>
Vulnerability not confirmed (contained in a comment)
HTTP response value:
<body><!--!--><script>alert("XSS!")</script><!----></body>
Vulnerability confirmed by rerun!
Vulnerability Reproduction Script
Rerun each test with substitutions
Execute Test with Dynamic Taint Tracking
GET /test?page=example SESSIONID=345
RIVULET Request Interceptor
App Under Test
Sink: HTTPResponse
Source: SESSIONID=345
Run original test suite with taint tracking and collect results

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.

string `javascript:alert('XSS');` for the parameter `"name"` will result in unexpected JavaScript code executing in the client's browser. The method `escapeHtml4` is insufficient to prevent this attack. This method escapes any special HTML characters in the input string. This would prevent an injection attack that included something like a `<script>` tag, but is insufficient for this case. By using the prefix `javascript:` in their payload, an attacker is able to subvert the sanitizer in this context. For this reason, many known XSS attack payloads do not include brackets or quotes (Open Web Application Security Project, 2019d).

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 (Spoto et al., 2019), typically assume that library methods pre-defined as sanitizers for a class of attack (e.g., XSS sanitizers) eliminate vulnerabilities for the all information flows into sinks related to that class of vulnerabilities. Rivulet does not make this assumption. Instead, Rivulet tests whether a flow is adequately sanitized by attempting to generate a counterexample (i.e., a malicious payload that produces a successful injection attack).

2.3 APPROACH

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 (Kieżun et al., 2009; Höschele and Zeller, 2017). 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 (Section 2.6), 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 neutralization function correctly protects the application. Hence, when a source-sink flow is observed, Rivulet generates malicious payloads based on contextual information extracted from the observed call to the sink method. Then, tests are re-executed replacing untrusted source values with generated payloads to probe for weak or missing input neutralization. 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 neutralization efforts without requiring these efforts to be explicitly enumerated or modeled. Verifying whether neutralization efforts are adequate in all cases is beyond the scope of this work. However, if a system fails to properly neutralize an input before it flows into a sink method, then one of Rivulet's malicious payloads may be able to demonstrate a successful attack, causing the flow to be verified. Our implementation of Rivulet (described in Section 2.4) automatically detects SQLi, RCE, and XSS vulnerabilities; developers do not need to specify any additional sources or sinks in order to find these kinds of vulnerabilities. Section 2.4 describes, in detail, the specific strategy that Rivulet uses to find these kinds of vulnerabilities.

2.3.1 Detecting Candidate Tests

Rivulet co-opts existing, functional test cases to check 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 input neutralization efforts. 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 and authentication (Kieżun et al., 2009).

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 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.

2.3.2 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. Each of an application's test cases may expose dozens of potentially vulnerable flows. Therefore, it is crucial to limit the number of times that the test needs to be perturbed and re-executed to evaluate the flow. Unfortunately, it is typical to consider dozens of different malicious payloads for each potentially vulnerable input (Open Web Application Security Project, 2019d; Kieżun et al., 2009).

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 by leveraging contextual information to create more effective payloads. Successful injection attacks often need to modify the syntactic structure of a query, document or command from what was intended by the developer (Su and Wassermann, 2006). 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. Thus, a source-sink flow can be evaluated by testing only payloads that could succeed in a particular context without exhaustively testing all payloads possibly capable of disrupting the type of structure expected by the sink.

For instance, when an untrusted value reaches a sink method vulnerable to SQLi, developers usually intend for the value to be treated as a string or numeric literal. Consider the 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 describe in Section 2.4.2.

2.3.3 Attack Detection

The attack detector component provides the oracle for each modified test (replacing the original test oracle), 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 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.

2.4 IMPLEMENTATION

Our implementation of RIVULET for Java is built using the PHOSPHOR taint tracking framework (Bell and Kaiser, 2014) (version 0.0.4-SNAPSHOT), 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. Out of the box, RIVULET detects SQLi, XSS, and OGNL injection vulnerabilities without any additional configuration. PHOSPHOR propagates taint tags by rewriting Java bytecode using the ASM bytecode instrumentation and analysis framework (OW2 Consortium, 2024), 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.

2.4.1 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 application programming interface (API) methods as taint sources, such as `ServletRequest.getParameter()` for parameters or `HttpServletRequest.getCookies()` for cookies (Spoto et al., 2019; Mohammadi et al., 2017; Chin and Wagner, 2009; Franz et al., 2005). However, these approaches can be brittle: if a single source is missed or a new version of the application engine is used, 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 (Montulli and Kristol, 2000).

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 (Apache Software Foundation, 2024) and Jetty (Eclipse Foundation, 2019), 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 a "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.

2.4.2 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, if any, 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 six through ten of the second call to the source `getQueryString()`. A payload replacement for this flow would indicate that the return value of the second call to `getQueryString()` 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.

Structured Query Language Injection

The rerun generator for SQLi uses all methods declared in `java.sql.Statement` and `java.sql.Connection` 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 between two and five SQLi payloads for a tainted value in a particular context out of twenty unique payloads that could be generated for the same tainted value across all of the contexts considered by the SQLi rerun generator. If wildcard payloads for `LIKE` clause are not generated then between two and three payloads are generated per context. This is a reduction from Kieżun et al. (2009)'s tool which uses six SQLi patterns and does not consider tainted backtick-quoted values, comments, or `LIKE` clauses.

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 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 (i.e., its “Content-Type” header’s value is “text/html” or not specified) 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 for parsing and manipulating HTML (Jonathan Hedley, 2024). This model is traversed, generating payloads for each tainted value encountered. The XSS rerun generator considers five 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 and whether content is contained in an element which causes the tokenizer to leave the data state during parsing (World Wide Web Consortium, 2019).

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 between three and seven XSS payloads for a tainted value in a particular context out of over one hundred 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 (Open Web Application Security Project, 2019d) and Kieżun et al. (2009) use 106 patterns for creating XSS attacks.

Command and Object-Graph Navigation Language Injection

The command injection rerun generator creates payloads with common UNIX commands like `ls`, and considers the methods declared in `java.lang.ProcessBuilder` and `java.lang.Runtime` 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 (Apache Software Foundation, 2019c). Applications using OGNL can limit the code that user-specified expressions can execute by whitelisting or blacklisting certain patterns (Apache Software Foundation, 2019d). However, the evaluation of improperly neutralized OGNL expressions can enable a user to execute arbitrary code. The OGNL rerun generator uses payloads that we collected from the Exploit Database (Exploit Database, 2019) and simplified to integrate more tightly with Rivulet's attack detection mechanism.

2.4.3 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 replacement value is appropriate (e.g., converting between a string and a character array).

2.4.4 Attack Detection

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

Structured Query Language Injection

Our approach for detecting SQLi builds on Halfond et al. (2006)'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. We determine a SQLi 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 SQLi. 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 (Open Web Application Security Project, 2019c). The attack detector for SQLi uses the ANTLR parser generation tool (Terence Parr, 2019) and JSqlParser (JSqlParser Project Authors, 2019), a SQL statement parser that supports multiple SQL dialects, to parse SQL statements that reach sink methods vulnerable to SQLi attacks.

Cross-Site Scripting

The World Wide Web Consortium (2017)'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". 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:

- The inner content of every "script" tag is classified as an embedded script.
- The HTML-entity-decoded value of every "src" attribute specified for a "script" tag is classified as an external script's address.
- 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 Uniform Resource Locator (URL)s.
- The HTML-entity-decoded value of every event handler attribute (e.g., "onload") specified for any tag is classified as an embedded script.
- The HTML-entity-decoded value of every attribute listed as having a URL value by World Wide Web Consortium (2017) (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, for instance `alert` is an appropriate target string for the payload `<script>alert(1)</script>`. 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 the jsoup HTML parser (Jonathan Hedley, 2024)) 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 (Terence Parr, 2019) and a simplified grammar for JavaScript to identify string literals, template literals, and comments. The application of the checking rules is implemented in a way that supports easy reconfiguration to allow for the addition or removal of script-triggering mechanisms and vulnerable attributes.

Command and Object-Graph Navigation Language 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.

2.5 LIMITATIONS

RIVULET is not 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. In our evaluation, we show that RIVULET can detect a real vulnerabilities even when presented with small test suites. This limitation could be mitigated by integrating our approach with an automatic test generation technique.

Vulnerabilities caused by a non-deterministic flows 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 (MITRE Corporation, 2019). 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.

Our implementation of RIVULET uses the PHOSPHOR dynamic taint tracking framework (version 0.0.4-SNAPSHOT) to trace information flows (Bell and Kaiser, 2014). This version of PHOSPHOR only supports Java versions 8 or lower. Therefore, our implementation of RIVULET can only be used on Java versions 8 or lower. In Java version 9, the Java Platform Module System (JPMS) was introduced to the JVM adding support for modules—groupings of related packages and resources (Oracle Corporation, 2024h). At runtime, the JVM enforces certain constraints on these modules and their usage. These constraints prevent PHOSPHOR from running on newer Java versions. Furthermore, newer Java versions make greater use of signature polymorphic methods, special methods that follow alternative linkage and invocation semantics (Lindholm et al., 2013). The design of PHOSPHOR is ill-suited for handling these semantics. In Chapter 5, we present an approach for dynamic taint tracking in modern JVMs that addresses these changes to the JVM, enabling tools like RIVULET to be built for newer Java versions.

Since PHOSPHOR is unable to track taint tags through code outside 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. At present, RIVULET can only detect vulnerabilities that result from explicit flows, and not from implicit flows or side-channels (Sabelfeld and Myers, 2006).

Our implementation of RIVULET only detects SQLi, XSS, OGNL injection, and certain types of RCE vulnerabilities. However, RIVULET could be extended to detect additional classes of vulnerabilities by writing custom attack detectors and rerun generators. For instance, a generator for Java Naming and Directory Interface (JNDI) injection could be created to allow RIVULET to detect JNDI vulnerabilities, like the one used in the high profile “Log4Shell“ vulnerability (National Institute of Standards and Technology, 2021).

2.6 EVALUATION

We performed an empirical evaluation of Rivulet to answer the following research questions:

RQ1: How accurately does Rivulet detect injection vulnerabilities in established benchmark suites?

RQ2: Does Rivulet scale to real, open-source projects and their test suites?

RQ3: How significantly does Rivulet's contextual payload generation reduce the number of reruns needed?

2.6.1 Methodology

Baselines

To evaluate RQ1, we compared Rivulet against Spoto et al. (2019) 's state-of-the-art static analysis tool, Julia. Unfortunately, it was not possible for us to perform a direct comparison between Rivulet and other commercial tools due to licensing restrictions. However, one benchmark suite, the OWASP benchmark suite, is distributed with anonymized results for six proprietary tools (Checkmarx CxSAST, Coverity Code Advisor, HP Fortify, IBM AppScan, Parasoft Jtest, and Veracode SAST) on the suite. We used these results to examine Rivulet's performance compared to a broader set of analysis tools.

For RQ3, we compared the number of reruns used by Rivulet to evaluate observed source-sink flows against the number of payloads that a naive, non-contextual attack generator, such as Kieżun et al. (2009)'s or Alhuzali et al. (2018)'s, would create for the same set of flows. To estimate the number of payloads used for XSS-related flows, we referred to the Open Web Application Security Project (2019d)'s XSS testing cheat sheet, which contains 152 distinct payloads. To estimate the number of payloads for SQLi-related flows, we took the three base payloads that Rivulet uses and multiplied that by the four different quote styles (described in Section 2.4.2) to get to twelve distinct payloads. We assume that for RCE-related flows, the naive generator would generate the same twelve payloads that Rivulet uses, because Rivulet does not use contextual information to generate RCE-related 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 a naive generator, we divide the number of reruns performed by Rivulet by the total number of payloads that Rivulet could create for a flow, and then multiply by the number of payloads that the naive generator would create.

Subjects

For RQ1, we used several established suites of vulnerability detection benchmarks: the National Institute of Standards and Technology (2017)'s Juliet benchmark suite (version 1.3), the Open Web Application Security Project (2019b)'s (OWASP) benchmark suite (version 1.2), Livshits (2005)'s Securibench-Micro, and Chen (2014)'s Web Application Vulnerability Scanner Evaluation Project (WAVSEP) benchmark suite (version 1.5). These suites are intentionally seeded with vulnerabilities. Each suite cans contain both true alarm tests, which are vulnerable to injection attacks, and false alarm tests which are not vulnerable.

Each suite consists of a series of web servlets and applications that are well-suited for static analysis. However, dynamic analyses like Rivulet require executable workloads. Each servlet is designed to be its own standalone application. 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, most of which were related to invalid SQL syntax in the benchmark. We ignored several tests from Securibench-Micro that were not suitable for dynamic analysis (e.g., tests containing infinite loops), and otherwise included only tests for the vulnerabilities targeted by Rivulet (RCE, SQLi, and XSS). For transparency and reproducibility, all benchmark code is included in the artifact for this work (Hough et al., 2020b).

These suites 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 open-source applications and their test suites for RQ2 and RQ3: iTrust, Jenkins, and Apache Struts. iTrust is an electronic health record system iteratively developed over 25 semesters by students at North Carolina State University (Heckman et al., 2018; iTrust Team, 2019). We evaluated iTrust version 1.23, the most recent version of iTrust1. A prior version of iTrust was also used in the evaluation of Mohammadi et al. (2017)'s XSS attack testing tool. We also assessed Jenkins (version 8349cebb) (Jenkins Project Developers, 2019), a popular open-source continuous integration server, using its test suite. Apache Struts is an open-source web application framework library which is used to build enterprise software (Apache Foundation, 2019). 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.

Metrics

For RQ1, we collected the number true positives, false positives, true negatives and false negatives recorded for Rivulet and Julia on each benchmark suite. If a tool reported a true alarm test as vulnerable, the test was counted as a true positive. Otherwise, if the tool did not report a true alarm test as vulnerable, the test was counted as a false negative. Similarly, if a tool reported a false alarm test as vulnerable, the test was counted as a false positive. Otherwise, if the tool did not report a false alarm test as vulnerable, the test was counted as a true negative. For RQ2, we report the time taken to run each application's original test suite, the time taken by Rivulet to analyze the application, the number of candidate source-sink flows detected by Rivulet, and the number of developer-confirmed vulnerabilities detected by Rivulet. For RQ3, we report the number of reruns performed by Rivulet and the estimated number of reruns that a naive generator would need to perform as described in Section 2.6.1.

Experimental Setup

We conducted 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 through the JuliaCloud web portal, using the most recent version publicly available as of August 2019. When available (for parts of the Juliet benchmark and all of the OWASP benchmark), we re-use results reported by the Julia authors (Spoto et al., 2019). 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.

Table 1: Comparison of Rivulet and Julia on established benchmark suites. For each vulnerability type (Type) in each benchmark suite (Suite), we report the number of test cases that are vulnerable (True Alarm) and not vulnerable (False Alarm) to injection attacks. For Rivulet and Julia, we report the number of true positives (TP), false positives (FP), true negatives (TN), false negatives (FN), and analysis time in minutes (Time). Times are aggregate for the whole benchmark suite.

				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
	SQLi	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
	SQLi	272	232	272	0	232	0		272	36	196	0	
	XSS	246	209	246	0	209	0		246	19	190	0	
Securibench-Micro	SQLi	3	0	3	0	0	0	1	3	0	0	0	1
	XSS	86	21	85	0	21	1		77	14	7	9	
WAVSEP	SQLi	132	10	132	0	10	0	2	132	0	10	0	2
	XSS	79	7	79	0	7	0		79	6	1	0	

2.6.2 Results

RQ1: Accuracy

Table 1 reports the accuracy of Rivulet and Julia on the four benchmark suites that we examined. 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 2.2), the probability of any attack succeeding on the test was just $\frac{1}{2^{32}}$, and Rivulet could not detect the vulnerability within a reasonable time bound. This test case likely represents the worst-case pathological application that Rivulet could encounter.

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

Listing 2.2: Simplified code of the vulnerability Rivulet misses. The call to `r.nextInt()` returns one of the 2^{32} integers randomly.

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 over-tainting 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 2.3 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`. We note

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

	RCE		SQLi		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

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.

```java
private PrintWriter writer;
void doGet(HttpServletRequest req, HttpServletResponse resp) {
    resp.getWriter().println(req.getParameter("dummy"));
    // A vulnerability was reported on the line above
    this.writer = resp.getWriter();
    this.writer.println(req.getParameter("other"));
    // No vulnerability was reported on the line above
}
```

Listing 2.3: Example of a false negative reported by Julia.

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. 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 applicable payload.

Table 2 presents a comparison of Rivulet's performance against the anonymized results of commercial analysis tools included with the OWASP benchmark suite. Rivulet outperforms each of these commercial analysis tools in both true positive and false positive detection rates.

RQ2: Performance on Open-Source Applications

Table 3 presents Rivulet's performance on the three open-source applications. Rivulet reported no false positives on any of the applications. We briefly discuss the vulnerabilities that Rivulet detected in each application below.

Table 3: Results of executing RIVULET on open-source applications. For each application (App) we show the number of lines of code as measured by `cloc` (Daniel, 2019) (LOC), the number of test methods (Tests), and the time it took to run those tests with (RIVULET) and without (Base) RIVULET in minutes. For each vulnerability type, we report 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 (R_n), the actual number of reruns (R), the number of reruns succeeding in exposing a vulnerability (Crit), and the number of unique vulnerabilities discovered (Vul). No SQLi-related flows were detected for any of the applications.

			Time		RCE					XSS				
App	LOC	Tests	Base	RIVULET	Flows	R_n	R	Crit	Vuln	Flows	R_n	R	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	152,582	15	0.3	5	53	2,609	2,609	4	1	9	6,254	228	0	0

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: ward, status and hospital. 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 report this page as vulnerable, it did observe the same potentially vulnerable data flow, and executed reruns to test it ultimately finding that it that this page was not 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 which was exposed by multiple test cases, and hence, RIVULET created nine 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 SQLi-related flows. We did not observe flows from user-controlled inputs to command execution APIs. Jenkins' 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 one 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 vulnerability. 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 SQLi-related flows.

The runtime for Rivulet varied from five minutes to about nineteen hours. It is not unusual for automated testing tools to run for a full day, or even several weeks (Klees et al., 2018), 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.

RQ3: Reduction in Reruns

Table 3 compares the number of reruns uses by Rivulet to test whether a given source-sink flow is vulnerable to an attack compared to a naive approach. As expected, Rivulet generates far fewer reruns, particularly with its XSS generator, where it generated twenty-two times fewer reruns for Jenkins than the naive generator would have. Furthermore, given that Rivulet took nineteen 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.

2.6.3 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 (Klees et al., 2018). To reduce the threat of benchmark selection, we used multiple, well-established, third-party benchmark suites. Nonetheless, it is possible that these benchmark suites are not truly representative of real defects. However, we are further encouraged because these suites include test cases that expose the known limitations of both Rivulet and Julia. For Rivulet, the benchmark suites contain vulnerabilities that are exposed only non-deterministically, and for Julia, the benchmark suites contain 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 publicly available under a MIT license (Hough et al., 2020b).

To demonstrate Rivulet's ability to find vulnerabilities using developer-written tests, we were unable to find any appropriate benchmarks, and instead evaluated 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 fifteen, showing that Rivulet can successfully find vulnerabilities even in projects with small test suites.

2.7 CONCLUSION

Despite developers' efforts to reduce their incidence in practice, injection vulnerabilities remain common, placing third in the 2021 ranking of critical security risks to web applications by OWASP. Rivulet our novel approach for new automatically detects 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 almost every true vulnerability and raising no false alarms. Using developer-provided tests, Rivulet found six new vulnerabilities and confirmed one old vulnerability in three open-source applications.

2.8 STATUS

This work was published in the *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (ICSE 2020) (Hough et al., 2020a). It was presented at ICSE 2020. The associated artifact for this work was evaluated and awarded the “Available” and “Reusable” badges (Hough et al., 2020b).

3 A PRACTICAL APPROACH FOR DYNAMIC TAINT TRACKING WITH CONTROL-FLOW RELATIONSHIPS

3.1 INTRODUCTION

Taint tracking is a technique for monitoring the flow of information through a system. Traditionally, it has been used in privacy analyses to prevent confidential data from leaking into a program's public outputs and in security analyses to detect the flow of untrusted values into sensitive program locations (Schwartz et al., 2010; Sabelfeld and Myers, 2006). In recent years, it has also been applied to other software development tasks, for instance, assisting automated input generation systems (fuzzers) (Rawat et al., 2017), helping to identify poorly-designed software tests (Huo and Clause, 2014), providing debugging guidance (Clause and Orso, 2009; Attariyan and Flinn, 2010), and creating performance models for configurable systems (Velez et al., 2021).

Dynamic taint tracking associates labels (also referred to as taint tags) with program data and propagates these labels through the system during the execution of a program. The set of rules defining how taint tags propagate when an operation executes forms the tainting tracking system's propagation policy. This policy effectively describes what it means for information to "flow" through a program. Most taint tracking systems focus on tracing the flow of data through assignment, arithmetic, and logical operations which directly pass information from their operands to their result. This direct passage of information is referred to as an explicit or data flow (Sabelfeld and Myers, 2006). In a data flow, the value of the flow's target, the operation's result, is derived from the value of flow's source, the operands of the operation. For instance, in line three of Listing 3.1 there is a data flow from `x` and `y` to `z`. Thus, the labels of `x` and `y` should propagate to `z`.

```
1 int x = taint(4, "X");
2 int y = taint(8, "Y");
3 int z = x + y;
4 int q = 0;
5 if(y == 8) {
6     q = 1;
7 }
```

Listing 3.1: A basic taint tracking example.

However, tracking only these explicit flows can provide an incomplete picture of the flow of information through the system. For example, one might expect `q` to be tainted after the code in Listing 3.1 executes, since it is clear that `q`'s value reveals `y`'s value. This indirect passage of information between values can occur as a result of conditional branches, array operations, and pointer dereferencing and is referred to as an implicit flow (McCamant and Ernst, 2008). Unlike a data flow, the value of an implicit flow's target is not related to the value of its source through some computation. Instead, the value of the implicit flow's source is used to "select" the value of the implicit flow's target. For example, if a tainted index is used to access an element of an array, then the retrieved element's value is not directly derived from the value of the tainted index. However, the retrieved element is selected from the collection of elements in the array as a result of the value of the tainted index. The lack of direct computational relationship between the target of an implicit flow and its source can mean that little to no information is passed along the flow. For example, consider a situation where a tainted index is used to access an element of an array containing the same item at every position. If the tainted index's value is guaranteed to be within the bounds of the array, then the value of the accessed element is not influenced by the value of the tainted index. In this case, propagating labels from the tainted index to the accessed element falsely conveys a relationship between the two values. This is typically referred to as over-tainting.

Implicit flows resulting from conditional branches are specifically referred to as control flows since information is passed via the control structure of the program. The source of a control flow is a tainted branch condition that guards the execution of an assignment statement. The branch condition's value impacts whether the assignment statement executes and therefore selects whether a new value is assigned to the statement's destination storage location. Like other implicit flows, not propagating along control flows can result in critical data relationships being lost, which is referred to as under-tainting. For instance, when looking up the value associated with a tainted key in an associative-array data structure, there is likely a control flow, but not a data flow between the key and the value. Thus, many existing taint tracking systems support the propagation of taint tags through control flows (Clause et al., 2007; Bell and Kaiser, 2014, 2015; Chandra and Franz, 2007).

The standard semantics for propagating control flows propagates the taint tag of a branch's predicate to every value written by an assignment statement whose execution is controlled by that branch. However, prior work has found that using the standard control flow propagation semantics resulted in severe over-tainting making it impractical for their applications (Bao et al., 2010; Clause and Orso, 2009; Staicu et al., 2019; Attariyan and Flinn, 2010). For example, in their tool for debugging configuration errors, Attariyan and Flinn (2010) reported that "a strict definition of causal dependencies [control flows] led to our tool outputting almost all configuration values as the root cause of the problem." Clause and Orso (2009) discovered that using control flow tracking in Penumbra, a tool for identifying inputs that are relevant to a failure, resulted in larger failure-relevant input sets and in the case of one application resulted in almost all the program's roughly fifteen million inputs being marked as failure-relevant. We performed a case study of Penumbra (detailed in Section 3.6.5) and observed a similar result: propagating along control flows resulted in impractically large failure-relevant input sets. We also found that this over-tainting could be reduced by using alternative control flow propagation semantics. However, due to the over-tainting that occurs when using the standard control flow propagation semantics, existing software engineering tools that use taint analysis typically ignore control flows, favoring precision over recall. This over-tainting is not caused by a defect in the taint tracking system, but by a mismatch between the standard control flow propagation semantics and the expectations of downstream analyses. In particular, the standard control flow propagation semantics tend to over report relationships between values. Downstream analyses expect information flows to be indicative of strong, causal relationships between values. If a single, specific condition results in a particular value being assigned to a particular location, then there is a strong relationship between that condition and that assignment. However, if that same assignment can be triggered by many different conditions, then the relationship between those conditions and that assignment is weaker.

Prior work has considered refinements to the standard control flow propagation policy to address control-flow-related over-tainting. For instance, Bao et al. (2010) proposed a refinement to control flow tracking that only considered control flows resulting from strict equality checks rather than control flows resulting from all comparison operators. Kang et al. (2011) used symbolic execution to identify and propagate along control flow paths that can only be reached by a single input value. Approaches like these, which reduce over-tainting by considering only a subset of control flows, cannot fully address under-tainting without also causing over-tainting. That is, even if it were possible to determine and propagate along the optimal, minimal subset of control flows necessary to prevent under-tainting, over-tainting could still occur.

What constitutes over-tainting is ill-defined; the types of relationships that need to be tracked varies between applications. Generally, within the context of an application of taint tracking, if a label assigned to a piece of data conveys a relationship between that data and the source of the label that is not useful in that application, then that data is said to be over-tainted. For example, a privacy analysis expects data labeled as confidential to contain enough information from the original private source that if that data were leaked publicly, it would violate some expectation of secrecy or confidentiality. Although this is still ill-defined, it has likely motivated prior work on reducing control-flow-related over-tainting to consider the root cause of the over-tainting to be the amount of information that is transferred across a control flow. However, not all low-information flows result in over-tainting. Consider the data flow from `x` to `y` in the statement `y = x % 2`. This flow transfers very little information about the value of `x` to `y`. Half of all possible values for `x` map to zero and the other half to one; so it is clearly not a one-to-one mapping. Regardless, the value of `y` is what it is because of the value of `x`, and taint tracking tools are generally expected to report such a flow. Propagating along these sorts of low information data flows does not seem to cause the same over-tainting issues as propagating along control flows. It is our position that control-flow-related over-tainting stems from a mismatch between the nature of control and data flows.

In particular, taint tags propagated at runtime along data flows only contain information about what has actually happened during a particular execution. Code that has not executed does not impact data flows; they are determined by what has happened and not what could have happened. Dynamic taint tracking only provides insights into observed executions; unlike a static taint analysis it cannot prove things. This is often presented as a disadvantage of dynamic taint tracking over static taint tracking. However, many software engineering techniques rely upon this behavior. For example, Huo and Clause (2014) used dynamic tainting tracking to evaluate the quality of a test suite and was therefore were only interested in code that was actually executed by the test suite. Similarly, Attariyan and Flinn (2010)'s technique for identifying the root cause of configuration errors, used dynamic taint tracking because the technique sought to identify the cause of the specific failure that actually occurred.

In contrast to data flows, control flows contain information about execution paths that did not happen; they are impacted by code that did not execute. A control flow is produced by a conditional branch splitting the flow of control into two or more paths. Some statements execute on only some and not all of those paths. These statements are therefore considered to be dependent on the branch's outcome. Thus, control flows are inherently concerned with execution paths that were not taken.

Recent applications of dynamic tainting tracking that need or benefit from precise, fine-grained tainting tracking such as OraclePolish (Huo and Clause, 2014), VUzzer (Rawat et al., 2017), and ConfAid (Attariyan and Flinn, 2010) underscore the potential benefits of bridging the gap between the existing semantics for data and control flow tracking. To that end, this chapter makes the following contributions:

- Alternative control flow scope semantics for reducing the amount of over-tainting
- A heuristic that considers both dynamic and static information to reduce control-flow-related over-tainting
- A benchmark for evaluating a control flow propagation policies' ability to precisely capture control flows in real-world Java programs

3.2 BACKGROUND

Prior work on taint tracking, information flow control, slicing, and other related topics has used a variety of terms to describe the same or similar concepts to ones discussed in this chapter. Thus, for the sake of clarity, the terminology used in this chapter is defined below:

Data flow. A data flow (also known as an explicit flow) occurs due to an assignment, arithmetic, or logical operation that directly passes information from its operands to its result (Chandra and Franz, 2007; Sabelfeld and Myers, 2006; McCamant and Ernst, 2008; Clause et al., 2007; Enck et al., 2010a). In a data flow, the value of the flow's target (the operation's result) is derived from the value of flow's source (the operands of the operation).

Implicit flow. An implicit flow is the indirect passage of information between values typically as a result of conditional branches, array operations, or pointer dereferencing (McCamant and Ernst, 2008). In an implicit flow, the value of the flow's source is used to "select" the value of the flow's target. This definition of implicit flows is broader than the one used by Chandra and Franz (2007), Sabelfeld and Myers (2006), and Enck et al. (2010a) which includes only the indirect passage of information as a result of control structures.

Control flow. A control flow is an implicit flow resulting from a conditional branch (Clause et al., 2007; Enck et al., 2010a).

Dominance. Let $G = [V, E]$ be a control flow graph with designated entry and exit nodes denoted by $v_{entry} \in V$ and $v_{exit} \in V$. A node $v_i \in V$ dominates a node $v_j \in V$ if all paths in G from v_{entry} to v_j contain v_i.

Post-dominance. Let $G = [V, E]$ be a control flow graph with designated entry and exit nodes denoted by $v_{entry} \in V$ and $v_{exit} \in V$. A node $v_i \in V$ post-dominates a node $v_j \in V$ if all paths in G from v_j to v_{exit} contain v_i. Note that by this definition every node post-dominates itself and the designated exit node post-dominates every node.

Immediate post-dominance. Let $G = [V, E]$ be a control flow graph with designated entry and exit nodes denoted by $v_{entry} \in V$ and $v_{exit} \in V$. A node $v_i \in V$ is the immediate post-dominator of a node $v_j \in V$ if $v_i \neq v_j$; v_i post-dominates v_j; and there does not exist some $v_k \in V$ such that $v_k \neq v_i$, $v_k \neq v_j$, v_k post-dominates v_j, and v_k does not post-dominate v_i.

Scope of influence of a branch. The scope of influence of the execution of a conditional branching statement is the set of statements that execute after that execution of the conditional branching statement but before the next execution of the first statement in the immediate post-dominator of the basic block containing the branching statement. This definition is equivalent to the range of influence of a branch used by Weiser (1984) and Denning and Denning (1977).

Control flow scope. The scope of a control flow is the dynamic set of instruction executions during which any taint tags associated with the source of the flow propagate to written values.

Standard control flow scope. We use the term standard control flow scope to refer to the typical definition for the scope of a control flow which is defined with respect to the post-dominance relation. In particular, the standard scope of a control flow introduced by some conditional branch is defined as the set of instructions that execute after the flow of control splits at the branch but before the flow of control rejoins at the immediate post-dominator of the basic block containing that branch (Denning and Denning, 1977).

Over-tainting. Over-tainting is when a taint tag assigned to a value falsely conveys a relationship between that value and the source of the taint tag. While conceptually, over-tainting can be caused by an imprecision in the underlying program analysis, this chapter focuses on over-tainting caused by a mismatch between a taint tracking system's propagation rules and the expectations of analyses built on top of that taint tracking system. This type of over-tainting behavior was described by Clause and Orso (2009), Staicu et al. (2019), and Attariyan and Flinn (2010).

Under-tainting. Under-tainting is when a value has not been assigned a particular taint tag falsely conveying a lack of relationship between the value and the source of the taint tag.

Propagation policy. A tainting tracking system's propagation policy is the set of rules defining how taint tags should propagate when an operation executes.

The typical approach to control flow tracking considers there to be a control flow from the predicate of a conditional branch to any values written within the control flow's "scope". Many dynamic taint analysis systems track these scopes using a stack (Bell and Kaiser, 2014, 2015; Clause et al., 2007; Chandra and Franz, 2007). The taint tag of a branch's predicate is pushed onto this stack at the start of a control flow's scope and popped at the end of its scope. Traditionally, this scope is defined with respect to the post-dominance relation. Specifically, the standard definition used for the scope of a control flow introduced by some conditional branch is defined as the set of instructions that execute after the flow of control splits at that branch but before the flow of control rejoins at the immediate post-dominator of the basic block containing that branch (Denning and Denning, 1977).

Bao et al. (2010) and Kang et al. (2011) propose propagating taint tags along a subset of control flows based on the "syntax of [the] comparison expression" using the standard, post-dominator-based definition for control flows' scopes. Additionally, Kang et al. (2011) attempt to identify "culprit" flows, control flows along which taint tags need be propagated in order to avoid under-tainting. However, even if propagation occurs only along the optimal, minimal subset of control flows necessary to prevent under-tainting (i.e., culprit flows), the standard control flow scope definition can cause over-tainting. Consider the code in Figure 2a. If taint tags are not propagated along control flows, under-tainting can occur because the relationship between a plus sign in the input and a space in the output is missed. The minimal subset of control flows needed to correct this under-tainting contains only the flow introduced by the branch from the `switch` statement's case on line 8. Figure 2d shows the labels expected to propagate to the output produced by `spaceDecode` when presented with an input array with each of its characters tainted with its position in the array (as shown in Figure 2b). However, the control flow graph for `spaceDecode` (depicted in Figure 2c) shows that the immediate post-dominator of the basic block that contains `switch(input[i])` is the exit node. Thus, once the branch associated with the case on line 8 is traversed, the label for the predicate of that branch will be pushed onto the taint stack and impact all subsequent instructions until the method is exited. This causes the output of `spaceDecode` to be over-tainted as described in Figure 2d.

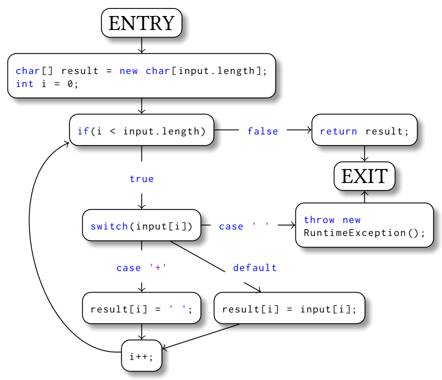
This over-tainting can be fixed by reducing the scope of the control flow to include only the basic block that contains the instructions on lines 9 and 10. However, even if control flow propagation occurs only along the minimal subset of control flows necessary to prevent under-tainting with the minimal, basic-block level scopes necessary to prevent under-tainting for each control flow, over-tainting can still occur. Consider the code in Figure 3a. If taint tags are not propagated along control flows, under-tainting can occur because the relationship between a percent sign in the input and a decoded character in the output would be missed.

```java
static char[] spaceDecode(char[] input) {
  char[] result = new char[input.length];
  for(int i = 0; i < input.length; i++) {
    switch(input[i]) {
      case ' ':
        throw new RuntimeException();
      case '+':
        result[i] = ' ';
        break;
      default:
        result[i] = input[i];
    }
  }
  return result;
}
```

(a) A simple Java method for decoding spaces encoded as plus signs.

Input Value	H	e	l	l	o	+	W	o	r	l	d
Applied Tags	0	1	2	3	4	5	6	7	8	9	10

(b) Sample tainted input for the `spaceDecode` method (Figure 2a). Each input character is tainted with its position in the input array (e.g., the first input character, "H", is tainted with the tag "0").



(c) Control flow graph for the `spaceDecode` method (Figure 2a).

Output Value	H	e	l	l	o		W	o	r	l	d
Expected Tags	0	1	2	3	4	5	6	7	8	9	10
Propagated Tags	0	1	2	3	4	5	5, 6	5, 7	5, 8	5, 9	5, 10

(d) Expected tainted output from the `spaceDecode` method (Figure 2a) when presented with the sample input from Figure 2b compared with the actual tainted output when propagating along the minimal subset of control flows necessary to prevent under-tainting using the standard scope definition.

Figure 2: An example of a program in which using the standard control flow propagation semantics results in over-tainting. The method `spaceDecode` in Figure 2a takes a sequence of characters that are not spaces. When a plus sign is encountered, a space is added to the output. Every other input character is copied to the output. When `spaceDecode` is executed with the tainted input displayed in Figure 2b, the taint tag of every input plus sign ("+") is expected to flow to the output with the produced space character. Additionally, the taint tag of every other input character is expected to flow to the output with the character. However, as shown in Figure 2d, the standard control flow propagation semantics over-taint the output.

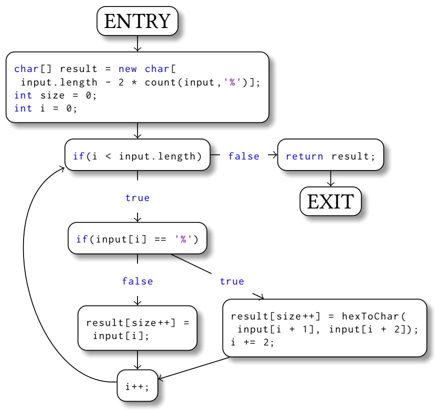
The minimal subset of control flows needed to correct this under-tainting contains only the branch on line 5. As shown in the control flow graph for `percentDecode` depicted in Figure 3c, the minimal scope for that control flow includes only the basic block that contains the instructions on lines 6 and 7. Figure 3d shows the labels expected to propagate to the output produced by `percentDecode` when presented with an input array with each of its characters tainted with its position in the array (as shown in Figure 3b). When a percent sign is encountered in the input and the branch on line 5 evaluates to true, the label for that input is pushed onto the taint stack. That label then correctly propagates to the element of the `result` array that is assigned a value on line 6. But, it also incorrectly propagates to the variable `size` and the looping variable `i` when their values are incremented on lines 6 and 7. The end of the control flow's scope is then hit and its label is popped from the taint stack. On subsequent iterations of the loop, when `i` is used to access elements of the input array, `i`'s taint tag propagates to the accessed element. Additionally, when `size` is used to select where in the output array to store a value, `size`'s taint tag propagates to the stored value. This causes the output of `percentDecode` to be over-tainted as described in Figure 3d.

```
1  static char[] percentDecode(char[] input) {
2    char[] result = new char[input.length - 2 * count
        (input, '%')];
3    int size = 0;
4    for(int i = 0; i < input.length; i++) {
5      if(input[i] == '%') {
6        result[size++] = hexToChar(input[i + 1],
        input[i + 2]);
7        i += 2;
8      } else {
9        result[size++] = input[i];
10     }
11   }
12   return result;
13 }
```

(a) A simple Java method for decoding percent-encoded values.

Input Value	%	4	8	%	6	9	%	2	1
Applied Label	0	1	2	3	4	5	6	7	8

(b) Sample tainted input for the `percentDecode` method (Figure 3a). Each input character is tainted with its position in the input array (e.g., the first input character, "%", is tainted with the tag "0").



```
char[] result = new char[
 input.length - 2 * count(input,'%')];
int size = 0;
int i = 0;

if(i < input.length)

return result;

if(input[i] == '%')

result[size++] =
 input[i];

result[size++] = hexToChar(
 input[i + 1], input[i + 2]);
i += 2;

i++;
```

(c) Control flow graph for the `percentDecode` method (Figure 3a).

Output Value	H	i	!
Expected Labels	{0, 1, 2}	{3, 4, 5}	{6, 7, 8}
Propagated Labels	{0, 1, 2}	{0, 3, 4, 5}	{0, 3, 6, 7, 8}

(d) Expected tainted output from the `percentDecode` method (Figure 3a) when presented with the sample input from Figure 3b compared with the actual tainted output when propagating along the minimal subset of control flows with the minimal, basic-block level scopes necessary to prevent under-tainting.

Figure 3: An example of a program in which using the standard control flow propagation semantics results in over-tainting. The method `percentDecode` in Figure 3a takes a sequence of characters and percent-encoded octets. Each input character that is not part of a percent-encoded octet is copied to the output. Each input percent-encoded octet is decoded into a character and that character is copied to the output. When `percentDecode` is executed with the tainted input displayed in Figure 3b, the taint tag of an input character that is not part of a percent-encoded octet is expected to flow to the output with the character. Additionally, the union of the taint tags of an input percent-encoded octet is expected to flow to the output with the character decoded from the octet. However, as shown in Figure 3d, the standard control flow propagation semantics over-taint the output.

3.3 APPROACH

Our approach, CONFLUX, aims to precisely propagate taint tags through information-preserving transformations in programs. CONFLUX leverages novel heuristics to identify control flows that are likely to correspond to information-preserving transformations and ignore taint tag propagation from others. Like Bao et al. (2010)'s approach, CONFLUX considers only a subset of control flows for propagation based on the comparative operator of the control flow's branch. Specifically, CONFLUX includes only control flows introduced by equality checks. However, even when considering only control flows introduced by equality checks, significant over-tainting can still occur. As discussed in Section 3.2, the standard, post-dominator based definition for the scope of a control flow is overly conservative. Thus, CONFLUX does not use the standard scope definition and instead introduces the notion of a "binding" scope which includes a subset of the basic blocks contained in the control flows' standard scope. However, this alone is not sufficient to produce the expected tainted output in Figure 3a, since the loop index `i` will still become tainted with the taint tag of `input[i]` on line 5. To address this and similar over-tainting, CONFLUX introduces a dynamic heuristic, "loop-relative stability", which reasons about the strength of the relationship between a conditional branch and an assignment statement by considering the impact of executing loops on program semantics.

3.3.1 Binding Scope

Overall, Conflux aims to identify conditional branch executions that are not strictly information preserving and avoid propagating along the resulting control flows. To achieve this, Conflux uses an alternative control flow scope definition that distinguishes between statements that can execute only as a result of a single, specific condition and statements that can execute under multiple conditions. The traditional definition for control flows' scopes is highly conservative; it considers every basic block between a branch in the flow of control and where that branch rejoins to be within the scope of the branch. Unlike traditional control flow scopes, which are defined with respect to nodes in the control flow graph, "binding" scopes are defined with respect to edges. In particular, binding scopes are defined with respect to the edges which are traversed when a conditional branching statement is "taken" or "not taken" (or for `switch` statements the edges associated with the cases of the `switch`). The binding scope of a branch edge includes instructions which only execute if the edge is traversed, i.e., an instruction is included if every path from the distinguished entry node of the control flow graph to that instruction contains the branch edge. Since Conflux only considers branch edges corresponding to equality checks (e.g., the branch taken side of an equality check or the branch not taken side of an inequality check), the execution of an instruction within the scope of a branch edge occurs only if some value was equal or "bound" to another value.

Binding scopes can be calculated for the branch edges in a method by constructing its control flow graph, $G = [V, E]$, with designated entry and exit nodes denoted by $v_{entry} \in V$ and $v_{exit} \in V$ respectively. Each node in V other than v_{entry} and v_{exit} represents a basic block and consists of a sequence of instructions. The successors of a node $u \in V$ is defined as $succ(u) = \{v \mid (u, v) \in E\}$. A node $v \in V$ is said to be a branch node if it has more than one successor. The last instruction of a branch node is its conditional branch instruction. We refer to the set of outgoing edges of a branch node as its branch edges. Each branch edge is associated with a set of conditions under which the edge is traversed. For example, an `if` statement will have two branch edges: one edge corresponding to the branch being taken with a singleton condition set corresponding to the statement's predicate evaluating to true and one edge corresponding to the branch not being taken with a singleton condition set corresponding to the statement's predicate evaluating to false. Whereas, a `switch` statement's branch edges' condition sets will partition the cases of the `switch` statement.

An instruction i is within the binding scope of a branch edge e if i can only execute if e has been traversed. By definition (Aho et al., 2006), all of the instructions within a basic block execute if and only if the first instruction in the basic block executes. Therefore, we can define binding scopes with respect to basic blocks instead of individual instructions. Thus, the binding scope of a branch edge e is the set of all basic blocks v such that all paths in G from v_{entry} to v contain e.

To calculate the binding scope of branch edges we first construct a modified control flow graph, G', from G by replacing each branch edge $(u, v) \in E$ with a new node $b_{u,v}$ and a pair of edges $(u, b_{u,v})$ and $(b_{u,v}, v)$. Given this construction, the binding scope of a branch edge e is the set of all $v \in V$ such that b_e dominates v in G'. Proof of the correctness of this calculation is as follows:

Proposition 1. *Given a branch edge $e \in E$ and b_e, its node replacement in G', b_e dominates a node $v \in G'$ if and only if all paths in G from v_{entry} to v contain e.*

Proof. Suppose that b_e dominates v in G'. Assume that there exists a path, $P = [v_{entry}, v_0, \ldots, v_n, v]$, in G that does not contain e. Since b_e dominates v in G', every path in G' from v_{entry} to v must go through b_e. Given that $b_e \notin V$ and therefore $b_e \notin P$, P must contain at least one edge, (v_i, v_{i+1}), that is not in E' such that all paths from v_i to v_{i+1} in G' contain b_e. Furthermore, $(v_i, v_{i+1}) \in E$ and $(v_i, v_{i+1}) \notin E'$ implies that (v_i, v_{i+1}) was a branch edge that was replaced during the construction of G'. Thus, $b_{v_i,v_{i+1}} \in V'$, $(v_i, b_{v_i,v_{i+1}}) \in E'$, and $(b_{v_i,v_{i+1}}, v_{i+1}) \in E'$. As a result, there is a path $[v_i, b_{v_i,v_{i+1}}, v_{i+1}]$ in G'. This path must contain b_e, therefore $b_{v_i,v_{i+1}} = b_e$. Therefore, given the construction process of G', $(v_i, v_{i+1}) = e$. This contradicts the assumption P does not contain e. Thus, b_e dominates v in G' implies that all paths in G from v_{entry} to v contain e.

Now suppose that all paths in G from v_{entry} to v contain a branch edge e. Assume that b_e does not dominate v in G' because there exists some path, $P = [v_{entry}, v_0, \ldots, v_n, v]$ in G' that does not contain b_e. Given a sequential pair of nodes v_i and v_{i+1}, in P, the edge (v_i, v_{i+1}) is either present in E or it was added when a branch edge in E was replaced. If $(v_i, v_{i+1}) \notin E$, then either $v_i \notin V$ or $v_{i+1} \notin V$ This is a consequence of the construction procedure for G', since each edge added to G' is between an element of the original set of nodes and a node not in the original set of nodes. Furthermore, as a result of this, every edge in G' uses at least one node that is an element of V. Since P begins at a node in V, if $v_i \notin V$, there must be a node, $v_{i-1} \in V$ immediately before v_i in P. Since P ends at a node in V, if $v_{i+1} \notin V$, there must be a node, $v_{i+2} \in V$ immediately after v_{i+1} in P. Thus, P can be broken into a sequence of sub-paths either of the form $[v_i, v_{i+1}]$ where $(v_i, v_{i+1}) \in E$ or the form $[v_i, v_{i+1}, v_{i+2}]$ where $v_i \in V$, $v_{i+1} \notin V$, and $v_{i+2} \in V$. For each such sub-path, there is an edge in E that does not equal e that connects the start of the sub-path directly to the end. Proof is as follows for each of two cases:

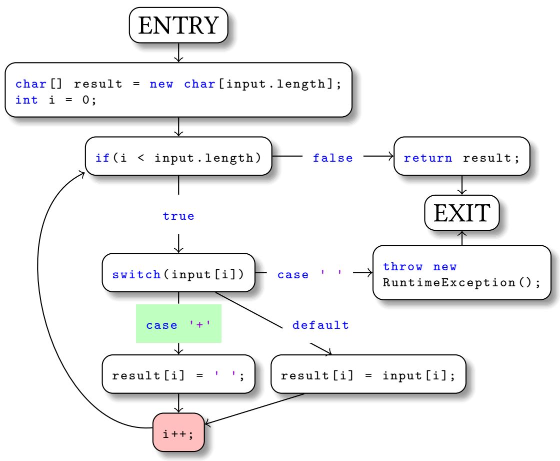
Case 1. $[v_i, v_{i+1}]$ *where* $(v_i, v_{i+1}) \in E$
Since $e \notin E'$, P *contains* (v_i, v_{i+1}), *and* P *is a path in* G', $(v_i, v_{i+1}) \neq e$.

Case 2. $[v_i, v_{i+1}, v_{i+2}]$ *where* $v_i \in V$, $v_{i+1} \notin V$, *and* $v_{i+2} \in V$
The edges (v_i, v_{i+1}) *and* (v_{i+1}, v_{i+2}) *must have been added to* E' *when the node* v_{i+1} *was added to* V' *as a replacement for the edge* (v_i, v_{i+2}). *Since* b_e *is not in* P *and* v_{i+1} *is in* P, $v_{i+1} \neq b_e$. *Therefore,* $(v_i, v_{i+2}) \in E$ *and* $(v_i, v_{i+2}) \neq e$.

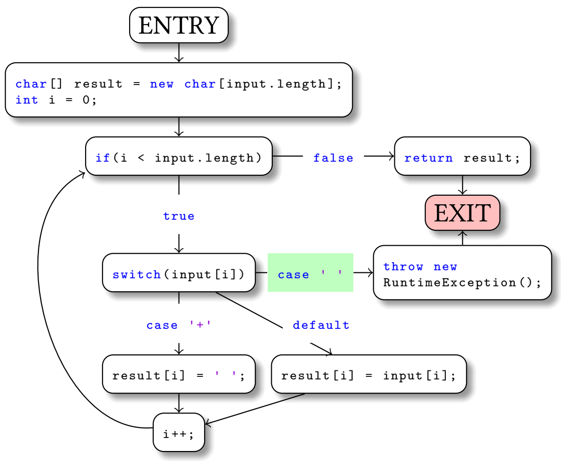
These edges form a path in G from v_{entry} to v not containing e contradicting the assumption that all paths in G from v_{entry} to v contain the branch edge e. Thus, if all paths in G from v_{entry} to v contain a branch edge e, b_e dominates v in G'. □

The binding scope of a branch edge is "scope-like"; all nodes within the scope lie on paths between the edge and its dominance frontier. Thus, the taint tag of a branch's predicate need only be pushed onto a taint stack once at the start of the branch edge's scope and can be safely popped at the ends of its scope. Another desirable property of binding scopes is that the set of basic blocks within a branch edge's binding scope is a subset of the basic blocks within its branch's standard (post-dominator based) scope. The immediate post-dominator of a branch node is reachable from all of its branch edges and therefore either part of or beyond the dominance frontier of its node replacement in the modified graph.

We can now apply the binding scope definition to the `spaceDecode` method in Figure 2a using its control flow graph shown in Figure 2c. There are two branch edges corresponding to equality checks: the one associated with `case '+'` and the one associated with `case ' '`. The `case '+'`'s edge's binding scope contains only the basic block with the instruction `result[i] = ' '`. The `case ' '`'s edge's binding scope contains only the basic block with the instruction `throw new RuntimeException()`. These scopes are depicted in Figure 4. In this case, using binding scopes allows the relationship between a plus sign in the input and a space in the output to be reflected without introducing over-tainting.



(a) Binding scope for the `case '+'` edge.



(b) Binding scope for the `case ' '` edge.

Figure 4: Binding scopes for the two branch edges corresponding to equality checks in the `spaceDecode` method shown in Figure 2a. Each scope is shown on the control flow graph for `spaceDecode`. The start of each scope (exclusive) is colored green. The end of each scope (exclusive) is colored red.

3.3.2 Loop-Relative Stability Heuristic

When propagating along control flows, taint tags often accumulate on program data during the execution of a loop leading to an "explosion" of taint tags. For example, regardless of whether taint tags are applied in standard control flow scopes or only in binding scopes, during the execution of the loop in the `percentDecode` method (Figure 3a), taint tags build up on the looping variable, `i`, resulting in progressively larger label sets for each successive output. CONFLUX mitigates this accumulation of taint tags by making special considerations when determining whether to propagate taint tags between the branch of a control flow and a statement within the control flow's scope. This process is guided by the novel "loop-relative stability" heuristic.

The underlying idea for loop-relative stability is that loops introduce alternative paths to statements within the scope of a control flow. For instance, within a single call to the `percentDecode` method (Figure 3a), the instruction on line 7, `i += 2`, can execute even if the branch on line 5 is not taken on a particular iteration of the loop. That is, on subsequent iterations of the loop, the branch on line 5 could evaluate to true, causing the statement `i += 2` to execute, thereby producing the same effect that would have happened had the branch been taken on the earlier iteration. This weakens the relationship between the value of `i` and the element of the `input` array that caused the branch on line 5 to be taken. However, when a value is stored to `result[size++]` on line 6, the storage location to which that value is stored is different on each iteration of loop. Like before, on subsequent iterations of the loop, the branch on line 5 could evaluate to true causing the statement on line 6 to execute. However, unlike the statement on line 7 which updates the same local variable `i` on different iterations, the statement on line 6 updates a *different* element in the `result` array on different iterations. This produces a stronger relationship between the value of `input[i]` on line 5 and the value written on line 6 to `result[size++]` than the relationship between the value of `input[i]` on line 5 and the value written on line 7 to `i`.

The loop-relative stability heuristic aims to identify these cases in which a loop introduces multiple conditions under which the same location could be assigned the same value. In order for this to occur, there must be a conditional branching statement that is contained within a loop and an assignment statement within the scope of that branch's control flow. If the values used by the branch change on different iterations of the loop but the values used by the assignment statement stay the same, then on each iteration of the loop a different condition could cause the same effect. Since the conditional branching statement might occur in a different method than the loop and assignment statement, the loop-relative stability heuristic is defined in terms of the dynamic execution of statements. To capture these ideas, we define an execution of a program statement as being "stable" relative to an executing loop if the values used by that statement are the same on every iteration of that loop. The loop-relative stability heuristic determines whether to propagate along a particular control flow based on the stabilities of statement executions. More specifically, let b be an execution of conditional branching statement and a be an execution of an assignment statement that is within the scope of b's control flow. The loop-relative stability heuristic propagates along the control flow from b to a only if b is stable relative to every loop to which a is relatively stable.

The loop-relative stability heuristic considers only "natural" loops as defined by (Aho et al., 2006). In particular, a natural loop is a single-point-of-entry cycle in the control flow graph defined with respect to a back edge, i.e., an edge whose target dominates its source. The natural loop of some back edge (u, v) consists of all nodes x such that v dominates x and there exists a path from x to u not containing v. The node v is said to be the header of the natural loop defined by the back edge (u, v). Any two natural loops with the same header are combined and treated as a single loop. This definition of loops ensures that any two loops are either disjoint or nested, i.e., one loop is fully contained within the other. The use of arbitrary `GOTO` statements may in some cases produce control flow graphs that contain cycles that do not correspond to natural loops. However, most structured programming languages do not allow programmers to produce control flow graphs that contain cycles that do not correspond to natural loops. Thus, we feel that it is appropriate for the loop-relative stability heuristic to only consider natural loops.

Instability Levels

Conceptually, the loop-relative stability heuristic could be expressed in terms of "stability sets"; a stability set is the set of loops with respect to which a statement execution is relatively stable. Let S_b be the stability set of an execution of conditional branching statement b and S_a be the stability set of an execution of an assignment statement within the scope of b's control flow. The loop-relative stability heuristic propagates along the control flow from b to a only if S_b is a superset of S_a. However, instead of tracking these stability sets, it is possible to express the same concept using a single number, an "instability level". An instability level is a numeric value between zero and the number of loops containing the statement currently executing. This number represents the "depth" of the innermost loop relative to which a statement is not stable. The loop-relative stability heuristic propagates along the control flow from an execution of a conditional branching statement b to an execution of an assignment statement a within the scope of b's control flow only if b's instability level is less than or equal to a's instability level. An explanation of why this simplification is possible is provided below.

Given a statement execution contained within two nested loops, if the values used by the statement change over the duration of the inner loop then they must also change over the duration of the outer loop, since the inner loop is fully contained within the outer loop. Thus, the stability set of a statement execution is defined by the innermost loop relative to which it is not stable. If an execution of a program statement occurs outside a loop, that execution must be stable relative to that loop, since it is not possible for the values used by the statement to change over the duration of the loop. As a result, when calculating the loop-relative stability for the statement currently executing, only the set of loops containing that statement need to be considered in the calculation.

However, this does not by itself guarantee that all comparisons for the loop-relative stability heuristic made at runtime only need to consider the set of loops containing the statement currently executing. The stability set of the execution of a conditional branching statement needs to be used before the execution of any assignment statement within the scope of the branch's control flow. This means that the stability set of an execution of a conditional branching statement may be considered by the loop-relative stability heuristic after the innermost loop relative to which the execution was not stable was exited. However, once the innermost loop relative to which an execution of a conditional branching statement is not stable is exited all subsequent statements will execute outside of that loop and therefore be stable relative to it. Since these executions are stable relative to a loop that the branch execution was not, the loop-relative stability heuristic will prevent the branch's predicate from propagating to these statements as a result of the control flow. Thus, Conflux stops all propagation along the control flow from the execution of a conditional branching statement as soon as the innermost loop relative to which that branch execution is not stable is exited. As a result, all comparisons for the loop-relative stability heuristic made at runtime only need to consider the set of loops containing the statement currently executing. Given this and the fact that the stability set of a statement execution can be defined by the innermost loop relative to which it is not stable, loop-relative stabilities can be specified at runtime as a single number, an instability level, between zero and the number of loops containing the statement currently executing.

These instability levels are calculated at runtime as a function of both static information, the stability "classifier" of a statement (Section 3.3.2), and dynamic information, the context of the statement's execution (Section 3.3.2). A purely dynamic approach could only consider instructions that have already executed. This is problematic because it's possible for the predicate of a conditional branching statement to be the same on all but the last iteration of a loop. By the time that last iteration occurs the loop-relative stability heuristic may have already been applied causing Conflux to incorrectly propagate along a control flow from that branch. Furthermore, the loop-relative stability is concerned with the presence of alternative conditions that could have produced the same outcome. It does not matter whether those conditions were met on a particular execution. For example, consider the `any` method on line 3 of Listing 3.2. The for-loop on lines 4–8 introduces multiple conditions under which the value of `x` is set to `true`, specifically if any of the elements of the array `z` is `true`. If `any` is passed an array `new boolean[]{false, false, false, true}`, then there happened to be one condition, `z[3] == true` that was satisfied and caused `x` to be assigned the value `true`. However, there were still possible alternative conditions under which that assignment could have occurred. Therefore, propagation should not occur from the branch on line 5 to the assignment statement on line 6 according to the loop-relative stability heuristic. To handle this, Conflux uses static information to reason about possible executions and assign stability classifiers to program elements.

Conflux also relies upon dynamic information about calling contexts. Consider the `any_` method on line 15 of Listing 3.2. The `any_` method is functionally equivalent to the `any` method on line 3 of Listing 3.2, but this functionality is split across two methods, `any_` and `setX`. Because `any_` is functionally equivalent to `any`, taint tag propagation for the two methods should ideally be the same, meaning that propagation should not occur during the execution of the assignment statement on line 12 of the `setX` method. However, `setX` is also called by the method `checkFirst` on line 25 of Listing 3.2. The conditional branching statement on line 24 of `checkFirst` does not necessarily occur within a loop. Therefore, propagation may need to occur during the execution of the assignment statement on line 12 of the `setX` method when called from the `checkFirst` method. Thus, it is not possible to determine whether propagation should occur during the execution of the assignment statement on line 12 without considering the dynamic execution context of the call to `setX`. Thus, Conflux constructs and passes between methods information about execution contexts at runtime. These execution contexts are used to calculate instability levels.

```java
static boolean x = false;

static void any(boolean[] z) {
    for (int i = 0; i < z.length; i++) {
        if (z[i]) {
            x = true;
        }
    }
}

static void setX(boolean value) {
    x = value;
}

```

```java
static void any_(boolean[] z) {
    for (int i = 0; i < z.length; i++) {
        if (z[i]) {
            setX(true);
        }
    }
}

static void checkFirst(boolean[] z) {
    if (z[0]) {
        setX(true);
    }
}
```

Listing 3.2: Java methods to demonstrate why both static and dynamic information is used to calculate loop-relative stabilities.

Stability Classifiers

Conflux statically assigns stability classifiers to program elements; these classifiers encode information about possible program executions. There are three types of stability classifiers: stable, dependent, and unstable. The stable classifier type indicates that all executions of a program element are stable relative to all loops regardless of the dynamic execution context. We use Stable to denote a stable-type classifier. The dependent classifier type indicates that the instability level of an execution of a program element depends on the instability level of one or more arguments passed to the method call that contains the execution. We refer to the set of parameters corresponding to these arguments as the dependency set of the classifier. We use Dependent$\langle d \rangle$ to denote a dependent-type classifier with a dependency set d. In addition to dependencies on arguments, a dependent-type classifier can also indicate a dependency on the execution context's return value location. We use α to denote a dependency on the execution context's return value location.

The unstable classifier type indicates that the instability level of an execution of a program element depends on the loops that contain the method call that contains the execution. We use Unstable$\langle l \rangle$ to denote an unstable-type classifier for a program element that is not stable relative to the elements of l, a set of loops within the method.

At runtime, Conflux determines instability levels based on stability classifiers and execution contexts. The instability level of a stable-type classifier is always zero. The instability level of a dependent-type classifier is dynamically calculated as the maximum instability level of the arguments corresponding to the parameters in the program element's dependency set. The instability level of an unstable-type classifier, Unstable$\langle l \rangle$, is dynamically calculated as the number of loops containing the method call plus $|l|$. We describe this more precisely in Section 3.3.2 below.

To calculate stability classifiers for a method, Conflux first converts the method into an intermediate representation in static single assignment (SSA) form. By converting the method into SSA form, Conflux can easily identify reaching definitions since there will be exactly one definition reaching each use. Next, Conflux creates a control flow graph for the method and uses this graph to identify which natural loops within the method contain each instruction. We use loops(i) to denote the set of natural loops containing an instruction i. Finally, Conflux calculates the stability classifier of each conditional branching statement, assignment statement, non-void return statement, method call receiver, method call argument, and method call return value location. The stability classifiers of conditional branching statements, assignment statements, and non-void return statements (which Conflux treats like interprocedural assignment statements) are directly used to determine whether to propagate along a control flow in accordance with the loop-relative stability heuristic. In order to construct an execution context for a method call, Conflux uses the stability classifier of the method call's receiver, arguments, and return value location. We discuss this in detail in Section 3.3.2.

For the sake of computing stability classifiers, we define a function merge(c_1, c_2) in Algorithm 1 for combining two classifiers, c_1 and c_2 to produce a classifier c_3 such that for any possible execution context the instability level of c_3 will be greater than or equal to the instability level for c_1 and c_2. Using this merge function, Conflux can then calculate stability classifiers for value expressions (program entities that are evaluated to produce a value) and storage locations (program entities that represent a place for storing a value, i.e., the left-hand-side of an assignment statement) as described in Algorithms 2 and 3, respectively.

Algorithm 1 Function for combining two stability classifiers c_1 and c_2.

Input: stability classifier c_1, stability classifier c_2
Output: a stability classifier

```
 1: function MERGE($c_1, c_2$)
 2:     if $c_1$ = STABLE then
 3:         return $c_2$
 4:     else if $c_2$ = STABLE then
 5:         return $c_1$
 6:     else if $c_1$ = DEPENDENT$\langle d_1 \rangle$ and $c_2$ = DEPENDENT$\langle d_2 \rangle$ then
 7:         return DEPENDENT$\langle d_1 \cup d_2 \rangle$
 8:     else if $c_1$ = UNSTABLE$\langle l_1 \rangle$ and $c_2$ = UNSTABLE$\langle l_2 \rangle$ then
 9:         return UNSTABLE$\langle l_1 \cup l_2 \rangle$
10:     else if $c_1$ = UNSTABLE$\langle l_1 \rangle$ then
11:         return $c_1$
12:     else
13:         return $c_2$
14:     end if
15: end function
```

Conflux directly uses the `valueClassifier` function (Algorithm 2) to calculate stability classifiers for method call receivers and arguments since these program elements are value expressions. The `valueClassifier` function (Algorithm 2) is also used to calculate the stability classifier of each conditional branching statement by applying the function to the predicate expression of the branching statement. To calculate the stability classifier of a non-void return statement, Conflux first calculates the stability classifier of the expression returned by the statement and then combines this classifier with a dependent-type classifier with a single dependency on the execution context's return value location, More formally, the stability classifier of a non-void return statement is merge(Dependent$\langle\{\alpha\}\rangle, c_v$), where c_v is the stability classifier of the expression returned by the statement. The stability classifier of a method call return value location is determined by applying the `locationClassifier` (Algorithm 3) function to the storage location assigned the value of the result of the method call. If the return value of a method call is unused, then the stability classifier of the return value location for that method call is Stable.

Conflux uses both the `valueClassifier` and `locationClassifier` functions to calculate the stability classifier of an assignment statement. However, Conflux makes a special consideration when calculating

Algorithm 2 Function for calculating the stability classifier of a value expression *e* that appears in an instruction *i*.

```
Input: value expression $e$, instruction $i$
Output: a stability classifier
function valueClassifier($e, i$)
    $f \leftarrow$ loops($i$)
    if $e$ is a constant or literal then
        return Stable
    else if $e$ is a storage allocation (e.g., an array definition) then
        return Unstable⟨f⟩
    else if $e$ loads a value from an array or field then
        return Unstable⟨f⟩
    else if $e$ is a method invocation then
        return Unstable⟨f⟩
    else if $e$ is a local variable $x$ then
        $v \leftarrow$ the value expression assigned to $x$ in the reaching definition of $x$
        $c_v \leftarrow$ valueClassifier($v, j$)
        if $c_v$ = Unstable$\langle g \rangle$ then
            return Unstable$\langle g \cap f \rangle$
        else
            return $c_v$
        end if
    else if $e$ is of the form $\diamond_u a$ (where $\diamond_u$ is a unary operator) then
        return valueClassifier($a, i$)
    else if $e$ is of the form $a \diamond_b b$ (where $\diamond_b$ is a binary operator) then
        $c_a \leftarrow$ valueClassifier($a, i$)
        $c_b \leftarrow$ valueClassifier($b, i$)
        return merge($c_a, c_b$)
    else if $e$ is a parameter or method receiver $x$ then
        return Dependent⟨{x}⟩
    else if $e$ is a caught exception $x$ then
        return Unstable$\langle f \rangle$
    else if $e$ is a $\Phi$ function then
        return Unstable$\langle f \rangle$
    end if
end function
```

the stability classifiers of assignment statements in order to address situations in which the value to be stored by an assignment was produced from an expression containing a term matching the current value at the destination storage location of the assignment statement. We refer to this as an “update” assignment and exclude any update terms, terms that match the current value at the storage location, from the stability calculation for the value. Thus, before Conflux can determine the stability classifier of an assignment statement, it must identify any portions of the right-hand-side of the assignment statement that correspond to excluded update terms. If the destination storage location of the assignment statement is a local variable, any uses of that definition of that local variable that reach the assignment statement are considered to be

Algorithm 3 Function for calculating the stability classifier of a storage location x that appears in an instruction i.

```
Input: storage location $x$, instruction $i$
Output: a stability classifier
 1: function locationClassifier($x, i$)
 2:     $f \leftarrow$ loops($i$)
 3:     if $x$ is a local variable then
 4:         if $x$ is directly used in an invoke expression on the right-hand side of an assignment statement then
 5:             return Unstable⟨f⟩
 6:         end if
 7:         $c \leftarrow$ Stable
 8:         $A \leftarrow$ the set of assignment statements that directly use the value at $x$
 9:         for $a \in A$ do
10:             $y \leftarrow$ the left-hand-side of $a$ (i.e., destination storage location)
11:             if $y$ is not a local variable then
12:                 $c_y \leftarrow$ locationClassifier($y$)
13:                 $c \leftarrow$ merge($c, c_y$)
14:             end if
15:         end for
16:         if the value at $x$ is directly used in a return statement then
17:             $c \leftarrow$ merge($c$, Dependent⟨$\alpha$⟩)
18:         end if
19:         return $c$
20:     else if $x$ is an instance field of the form $a$.field then
21:         return valueClassifier($a, i$)
22:     else if $x$ is a class field or global variable then
23:         return Stable
24:     else if $x$ is an array element of the form $a[b]$ then
25:         $c_a \leftarrow$ valueClassifier($a, i$)
26:         $c_b \leftarrow$ valueClassifier($b, i$)
27:         return merge($c_a, c_b$)
28:     end if
29:     return Stable
30: end function
```

update terms. If the destination storage location is an array, Conflux considers any values that are loaded from the same array at the same position to be candidate update terms. These values may represent update terms, but it is also possible that the value at that storage location was written between the instruction that loaded the candidate update term and the assignment statement. Conflux ignores the potential for concurrent writes to that storage location and accepts a candidate as being a true update term if there does not exist an execution path containing an array store or a method call between the instruction that loaded the candidate and the assignment statement. In practice, checking all possible paths between the two instructions may be impractical. Thus, for small control flow graphs (those containing less than ten basic blocks), Conflux checks all possible paths between the two instructions. For larger control flow graphs, Conflux assumes that such a path exists if the two instructions are not in the same basic block. If the destination storage location of the assignment statement is a field, Conflux considers any values loaded from the same field and, for instance fields, from the same instance to be candidate update terms. Once again, Conflux checks all paths between the instruction that loads the candidate update term and the assignment statement. However, in this case, the paths are checked for method calls and field stores.

Once Conflux has identified any portions of the right-hand-side of the assignment statement that are considered to be excluded update terms, it calculates the stability classifier of the assignment statement based on the remaining portions of the right-hand-side of the assignment statement and the left-hand-side of the assignment statement (i.e., its destination storage location). Algorithm 4 shows how Conflux performs this calculation.

Algorithm 4 Function for calculating the stability classifier of an assignment statement that appears in an instruction i and assigns a value expression v to a storage location x.

```
Input: storage location $x$, value expression $v$, instruction $i$
Output: a stability classifier
1: function assignmentClassifier($x, v, i$)
2:     $W \leftarrow$ set containing the portions of $v$ that are not excluded update terms
3:     $c \leftarrow$ locationClassifier($x, i$)
4:     for $w \in W$ do
5:         $c_w \leftarrow$ valueClassifier($w, i$)
6:         $c \leftarrow$ merge($c, c_w$)
7:     end for
8:     return $c$
9: end function
```

Execution Contexts

The loop-relative stability of a statement can vary between executions depending upon the dynamic execution context in which the call to the method that contains the statement was made. For example, if a call is made to a method from within a loop, then statements within the method may vary with respect to that loop. Additionally, whether a statement within the callee method is stable with respect to a loop may depend on the arguments passed to that method. As a result, the loop-relative stability of the execution of a statement cannot be fully determined through purely static analysis. Instead, it must be calculated at runtime within a specific execution context. To address this, at runtime Conflux records information about execution contexts and passes this information between methods.

The execution context for a particular method call consists of two components: a depth and a level map. The depth of a method call is the number of loops that contain the call to that method. We use $depth(e)$ to denote the depth of an execution context e. The level map of a method call specifies the instability levels of the arguments passed to the method call, the method call's receiver (if the method is an instance method), and the storage location for the return value of the method call (if the method is non-void). We use $\text{level}(e, a)$ to denote the instability level of an argument, method receiver, or return value location a specified by some execution context e.

Before a method call is made from some caller method, m, Conflux constructs an execution context for the call, e'. This execution context is created using the calling method's execution context and the stability classifiers that were determined for the method call's arguments, receiver, and return value location as described in Section 3.3.2. Conflux defines $depth(e')$ to be the sum of $depth(e)$ and the number of loops within m that contain the method call about to be made. Conflux calculates the instability level of each of the arguments passed to the method call using e and the stability classifier for the argument; this value is then recorded in e''s level map. If the callee method is an instance method, Conflux calculates and records the instability level of the method call's receiver using e and the stability classifier for the receiver. If the callee method is a non-void method, Conflux calculates and records the instability level of the method call's return value location using e and the stability classifier of the return value location. Finally, Conflux passes e' to the callee method.

At the program entry point, Conflux creates an initial execution context for the initial method call, since this call does not have a caller from which it would receive an execution context. Thus, Conflux uses an initial execution context e such that $depth(e) = 0$ and $\text{level}(e, a) = 0$ for all method elements a.

Applying the Loop-relative Stability Heuristic

Algorithm 5 Function for calculating an instability level based on a stability classifier c and an execution context e.

```
Input: stability classifier $c$, execution context $e$
Output: an instability level
1: function INSTABILITYLEVEL($c, e$)
2:     if $c$ = STABLE then
3:         return 0
4:     else if $c$ = DEPENDENT$\langle d \rangle$ then
5:         return $\max_{a \in d} \text{level}(e, a)$
6:     else$c$ = UNSTABLE$\langle l \rangle$
7:         return $\text{depth}(e) + |l|$
8:     end if
9: end function
```

At runtime, Conflux combines the static stability classifier (Section 3.3.2) and the dynamic execution context (Section 3.3.2) of an executing program element in order to compute its instability level. This computation is detailed in Algorithm 5. Computed instability levels are used to construct execution contexts as discussed in Section 3.3.2 and, ultimately, to apply the loop-relative stability heuristic. In particular, let b be an execution of a conditional branching statement and a be an execution of an assignment statement that is within the scope of b's control flow. The loop-relative stability heuristic propagates along the control flow from b to a only if b's instability level is less than or equal to a's instability level.

For example, consider the programs shown in Listings 3.3 and 3.4 in Figure 5. These programs are nearly identical; they differ only in the names of methods and the value passed to the call to the method `set` on line 13. However, this slight, semantic difference is enough to the impact instability levels in a way that produces different taint tag propagation.

Consider two program executions: one starting from the `main` method in Listing 3.3 and the other starting from the `main` method in Listing 3.4. Both executions proceed as follows. The `main` method calls an intermediate method, `indexOf` in one case and `contains` in the other. This intermediate method contains one natural loop L (the for-loop on lines 9–15). In both executions, there are two iterations of this loop. On the first iteration, when `i` is `0`, the predicate of the branch on line 12 evaluates to true causing a call to be made to the `set` method. The `set` method assigns the value 0 to a static field `x` and then returns. On the

🟩 STABLE 🟨 DEPENDENT⟨{y}⟩ 🟧 UNSTABLE⟨∅⟩ 🟥 UNSTABLE⟨{L}⟩

set: line 4

Variables

	Value	**Tags**
x	−1	∅
y	0	∅

Execution Context

depth: 1
level(y): 1

indexOf: line 13

Variables

	Value	**Tags**
a[0]	‘h’	{0}
a[1]	‘i’	∅
i	0	∅

Execution Context

depth: 0
level(a): 0

main: line 23

Variables

	Value	**Tags**
a[0]	‘h’	{0}
a[1]	‘i’	∅

Execution Context

depth: 0
level(args): 0

(a)

```java
 1 static int x = -1;
 2
 3 static void set(int y) {
 4     x = y;
 5 }
 6
 7 static void indexOf(char[]
       a) {
 8     // Loop L
 9     for (int i = 0;
10             i < a.length;
11             i++) {
12         if (a[i] == 'h') {
13             set(i);
14         }
15     }
16 }
17
18 public static void main(
       String[] args) {
19     char[] a = new char[] {
20             taint('h', 0),
21             'i'
22     };
23     indexOf(a);
24 }
```

Listing 3.3: A simple Java program containing a single natural loop L (the for-loop on lines 9–15).

```java
 1 static int x = -1;
 2
 3 static void set(int y) {
 4     x = y;
 5 }
 6
 7 static void contains(char[]
       a) {
 8     // Loop L
 9     for (int i = 0;
10             i < a.length;
11             i++) {
12         if (a[i] == 'h') {
13             set(0);
14         }
15     }
16 }
17
18 public static void main(
       String[] args) {
19     char[] a = new char[] {
20             taint('h', 0),
21             'i'
22     };
23     contains(a);
24 }
```

Listing 3.4: A simple Java program that is identical Listing 3.3 except for method names and line 13.

set: line 4

Variables

	Value	**Tags**
x	−1	∅
y	−2	∅

Execution Context

depth: 1
level(y): 0

contains: line 13

Variables

	Value	**Tags**
a[0]	‘h’	{0}
a[1]	‘i’	∅
i	0	∅

Execution Context

depth: 0
level(a): 0

main: line 23

Variables

	Value	**Tags**
a[0]	‘h’	{0}
a[1]	‘i’	∅

Execution Context

depth: 0
level(args): 0

(b)

Figure 5: Listings 3.3 and 3.4 depict Java methods with programs elements colored green, yellow, orange, or red to indicate a stability classifier of STABLE, DEPENDENT⟨{y}⟩, UNSTABLE⟨∅⟩, or UNSTABLE⟨{L}⟩, respectively. Figure 5a shows the stack trace for an execution of the program in Listing 3.3 just before the instruction on line 4 of the `set` method in Listing 3.3 is executed for a program execution starting from the `main` method in Listing 3.3. Figure 5b shows the stack trace just before the instruction on line 4 of the `set` method in Listing 3.4 is executed for a program execution starting from the `main` method in Listing 3.4. Both of these stack traces (Figures 5a and 5b) consist of three frames. For each frame, we show the value (Value) and associated taint tags (Tags) of each local variable within the scope of the frame. We also show the execution context (Execution Context) calculated by CONFLUX for each frame.

second iteration of L, the predicate of the branch on line 12 evaluates to false, and no call is made to the `set` method. After the loop L ends, the intermediate method returns. Then, finally, the method `main` returns.

In both executions, when the branch on line 12 of the intermediate method was taken, the value of the operand `a[0]` of the predicate of that branch was tainted (as depicted in the middle frame of Figures 5a and 5b). Thus, when the assignment statement on line 4 of `set` executed, it wrote a value within the scope of a control flow introduced by a tainted predicate. Figure 5a shows the stack trace for the moment just before the assignment statement on line 4 of the `set` method executed for the execution of the program in Listing 3.3. Similarly, Figure 5b shows the stack trace for the same moment for the execution of the program in Listing 3.4. In order to determine whether to propagate along these control flows, CONFLUX must know the static stability classifiers for the methods called in these two executions and the dynamic execution contexts of the calls made at runtime to these methods.

We annotated Listings 3.3 and 3.4 to indicate the stability classifiers of program elements. These stability classifiers are statically determined by applying the rules described in Section 3.3.2. The assignment statement `x = y` on line 4 of both programs has a stability classifier of DEPENDENT⟨{y}⟩. The storage location `x` always refers to the same location regardless of the context in which it executes. However, whether the value represented by the expression `y` changes over the iterations of some theoretical loop containing a call to `set` depends on whether the argument `y` changes over the iterations of that loop. Thus, this assignment statement has a stability classifier of DEPENDENT$\langle\{y\}\rangle$. The assignment statement `i = 0` on line 9 of both programs has a stability classifier of STABLE because, regardless of the context in which it executes, it always assigns the same value, `0`, to the same location, the local variable `i`. The statement `i++` on line 11 of both programs is an augmented assignment equivalent to `i = i + 1`. The term `i` in the assignment `i = i + 1` is an excluded update term, and the remaining term `1` is constant. Therefore, the statement `i++` on line 11 of both programs has a stability classifiers of STABLE. The stability classifiers of the branch on line 10 of both programs (`i < a.length`) and the branch on line 12 of both programs (`a[i] == 'h'`) are both UNSTABLE$\langle\{L\}\rangle$ because both statements use the value of `i` which changes over the iterations of L. The argument `i` passed to the call to `set` on line 13 of Listing 3.3 has a stability classifier of UNSTABLE$\langle\{L\}\rangle$ because the value of `i` changes over the iterations of L. By contrast, the argument `0` passed to the call to `set` on line 13 of Listing 3.4 has a stability classifier of STABLE because it is a literal. The assignment statement on lines 19–22 of both programs contains a storage allocation, thus its stability classifier is UNSTABLE$\langle\varnothing\rangle$. The argument `a` passed to the method call on line 23 of both programs has a stability classifier UNSTABLE$\langle\varnothing\rangle$ because the value of the reaching definition of `a` from lines 19–22 contains a storage allocation.

In addition to these stability classifiers, CONFLUX must also calculate the dynamic execution contexts of method calls according to the rules described in Section 3.3.2. Let e_{main}, $e_{indexOf}$, and e_{set} denote the execution contexts for the calls made to `main`, `indexOf`, and `set`, respectively, in the program execution for Listing 3.3. Let $e_{main'}$, $e_{contains}$, and $e_{set'}$ denote the execution contexts for the calls made to `main`, `contains`, and `set`, respectively, in the program execution for Listing 3.4.

Since `main` is the program entry point, e_{main} and $e_{main'}$ are initial execution contexts. Thus, $\text{depth}(e_{main})$ and $\text{depth}(e_{main'})$ are 0, and $\text{level}(e_{main}, \texttt{args})$ and $\text{level}(e_{main'}, \texttt{args})$ are also 0. The bottom frame of Figure 5a (labeled "main: line 23") depicts e_{main}, and the bottom frame of Figure 5b (labeled "main: line 23") depicts $e_{main'}$.

CONFLUX constructs $e_{indexOf}$ using e_{main} and $e_{contains}$ using $e_{main'}$. Since there are no natural loops inside `main` that contain the call to `indexOf`, $\text{depth}(e_{indexOf})$ is $\text{depth}(e_{main}) + 0 = 0$. The argument passed to the call to `indexOf` has a stability classifier of UNSTABLE$\langle\varnothing\rangle$, thus $\text{level}(e_{indexOf}, \texttt{a})$ is $\text{depth}(e_{main}) + |\varnothing| = 0$. The middle frame of Figure 5a (labeled "indexOf: line 13") depicts $e_{indexOf}$. Since the `main` methods for the two programs are identical and $e_{main} = e_{main'}$, $e_{contains}$ is identical to $e_{indexOf}$. The middle frame of Figure 5b (labeled "contains: line 13") depicts $e_{contains}$.

Next, CONFLUX constructs e_{set} using $e_{indexOf}$ and $e_{set'}$ using $e_{contains}$. Since the loop L contains the call to `set` in `indexOf`, $\text{depth}(e_{set})$ is $\text{depth}(e_{indexOf}) + 1 = 1$. For the same reason, $\text{depth}(e_{set'})$ is $\text{depth}(e_{contains}) + 1 = 1$ The argument passed to the call to `set` in `indexOf` has a stability classifier of UNSTABLE$\langle\{L\}\rangle$. Therefore, $\text{level}(e_{set}, \texttt{y})$ is $\text{depth}(e_{indexOf}) + |\{L\}| = 1$. However, the argument passed to the call to `set` in `contains` has a stability classifier of STABLE. Therefore, $\text{level}(e_{set'}, \texttt{y})$ is 0. The top frame of Figure 5a (labeled "set: line 4") depicts e_{set}. The top frame of Figure 5b (labeled "set: line 4") depicts $e_{set'}$.

Finally, these execution contexts and stability classifiers can be used to calculate instability levels for the execution of the program in Listing 3.3. The stability classifier of the branch on line 12 of `indexOf` is UNSTABLE$\langle\{L\}\rangle$, and the execution of this branch occurred within the runtime execution context $e_{indexOf}$. Therefore, the instability level of the execution of the conditional branching statement is $\text{depth}(e_{indexOf}) + |\{L\}| = 1$. The stability classifier of the assignment statement on line 4 of `set` is DEPENDENT$\langle\{y\}\rangle$. The execution of the assignment statement occurred within the runtime execution context e_{set}. Therefore, the instability level of the execution of the assignment statement is $\max_{a\in\{y\}} \text{level}(e_{set}, a) = 1$. Overall, the instability level of the execution of the conditional branching statement was less than or equal to that of the execution of the assignment statement. Thus, following the loop-relative stability heuristic, Conflux would propagate along the control flow between the two statement executions.

As was done for the execution of the program in Listing 3.3, the execution contexts and stability classifiers can be used to calculate instability levels for the execution of the program in Listing 3.4. The stability classifier of the branch on line 12 of `contains` is Unstable$\langle\{L\}\rangle$, and the execution of this branch occurred within the runtime execution context $e_{contains}$. Therefore, the instability level of the execution of the conditional branching statement is $\text{depth}(e_{contains}) + |\{L\}| = 1$. The stability classifier of the assignment statement on line 4 of `set` is Dependent$\langle\{y\}\rangle$. The execution of the assignment statement occurred within the runtime execution context $e_{set'}$. So, the instability level of the execution of the assignment statement is $\max_{a\in\{y\}} \text{level}(e_{set'}, a) = 0$. In this case, the instability level of the execution of the conditional branching statement was greater than that of the execution of the assignment statement. Therefore, unlike in the other execution, Conflux would not propagate along the control flow between the two statement executions.

3.4 IMPLEMENTATION

Although our overall approach is language-agnostic and suitable to many languages, we chose Java as a target language, implementing Conflux as an extension to Phosphor (Bell and Kaiser, 2014), a Java taint tracking framework which propagates taint tags by rewriting Java bytecode using the ASM bytecode instrumentation and analysis framework (OW2 Consortium, 2024). Phosphor is a state-of-the-art taint tracking tool upon which several software engineering tools have already been built (Toman and Grossman, 2016; Jiang et al., 2017; Singleton, 2018; Tsai et al., 2020; Velez et al., 2021). Implementing Conflux as an extension to Phosphor allows Conflux to be easily integrated with these tools and any future tools built on top of Phosphor. Furthermore, this choice allows Conflux to support any existing features supported by Phosphor, such as Phosphor's "auto-tainting mode" which allows developers to specify "source" methods at which taint tags are automatically applied and "sink" methods at which taint tags are automatically checked (Bell and Kaiser, 2015). Phosphor creates a shadow variable for each local variable, object field, and method argument to store taint information. It propagates control flows by adding a parameter of the type `ControlFlowStack` to methods' signatures. This allows Phosphor to use `ControlFlowStack` instances to track the scopes of control flows between method boundaries by passing a `ControlFlowStack` instance from the caller method to callee method as an argument. In a similar fashion, Conflux uses `ControlFlowStack` instances to pass loop-relative stability information and execution contexts between methods.

We modified Phosphor to support custom control flow propagation policies by allowing users to extend Phosphor's default behavior at three phases: annotation, instrumentation, and runtime; we then used these extension points to implement Conflux. During the annotation phase, the system is provided with a list containing a method's instructions and may insert annotations, special notes used to inform the instrumentation process, into this list. In the instrumentation phase, Phosphor traverses the instructions of a method forwarding any annotations it encounters to the extending system, informing the extending system when certain structures within the method are encountered, and allowing the extending system to add instructions to the method in response. Phosphor allows a system to modify behavior during the runtime phase by specifying a custom subclass of `ControlFlowStack` for use at runtime.

3.4.1 Annotation

Before a method is instrumented, CONFLUX statically annotates its instructions with control flow information. These annotations mark static features of the method (e.g., the scopes of control flows) as well as properties of those features (e.g., stability classifiers). When the method is instrumented, these features and properties are used to determine what code to generate.

CONFLUX starts by adding annotations to delineate the binding scopes of control flows. The control flow graph of the method being annotated is created to identify branch edges that introduce control flows. Each branch edge that performs an equality check is marked as introducing a control flow that should be propagated at runtime. A branch edge is considered to perform an equality check if one of the following is true:

- it is the edge between a branching instruction conditioned on a non-null equality check and its branch target
- it is the edge between a branching instruction conditioned on a non-null inequality check and the instruction following it
- it is the edge between a `switch` instruction and the branch target associated with one of its non-default cases and no other case for that `switch` instruction has the same branch target
- it is either of the edges out of a branching instruction conditioned on a boolean comparison

The last condition, which deals with boolean comparisons, is a special case. Since there are only two possible values for a boolean, both edges out of the branch are traversed for only a single value. Thus, both edges correspond to equality checks. The Java Virtual Machine does not provide dedicated boolean instructions and instead uses instructions that operate on integer values (Lindholm et al., 2014). As a result, CONFLUX will mark both edges of an integer conditional branch instruction for propagation if it statically determines that the instruction likely performs a boolean comparison.

Once branch edges have been marked for propagation, CONFLUX annotates the start of the scope of the control flow associated with each of the edges. Then, it constructs the modified control flow graph described in Section 3.3.1, and uses Cooper et al. (2006)'s algorithm to calculate the dominance frontier of each replacement node (b_e) associated with each marked branch edge (e). An annotation is added at the beginning of each basic block in the dominance frontier of a replacement node to indicate the end of the scope of the control flow associated with the node.

Next, CONFLUX annotates program elements with their stability classifiers. The method being annotated is converted from Java bytecode into a register-based intermediate representation and then placed into SSA form using Cytron et al. (1991)'s algorithm. CONFLUX maintains a direct correspondence between this intermediate representation and the original Java bytecode so that properties calculated on the intermediate representation can be used to annotate instructions in the original bytecode without having to convert out of the intermediate form. This intermediate representation is used to calculate the stability classifiers of program elements in the method being analyzed (as described in Section 3.3.2). CONFLUX labels each conditional branching statement, assignment statement, non-void return statement, method call receiver, method call argument, and method call return value location with its calculated stability classifier.

Finally, CONFLUX uses the control flow graph for the method to record loop information. Each method call is annotated with the number of natural loops that contains it within the method being annotated. Additionally, CONFLUX records the exit points (i.e., the first instruction of a basic block not contained within a particular loop whose predecessor is contained within that loop) of any loops within the method. These features of the method are used at runtime to calculate execution contexts and instability levels.

3.4.2 Instrumentation and Runtime

During the instrumentation of the method, the annotations added by CONFLUX are used to generate method calls to the `ControlFlowStack`, the structure PHOSPHOR uses to store the taint tags of branches and propagate control flows (Bell and Kaiser, 2015). PHOSPHOR's default behavior adds method calls to push the taint tag associated with each control flow's predicate at the start of its scope and pop any taint tags pushed for each control flow at the end of its scope as described in Section 3.2. CONFLUX modifies this behavior to support the loop-relative stability heuristic.

Before making a method call, the `ControlFlowStack` prepares an execution context for the call. The instability levels of the receiver, arguments, and return value location of the method call are calculated based on their stability classifiers and the current method call's execution context. These instability levels are recorded by the `ControlFlowStack`. `ControlFlowStack` also calculates the number of loops containing the method call based on the number of loops within the current method containing the method call and the current method call's execution context.

At the start of the callee method, the `ControlFlowStack` initializes an execution context based on the values prepared by the caller method. Then, the `ControlFlowStack` pushes this new execution context onto a stack of execution contexts. Before a method exits, the current method call's execution context is popped from this stack. When the start of a control flow's scope is hit, the `ControlFlowStack` calculates the instability level of the control flow based on the conditional branching statement's stability classifier and the current method call's execution context. Then, the `ControlFlowStack` records the taint tag of the predicate of the branch with the flow's instability level. At the end of the scope of a control flow, any taint tags recorded for the flow are cleared. As discussed in Section 3.3.2, in order to use instability levels, all control flows from a branch's predicate must be terminated as soon as the innermost loop relative to which that branch is not stable is exited. Thus, at the exit point of a loop any taint tags recorded at the instability level associated with that loop are removed from the `ControlFlowStack`.

When a non-void return statement or assignment statement executes, the `ControlFlowStack` calculates its instability level based on its stability classifier and the current method call's execution context. The taint tags of any control flow recorded at an instability level less than or equal the instability level of the statement propagate to the program data being assigned a value by the statement.

At instrumentation boundaries, such as native code calls, the callee method will not receive a prepared execution context from the caller. In these cases, CONFLUX assumes that no loops contain the method call. If there are no loops that contain the method call, there are no loops with respect to which the method call's receiver, arguments, and return value location could be non-stable. Thus, CONFLUX uses an execution context e such that $depth(e) = 0$ and $\text{level}(e, a) = 0$ for all method elements a.

3.5 LIMITATIONS

Like prior work from Bao et al. (2010) and Kang et al. (2011), CONFLUX uses a heuristic approach to mitigate control-flow-related over-tainting. Therefore, malicious programs, those intentionally designed to circumvent analyses, are outside the scope of this work. Some applications of dynamic taint tracking, such as confidentially enforcement, may be interested in analyzing malicious programs. However, many applications of dynamic taint tracking primarily consider non-malicious programs (Huo and Clause, 2014; Clause and Orso, 2009; Hough et al., 2020a; Halfond et al., 2006; Attariyan and Flinn, 2010).

Additionally, since Conflux relies on a heuristic rather than a sound or complete flow analysis it is difficult to determine how appropriate it is for a particular application. Furthermore, the heuristic we use relies on conservative assumptions about values loaded from arrays and fields, and it only takes into account direct uses of a local variable when determining its storage location stability. The loop-relative stability heuristic does not consider cycles introduced via recursion which could potentially impact its applicability to languages that favor recursion over loops. Nonetheless, in our evaluation of real-world Java programs, we found our approach to be quite effective.

Conflux does not address all sources of control-flow-related over-tainting. For example, if an element conditionally added to a resizable array triggers an array resize, then all the elements copied from the original array to the new, larger array will be tainted with the label associated with the branch that triggered the element to be added to the resizable array. However, mature Java code typically uses `System.arraycopy` to copy elements from the original array to the new, larger array. `System.arraycopy` is a native method, and therefore not instrumented by Phosphor. Phosphor uses predefined propagation logic for many native methods (regardless of the propagation policy). If this were not the case, there would likely be over-tainting in `System.arraycopy`. However, code performing this sort of array resizing is often contained within a single method. Therefore, it could be identified statically and handled specially by a taint tracking system.

Finally, Conflux does not aim to address over-tainting from sources other than control-flow propagation, e.g., bit-packing and caches. This is an interesting topic for future work.

3.6 EVALUATION

We performed an evaluation of Conflux to answer the following research questions:

RQ1: How accurately does Conflux propagate taint tags?

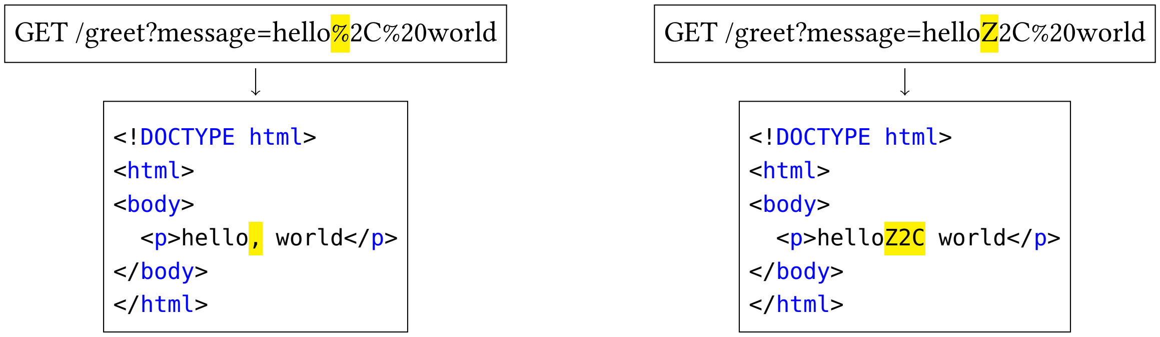
RQ2: How does Conflux's performance vary with input sizes?

RQ3: How does Conflux impact a concrete application of taint tracking?

RQ1 and RQ2 examine the utility of Conflux. For these questions we report metrics, like F1 score, which quantify the accuracy of Conflux by comparing taint tags propagated by Conflux against a ground truth. F1 score is calculated as $\frac{TP}{TP+0.5*(FP+FN)}$ where TP is the number of true positives, FP is the number of false positives, and FN is the number of false negatives. However, it is difficult to know for an arbitrary, real-world program which taint tags should propagate to which values and, therefore, challenging to specify a ground truth.

Past works have explored automated approaches for determining ground truth taint tag sets. For example, Bao et al. (2010) and Jee (2015) used automated approaches to determine ground truth taint tag sets in their evaluations of taint tracking systems. These automated approaches run the same program multiple times with different inputs and compare the outputs. Jee (2015) interpreted differences in the outputs for different input values as indicating that taint tags from the input should have propagated to the output. Bao et al. (2010) assume that if different input values lead to the same output, the input's tag should not propagate to the output. However, it is infeasible to exhaustively explore any non-trivial input space. Thus, these approaches can only consider a sample of possible input values and their efficacy is tied to the quality of that sampling. This is problematic because automatically generating diverse samples from an input space is a challenging problem in its own right.

Furthermore, in ordered output, like text, different inputs may cause outputs to shift positions without impacting their actual values. This makes it challenging to determine which outputs were impacted by a particular change to the input. For example, consider a simple program that receives an HTTP request containing a "message" query parameter and returns an HTML document that contains that parameter. The inputs used in the execution in Figure 6a and Figure 6b differ only in a single character. However, the outputs differ in every character after "hello". An automated approach may misinterpret this change and expect the taint tag of the changed input character to propagate to every output character following "hello".



```
GET /greet?message=hello%2C%20world
```

```
<!DOCTYPE html>
<html>
<body>
  <p>hello, world</p>
</body>
</html>
```

(a)

```
GET /greet?message=helloZ2C%20world
```

```
<!DOCTYPE html>
<html>
<body>
  <p>helloZ2C world</p>
</body>
</html>
```

(b)

Figure 6: Two sample executions of a program that receives an HTTP request and replies with an HTML document. In each execution, the value of the "message" query parameter of the input HTTP request is inserted into the output HTML document. Yellow is used to highlight differences between the executions.

Due to these issues, we chose to not use an automated approach for determining expected taint tag sets and, instead, manually determined expected taint tag sets. Manually determining expected taint tag sets for arbitrary, real-world programs is error-prone and subjective. However, it is possible to leverage known properties of a program to determine a ground truth. This approach was used by McCamant and Ernst (2008) to evaluate Flowcheck, their system for measuring the amount of information that flows from secret program inputs to public outputs. One of the experiments conducted by McCamant and Ernst (2008) used *bzip2*, a lossless compression tool, as an evaluation subject. For their purposes, *bzip2* was an appealing subject because the expected flow size for a valid input could easily be determined due to the nature of the tool. Specifically, when presented with a compressible, secret input, all of *bzip2*'s output (except a small amount of input-independent output like headers) will leak secret information because *bzip2*'s compression algorithm is lossless.

Like McCamant and Ernst (2008), we chose to leverage program properties to construct a ground truth for programs included in our benchmark for evaluating RQ1 and RQ2. Unfortunately, this limited the types of programs that could be included in the benchmark to those that met certain criteria. These criteria are discussed in detail in Section 3.6.2. However, the subjects used to evaluate RQ3 were not impacted by this limitation because we do not report quantitative metrics for RQ3 and, therefore, do not require a known ground truth.

RQ3 explores the practical impacts of Conflux on an application of taint tracking. For this purpose, we chose to examine Clause and Orso (2009)'s approach for identifying failure-relevant inputs. The primary assessment originally performed by Clause and Orso (2009) was qualitative. Therefore, we decided to also provide a qualitative assessment in our case study instead of a quantitative one. This choice allowed us to be less constrained in our selection of subject applications for the case study since we no longer needed to determine a ground truth.

3.6.1 Experimental Setup

We evaluated CONFLUX in comparison to three other propagation policies: DATA-ONLY, BASIC-CONTROL, and SCD, an implementation of the control flow semantics presented by Bao et al. (2010). Each of these policies was implemented using PHOSPHOR. All four of the policies propagate taint tags along data flows in a similar fashion to what was described by Bell and Kaiser (2014). Additionally, all the policies propagate taint tags from an index used to access an element of an array to the element accessed; this applies to both read and write accesses. Only the DATA-ONLY and BASIC-CONTROL policies propagate taint tags through *INSTANCEOF* operations as that instruction does not match the semantics targeted by CONFLUX and SCD. CONFLUX, BASIC-CONTROL, and SCD all propagate taint tags through pointer dereferencing. Each of the policies uses different semantics for propagating control flows. DATA-ONLY does not propagate along control flows. BASIC-CONTROL propagates along all control flows using the standard scoping semantics described in Section 3.2. SCD uses the standard scoping semantics, but only propagates along edges corresponding to equality checks. These edges are determined using the same rules described in Section 3.4.1.

All of our experiments were conducted on OpenJDK version 1.8.0_222. Tests for each propagation policy were run in a JVM that was instrumented according to the policy. Before a test started any taint tags on values in the JVM were cleared to ensure that taint tags from one test could not impact the results of another test.

3.6.2 Benchmark

The benchmark we created to answer RQ1 and RQ2 consists of methods for encoding and decoding text drawn from the OpenJDK Java Class Library (version 1.8.0_222) (Oracle Corporation, 2024i) and seven different real-world Java projects:

- Apache Commons Text (version 1.8) (Apache Software Foundation, 2019b)
- Apache Commons Codec (version 1.14) (Apache Software Foundation, 2019a)
- Bouncy Castle Provider (version 1.46) (Legion of the Bouncy Castle Inc., 2011)
- Guava (version 28.2-jre) (Google LLC, 2020)
- jsoup (version 1.11.3) (Jonathan Hedley, 2024)
- Spring Web (version 5.2.5) (Pivotal Software, 2020)
- Tomcat Embed Core (version 9.0.19) (Apache Software Foundation, 2024)

To the best of our knowledge, DroidBench (Arzt et al., 2014) is the only existing Java taint tracking benchmark that contains tests that consider control flows. However, DroidBench only contains four tests involving control flows and was designed for evaluating static analyzers. Thus, we choose not to use it in our evaluation.

The programs featured in our benchmark for encoding and decoding text are all information preserving. Each of these programs transforms one representation of a sequence of abstract entities into another representation of the same sequence of abstract entities. An abstract entity is an atomic unit of information. For example, when escaping text for inclusion in HTML, the character <, the character entity reference &lt;, and the numeric character reference &#60; all represent the same abstract entity. The information-preserving nature of the programs included in our benchmark provides a clear ground truth: the expected set of taint tags for a program output contains the taint tags of the program inputs that represent the same abstract entity that the output represents. For example, a program might perform percent encoding on the string :@ resulting in the output string %3A%40. The first percent-encoded octet, %3A, represents the same abstract entity represented by the input character :. The second percent-encoded octet, %40, represents the same abstract entity represented by the input character @. Thus, the expected label set for each of the characters in the first octet would contain the unique label assigned to :, and the expected label set for each of the characters in the second octet would contain the unique label assigned to @.

Each method selected for the benchmark we created had to meet several criteria. First, the method had to be deterministic. For valid input, the output of the method had to represent the same sequence of abstract entities as the input. A taint tracking policy that does not propagate control flows should not produce false positives when tracking taint tags through the method. Likewise, a taint tracking policy that propagates along every control flow using the standard semantics described in Section 3.2 should not produce false negatives when tracking taint tags through the method. This limits the scope of the benchmark to situations in which observed under-tainting or over-tainting is likely related to control flow propagation, as opposed to other sources of imprecision such as caches and bit-packing.

Every test in the benchmark uses one of the selected methods and follows the same general format. The test starts by creating an input representing a sequence of abstract entities appropriate for the method. Each character (or byte in the case of hex encoding) in the input is assigned a unique taint label. The test then transforms the tainted input using the target test method. The taint label set that is propagated to each character (or byte in the case of hex decoding) of the method's output is compared to the set of labels expected for that element. Labels in both the propagated and expected sets are counted as true positives. Labels in the expected set, but not the propagated set are counted as false negatives. Labels in the propagated set, but not the expected set are counted as false positives.

In some cases, a single method is used in multiple tests (e.g., the method `PercentCodec.encode()` from Apache Commons Codec is used in tests in the groups *unicode-percent-decode* and *reserved-percent-decode*). In these instances, we choose to separate different transformations performed by a single method into multiple tests so that their results could be more directly compared to a different implementation of the same transformation.

3.6.3 RQ1: Accuracy

In order to compare the accuracy of DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX, we applied each of the propagation policies to the tests in our benchmarks. For each test we created an input sequence of eight abstract entities (RQ2 considers different length inputs). We recorded the number of false negatives, false positives, and true positives that were reported by each propagation policy on each test.

Table 4 presents our findings for the different propagation polices on the benchmarks. Overall, CONFLUX had the highest F1 score on a majority of the tests, forty-three out of the forty-eight total tests. DATA-ONLY had the highest F1 score on twenty-three tests. SCD had the highest F1 score on twenty-one tests. And finally, BASIC-CONTROL had the highest F1 score on only three tests.

Due to the selection criteria for the methods used in the benchmark, DATA-ONLY reports no false positives and BASIC-CONTROL reports no false negatives. In some of the tests, DATA-ONLY fails to propagate any

Table 4: Comparison of DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX on our benchmark. Each row reports results for a single test. Tests are grouped by the type of transformation they perform. For each of the propagation policies, we report the number false negatives (FN), the number of false positives (FP), the number of true positives (TP), and the F1 score (F1) recorded for each test. The highest F1 score or scores for each test are colored red.

			DATA-ONLY				BASIC-CONTROL				SCD				CONFLUX			
Test Group	Project	Implementation	F1	FN	FP	TP	F1	FN	FP	TP	F1	FN	FP	TP	F1	FN	FP	TP
hex-decode	Apache Commons Codec	Hex	1.00	0	0	16	0.22	0	112	16	1.00	0	0	16	1.00	0	0	16
	Bouncy Castle Provider	Hex	1.00	0	0	16	1.00	0	0	16	1.00	0	0	16	1.00	0	0	16
	Java Class Library	DatatypeConverter	1.00	0	0	16	0.22	0	112	16	1.00	0	0	16	1.00	0	0	16
	Tomcat Embed Core	HexUtils	1.00	0	0	16	0.22	0	112	16	1.00	0	0	16	1.00	0	0	16
hex-encode	Apache Commons Codec	Hex	1.00	0	0	16	0.22	0	112	16	1.00	0	0	16	1.00	0	0	16
	Bouncy Castle Provider	Hex	1.00	0	0	16	0.22	0	112	16	1.00	0	0	16	1.00	0	0	16
	Java Class Library	DatatypeConverter	1.00	0	0	16	1.00	0	0	16	1.00	0	0	16	1.00	0	0	16
	Tomcat Embed Core	HexUtils	1.00	0	0	16	1.00	0	0	16	1.00	0	0	16	1.00	0	0	16
html-escape	Apache Commons Text	StringEscapeUtils	0.00	36	0	0	0.22	0	252	36	1.00	0	0	36	1.00	0	0	36
	Guava	HtmlEscapers	0.00	36	0	0	0.22	0	252	36	1.00	0	0	36	1.00	0	0	36
	Spring Web	HtmlUtils-ISO-8859-1	0.00	36	0	0	0.22	0	252	36	1.00	0	0	36	1.00	0	0	36
	Spring Web	HtmlUtils-UTF-8	0.00	36	0	0	0.22	0	252	36	1.00	0	0	36	0.00	36	0	0
	jsoup	Entities	0.00	36	0	0	0.22	0	252	36	0.54	0	62	36	1.00	0	0	36
html-unescape	Apache Commons Text	StringEscapeUtils	0.00	36	0	0	0.22	0	252	36	1.00	0	0	36	1.00	0	0	36
	Spring Web	HtmlUtils	0.71	16	0	20	0.22	0	252	36	0.56	0	56	36	0.71	16	0	20
javascript-escape	Apache Commons Text	StringEscapeUtils	0.50	16	0	8	0.22	0	168	24	0.54	4	30	20	0.91	4	0	20
	Spring Web	JavaScriptUtils	0.00	24	0	0	0.22	0	168	24	0.62	0	30	24	1.00	0	0	24
javascript-unescape	Apache Commons Text	StringEscapeUtils	0.59	14	0	10	0.30	0	112	24	0.65	0	26	24	0.96	2	0	22
quoted-printable-decode	Apache Commons Codec	QuotedPrintableCodec	0.80	14	0	28	0.22	0	294	42	0.48	0	90	42	1.00	0	0	42
quoted-printable-encode	Apache Commons Codec	QuotedPrintableCodec	0.80	14	0	28	0.22	0	294	42	0.28	10	156	32	0.80	14	0	28
reserved-percent-decode	Apache Commons Codec	PercentCodec	0.80	8	0	16	0.22	0	168	24	0.63	0	28	24	1.00	0	0	24
	Apache Commons Codec	URLCodec	0.80	8	0	16	0.22	0	168	24	0.46	0	56	24	1.00	0	0	24
	Java Class Library	URLDecoder	0.80	8	0	16	0.22	0	168	24	0.46	0	56	24	0.63	0	28	24
	Spring Web	UriUtils	0.80	8	0	16	0.22	0	168	24	0.46	0	56	24	1.00	0	0	24
	Tomcat Embed Core	UDecoder	0.80	8	0	16	0.22	0	168	24	0.63	0	28	24	1.00	0	0	24
reserved-percent-encode	Apache Commons Codec	PercentCodec	0.80	8	0	16	0.22	0	168	24	0.80	8	0	16	0.80	8	0	16
	Apache Commons Codec	URLCodec	0.80	8	0	16	0.22	0	168	24	0.22	0	168	24	0.80	8	0	16
	Guava	UrlEscapers	0.80	8	0	16	0.22	0	168	24	0.70	0	21	24	0.80	8	0	16
	Java Class Library	URLEncoder	0.80	8	0	16	0.22	0	168	24	0.22	0	168	24	0.80	8	0	16
	Spring Web	UriUtils	0.80	8	0	16	0.22	0	168	24	0.80	8	0	16	0.80	8	0	16
	Tomcat Embed Core	UEncoder	0.80	8	0	16	0.22	0	168	24	0.36	0	84	24	0.80	8	0	16
spaces-url-decode	Apache Commons Codec	PercentCodec	0.00	8	0	0	0.22	0	56	8	0.36	0	28	8	1.00	0	0	8
	Apache Commons Codec	URLCodec	0.00	8	0	0	0.22	0	56	8	0.22	0	56	8	1.00	0	0	8
	Java Class Library	URLDecoder	0.00	8	0	0	0.22	0	56	8	0.22	0	56	8	1.00	0	0	8
	Tomcat Embed Core	UDecoder	0.00	8	0	0	0.22	0	56	8	0.36	0	28	8	1.00	0	0	8
spaces-url-encode	Apache Commons Codec	PercentCodec	0.00	8	0	0	0.22	0	56	8	0.36	0	28	8	1.00	0	0	8
	Apache Commons Codec	URLCodec	0.00	8	0	0	0.22	0	56	8	1.00	0	0	8	0.00	8	0	0
	Guava	UrlEscapers	0.00	8	0	0	0.22	0	56	8	0.70	0	7	8	1.00	0	0	8
	Java Class Library	URLEncoder	0.00	8	0	0	0.22	0	56	8	0.70	0	7	8	0.00	8	0	0
unicode-percent-decode	Apache Commons Codec	PercentCodec	0.80	20	0	40	0.22	0	420	60	0.37	0	204	60	1.00	0	0	60
	Apache Commons Codec	URLCodec	0.80	20	0	40	0.22	0	420	60	0.37	0	204	60	1.00	0	0	60
	Java Class Library	URLDecoder	0.80	20	0	40	0.22	0	420	60	0.30	0	276	60	0.64	0	68	60
	Spring Web	UriUtils	0.80	20	0	40	0.22	0	420	60	0.30	0	276	60	1.00	0	0	60
unicode-percent-encode	Apache Commons Codec	PercentCodec	0.80	20	0	40	0.22	0	420	60	0.80	20	0	40	0.80	20	0	40
	Apache Commons Codec	URLCodec	0.80	20	0	40	0.22	0	420	60	0.80	20	0	40	0.80	20	0	40
	Guava	UrlEscapers	0.75	24	0	36	0.22	0	420	60	0.50	22	54	38	0.75	24	0	36
	Java Class Library	URLEncoder	0.80	20	0	40	0.22	0	420	60	0.80	20	0	40	0.80	20	0	40
	Spring Web	UriUtils	0.80	20	0	40	0.22	0	420	60	0.80	20	0	40	0.80	20	0	40

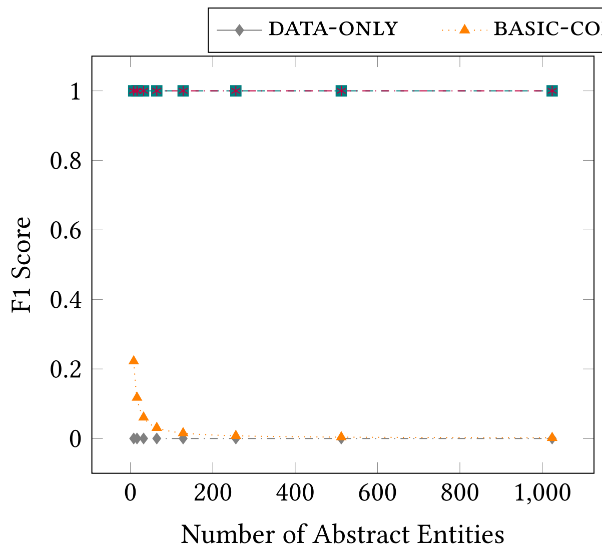
tags to the output because there are no data flows between the input and the output. For example, in the *html-escape* test using jsoup's `Entities` class, input values flow into a `switch` statement which selects a constant string to append to the output; all of the information that flows between the input and the output is transferred through the `switch` statement. In other tests, DATA-ONLY does report some or all of the possible true positives because data flows are able to fully or partially capture the relationship between the input and the output. For instance, for tests in the *reserved-percent-encode* group, DATA-ONLY reports two true positives for every false negative. This occurs because tests in this group take a sequence of Uniform Resource Identifier (URI)-reserved characters and encode them using percent-encoded octets (e.g., the character @ would end up encoded as %40). There is a data flow between the value of the input character and the two hex digits in the octet, but there is only a control flow between the input character and the percent sign. Thus, data-only correctly tracks the relationship between an input character and the two hex digits of the output octet. However, it misses the relationship between an input character and the percent sign of the output octet. By contrast, basic-control never missed a relationship between an input and an output, but reported a relatively large number of false positives on all but three tests. In many cases, basic-control marked all the inputs as being related to all the outputs.

SCD reported relatively few false negatives, and 102 out of the 132 of these false negatives occurred in tests from the *unicode-percent-encode* group. Tests in the *unicode-percent-encode* group take a sequence of non-ASCII characters and transform them into UTF-8, percent-encoded octets. Typical implementations of this transformation determine whether an input character needs to be encoded either by checking if it falls into some value range, is outside some value range, or is not present in some set of values that do not need to be encoded. These checks are generally not equality checks, so SCD does not propagate along the branches associated with them resulting in under-tainting. Conflux also under-taints in these tests for the same reason.

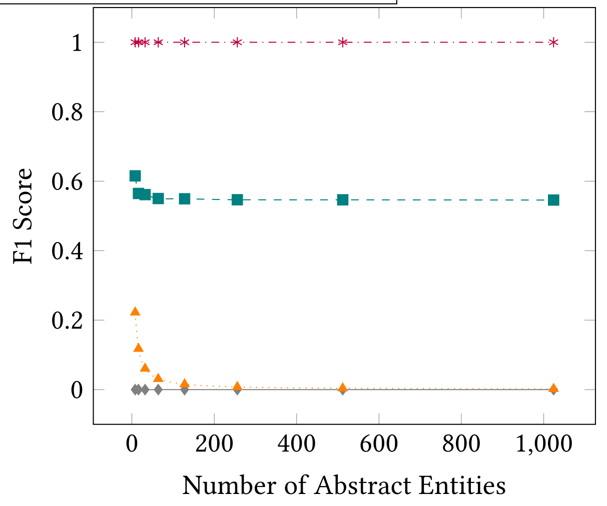
Conflux only reported false positives in two tests, as opposed to the twenty-eight tests in which SCD reported false positives and the forty-five tests in which basic-control reported false positives. Both of these two tests have the same problematic flow. A simplified version of this flow is shown in Listing 3.5. Conflux marks both the branch on line 6 and the statement on line 12 as unstable with respect to all loops that contain them. Thus, Conflux propagates taint tags from the predicate of the branch on line 6 to the assigned value on line 12, `c`, resulting in over-tainting. In this case, the loop header (`c == '%'`) is the source of the flow, rather than the flow occurring within the body of the loop. Had the loop header instead been written as `input[inputPosition] == '%'`, then the load would have been considered as outside of the binding scope of the loop header, and Conflux would not have over-tainted.

```
 1 public static String decode(char[] input) {
 2   char[] output = new char[input.length / 3];
 3   int outputPosition = 0;
 4   int inputPosition = 0;
 5   char c = input[0];
 6   while(c == '%') { // source of problematic flow
 7     output[outputPosition++] = hexToChar(input[inputPosition + 1], input[inputPosition + 2])
       ;
 8     inputPosition += 3;
 9     if(inputPosition + 2 >= input.length) {
10       break;
11     }
12     c = input[inputPosition]; // target of problematic statement
13   }
14   return new String(output);
15 }
```

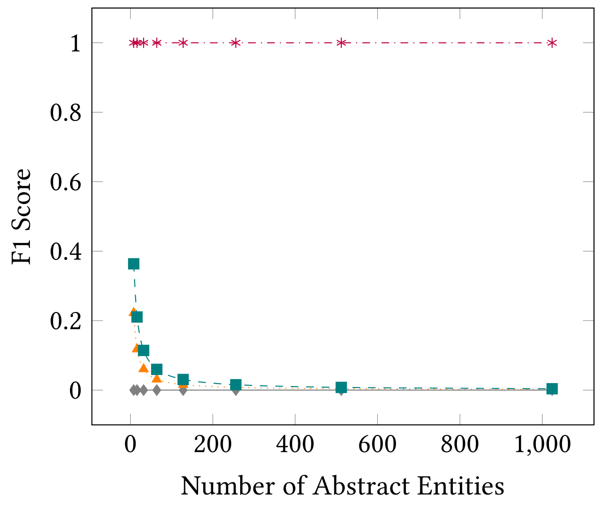
Listing 3.5: Simplified code for the tests where Conflux reports false positives.



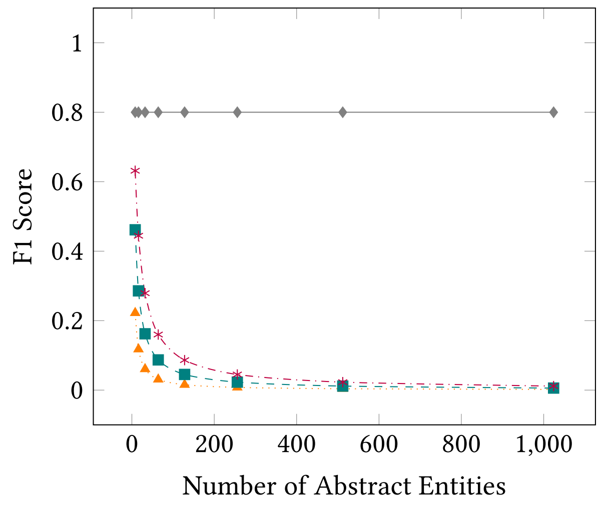
(a) Results for the *html-escape* test using Guava's `HtmlEscapers` class.



(b) Results for the *javascript-escape* test using Spring Web's `JavaScriptUtils` class.



(c) Results for the *spaces-url-decode* test using Apache Commons Codec's `PercentCodec` class.



(d) Results for the *reserved-percent-decode* test using the Java Class Library's `UrlDecoder` class.

Figure 7: Comparison of DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX on selected tests from our benchmark for varying input lengths.

3.6.4 RQ2: Accuracy Versus Input Size

Taint tags often accumulate on program data over loop iterations leading to progressively more over-tainting on each iteration. Many common applications of taint tracking (e.g., fuzzing guidance) tend to use relatively large inputs which often trigger a large number of loop iterations. In these cases, it may be impractical to use the standard semantics for control flow propagation due to the "explosion" of taint tags resulting from their accumulation in loops. To evaluate whether CONFLUX could be used to address this issue, we applied the propagation policies (DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX) to our benchmark with inputs of various lengths (8, 16, 32, 64, 128, 256, 512, and 1024 abstract entities). We then recorded the F1 score for each policy on each test for each of the lengths and produced a series of plots. We examined these plots to determine how the F1 score for each policy changed as the size of the input scaled. Figure 7 shows four of these plots that represent common cases seen across many of the plots.

In all the tests, the F1 score for DATA-ONLY was constant as the size of the input increased. The behavior of the F1 scores for the other three policies either stayed constant, decreased to some non-zero value, or decreased to zero as the size of the input increased. We divided tests into categories based on how the F1 score reported for the different policies changed as the size of the input increased.

There were three tests where the F1 scores for all the policies remained constant as the input size increased. In seventeen tests, the F1 scores for DATA-ONLY, SCD, and CONFLUX remained constant, but the F1 scores for BASIC-CONTROL decreased to zero. A plot for one of these tests is depicted in Figure 7a. In six tests, the F1 scores for DATA-ONLY and CONFLUX remained constant, the F1 scores for BASIC-CONTROL decreased to zero, and the F1 scores for SCD decreased to some non-zero value. A plot for one of these tests is depicted in Figure 7b. In twenty of the tests, the F1 scores for DATA-ONLY and CONFLUX remained constant, but the F1 scores for BASIC-CONTROL and SCD decreased to zero. A plot for one of these tests is depicted in Figure 7c. There were only two tests in which the F1 scores for DATA-ONLY remained constant, and the F1 scores for BASIC-CONTROL, SCD, and CONFLUX decreased to zero. A plot for one of these tests is depicted in Figure 7d.

Overall, CONFLUX's F1 score stayed constant in all but two tests, similar to DATA-ONLY. By contrast, the F1 score of BASIC-CONTROL and SCD typically degraded as input sizes scaled. In this respect, CONFLUX's control flow tracking behaved more similarly to data flow tracking than the control flow tracking performed by BASIC-CONTROL and SCD.

3.6.5 RQ3: Impact on a Concrete Application of Taint Tracking

To examine the effect of CONFLUX on a practical application of taint tracking, we implemented a prototype of Clause and Orso (2009)'s approach for identifying which failure-inducing inputs (i.e, inputs that produce a failure) are failure-relevant (i.e, useful for analyzing the failure). We used this prototype to perform a case study exploring the impact of CONFLUX on Clause and Orso (2009)'s approach. We conducted an experiment similar to the one originally performed by Clause and Orso (2009). In this experiment, we provide a qualitative assessment of the failure-inducing inputs that were marked as relevant by the different propagation policies, DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX, on five failures from popular, open-source Java projects. Our assessment is limited to the impact of the propagation policies on the values marked as failure-relevant; it does not examine the usefulness of Clause and Orso (2009)'s approach in general since that was explored in the original work.

Table 5 details the five failures that we included in our case study. Each of the these failures was chosen from an issue reported in a project's issue tracker that described a system failure. Only issues in which the system accepted a human-interpretable input or inputs were considered since it would otherwise be difficult to apply Clause and Orso (2009)'s approach. For the sake of simplicity, we only selected issues in which the described failure could be consistently reproduced. Furthermore, we only selected issues that had already been fixed in a single, associated commit in order to facilitate analysis of the failure. A similar criterion was used by Just et al. (2014) in the selection of bugs for Defects4J, a widely-used database of real Java defects. We used these fixing commits along with the issues filed in the issue trackers to construct appropriate failure-inducing inputs for each failure.

Additionally, for each of the failures, we identified developer comments made in either the fixing commit or the issue that discussed the conditions that produced the failure. These comments reflect the developers' understanding of the failure and the conditions under which the failure manifests. However, mapping these conditions to specific portions of the input is subjective, and the developers' understanding of the failure may be flawed. Thus, these developer-identified failure conditions cannot be used as a ground truth for the failure-relevant portions of an input. For each of the failures, we also created a simplified, failure-inducing input using a combination of random trials of reduced inputs and Zeller and Hildebrandt (2002)'s *ddmin* algorithm. The *ddmin* algorithm produces a failure-inducing input that is guaranteed to be "1-minimal", i.e, removing any single element would cause the input to no longer induce the failure; however, the simplified input is not guaranteed to be minimal (Zeller and Hildebrandt, 2002). Even though a simplified, failure-inducing input can be useful in understanding a failure, it does not necessarily correspond to the failure-relevant portions of the original, failure-inducing input. For example, even a minimized, failure-inducing input can contain portions of the input that are necessary to pass a validation step, but not necessarily useful for investigating the failure. While neither developer-identified failure conditions nor simplified failure-inducing inputs can be used as a ground truth for the failure-relevant portions of an input, they both provide valuable insights into the nature of a failure. Therefore, we used these insights to guide our qualitative assessment of the failure-inducing inputs that were marked as relevant by the different propagation policies.

Table 5: Evaluation subjects used in the RQ3 case study. For each subject, we list the name and version of the project (Project), the issue in which the failure was reported (Issue), and the commit in which the bug that produced the failure was corrected (Fix).

Project		Issue		Fix	
Checkstyle (version 8.37)	[38]	#8934	[37]	70c7ae0	[36]
Google Closure Compiler (version v20140814)	[47]	#652	[46]	aac5d11	[45]
Mozilla Rhino (version 1.7.11)	[124]	#539	[123]	0c0bb39	[122]
OpenRefine (version 3.4-SNAPSHOT)	[149]	#2584	[148]	825e687	[147]
H2 Database Engine (version 1.4.200)	[76]	#2550	[75]	6c564e6	[74]

Our prototype implementation applies a unique taint tag to each character input presented to the applications. These taint tags are propagated in accordance with a particular propagation policy as described in Section 3.6.1. In addition to this policy-based propagation, our prototype employs special propagation logic for exceptions. If a Java exception is thrown by the execution of an instruction (as detailed in the JVM Specification (Lindholm et al., 2014)), then our prototype propagates the taint tags of the operands of that instruction to the exception. Any input values associated with taint tags that propagate to the exception that produces the studied system failure are marked as failure-relevant.

For each of the bugs that we analyzed, we display the entire program input with annotations that specify which portions of the input were marked as failure-relevant by each policy. These visualizations demonstrate the impact of propagation policies on a concrete software engineering application of taint tracking. Across all five examples, DATA-ONLY marks only a single character of input as failure-relevant, demonstrating the need to propagate taint tags along control flows in this application. Meanwhile, BASIC-CONTROL marks almost every input character as failure-relevant, underscoring the consequences of control-flow-related over-tainting. We discuss in detail the input values marked as failure-relevant by each policy for each of the studied failures below.

```
throw
```

Listing 3.6: The simplified, failure-inducing JavaScript input for the Closure failure reported in issue #652 (Closure Compiler Authors, 2014b).

```

2
```

Listing 3.7: The simplified, failure-inducing CSV input for the OpenRefine failure reported in issue #2584 (OpenRefine Contributors, 2020b). The empty first line is required to induce the failure.

```
{
    "guessCellValueTypes":
    true,
    "trimStrings": true
}
```

Listing 3.8: The simplified, failure-inducing JSON input for the OpenRefine failure reported in issue #2584 (OpenRefine Contributors, 2020b). Whitespace characters have been added to improve the readability of the input.

```
function() {
    try {
    } finally {
        v
    }
  yield
}
```

Listing 3.9: The simplified, failure-inducing JavaScript input for the Rhino failure reported in issue #539 (MDN Contributors, 2019b). Whitespace characters have been added to improve the readability of the input.

```
CREATE TABLE t;
MERGE INTO t
USING (SELECT 1)
ON ()
WHEN NOT MATCHED AND b THEN
     INSERT VALUES()
```

Listing 3.10: The simplified, failure-inducing SQL input for the H2 failure reported in issue #2550 (H2 Contributors, 2020b). Whitespace characters have been added to improve the readability of the input.

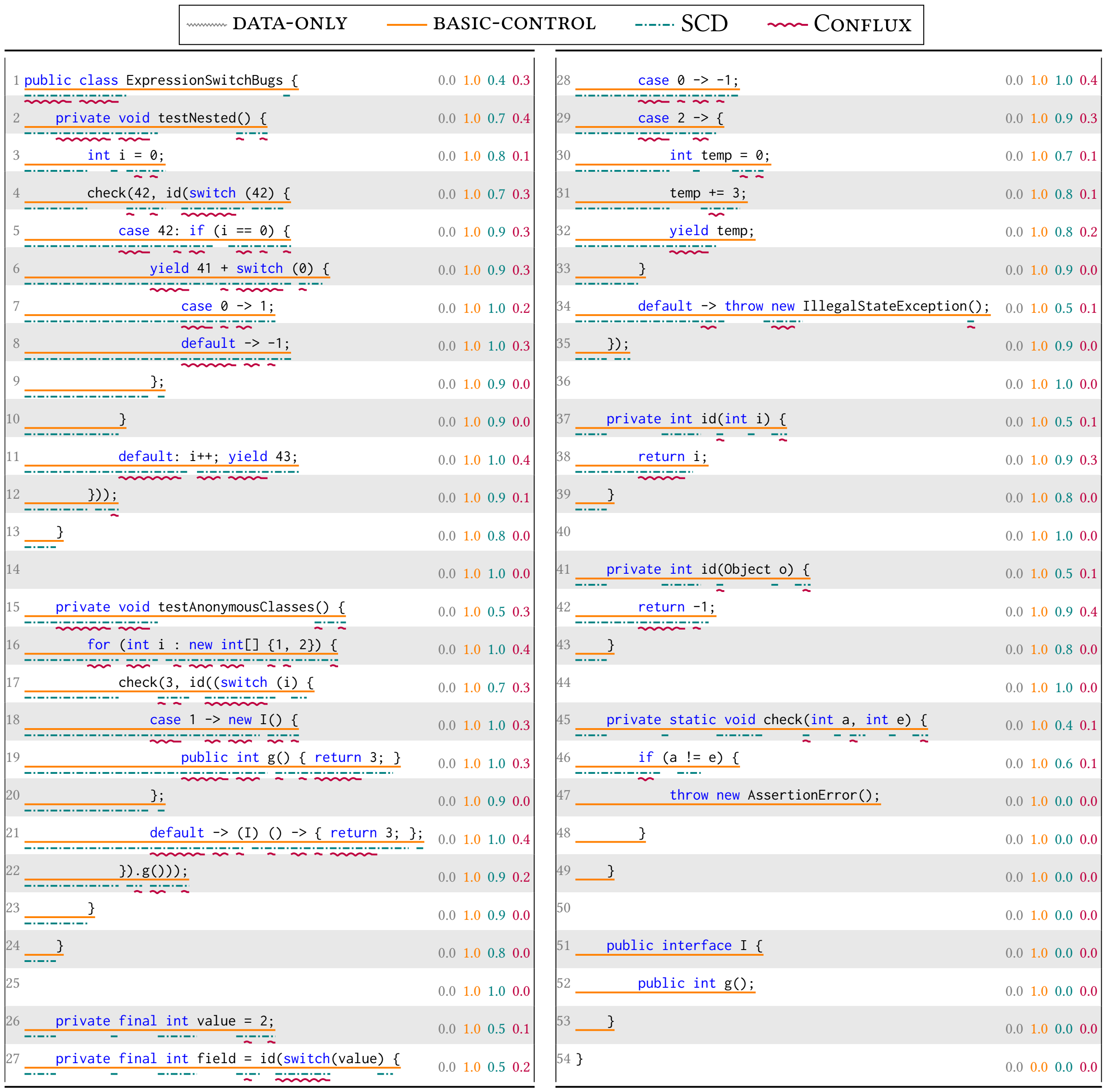
```
class E {
    d t = (switch(a) {
        case 0 -> 1;
        case 2 -> n;
    })
}
```

Listing 3.11: The simplified, failure-inducing Java input for the Checkstyle failure reported in issue #8634 (Checkstyle Contributors, 2020b). Whitespace characters have been added to improve the readability of the input.

Checkstyle

Checkstyle is a static analysis tool that finds and reports violations of coding standards in Java code (Checkstyle Contributors, 2020c). We studied the failure reported in Checkstyle issue #8934 (Checkstyle Contributors, 2020b). A Checkstyle developer described this failure by saying, "FinalLocalVariable throws a NPE on Switch expression in assignment" (Checkstyle Contributors, 2020b). However, in the fixing commit for the failure, a different Checkstyle developer noted that, "assigning to [the] `switch` is not the problem . . . wrapping [the] `switch` inside a function foo() makes the problem disappear" (Checkstyle Contributors, 2020a). These developer comments indicate that the failure reported in Checkstyle issue #8934 occurs when the "FinalLocalVariable" rule is applied to a Java source code class containing a `switch` expression that is not contained within a method-level or block-level scope (Checkstyle Contributors, 2020a,b).

The failure-inducing input that we used to reproduce the failure reported in Checkstyle issue #8934 was a Java source code class based on the Java source code class included in issue #8934 (Checkstyle Contributors, 2020b). Figure 8 shows the input Java source code class in its entirety along with the portions of the input identified as failure-relevant by each propagation policy. Listing 3.11 depicts the simplified, failure-inducing input produced for the failure-inducing input in Figure 8. This simplified, failure-inducing input consists of a Java class declaration containing a field declaration that initializes the field to be the value of a `switch` expression. Both the developer-identified failure conditions and the simplified, failure-inducing input suggest that the `switch` expression that appears on lines 27 through 35 of Figure 8 is the cause of the failure.



```
public class ExpressionSwitchBugs {                          0.0 1.0 0.4 0.3
    private void testNested() {                              0.0 1.0 0.7 0.4
        int i = 0;                                           0.0 1.0 0.8 0.1
        check(42, id(switch (42) {                           0.0 1.0 0.7 0.3
            case 42: if (i == 0) {                           0.0 1.0 0.9 0.3
                yield 41 + switch (0) {                      0.0 1.0 0.9 0.3
                    case 0 -> 1;                             0.0 1.0 1.0 0.2
                    default -> -1;                           0.0 1.0 1.0 0.3
                };                                           0.0 1.0 0.9 0.0
            }                                                0.0 1.0 0.9 0.0
            default: i++; yield 43;                          0.0 1.0 1.0 0.4
        }));                                                 0.0 1.0 0.9 0.1
    }                                                        0.0 1.0 0.8 0.0
                                                             0.0 1.0 1.0 0.0
    private void testAnonymousClasses() {                    0.0 1.0 0.5 0.3
        for (int i : new int[] {1, 2}) {                     0.0 1.0 1.0 0.4
            check(3, id((switch (i) {                        0.0 1.0 0.7 0.3
                case 1 -> new I() {                          0.0 1.0 1.0 0.3
                    public int g() { return 3; }             0.0 1.0 1.0 0.3
                };                                           0.0 1.0 0.9 0.0
                default -> (I) () -> { return 3; };          0.0 1.0 1.0 0.4
            }).g()));                                        0.0 1.0 0.9 0.2
        }                                                    0.0 1.0 0.9 0.0
    }                                                        0.0 1.0 0.8 0.0
                                                             0.0 1.0 1.0 0.0
    private final int value = 2;                             0.0 1.0 0.5 0.1
    private final int field = id(switch(value) {             0.0 1.0 0.5 0.2
        case 0 -> -1;                                        0.0 1.0 1.0 0.4
        case 2 -> {                                          0.0 1.0 0.9 0.3
            int temp = 0;                                    0.0 1.0 0.7 0.1
            temp += 3;                                       0.0 1.0 0.8 0.1
            yield temp;                                      0.0 1.0 0.8 0.2
        }                                                    0.0 1.0 0.9 0.0
        default -> throw new IllegalStateException();        0.0 1.0 0.5 0.1
    });                                                      0.0 1.0 0.9 0.0
                                                             0.0 1.0 1.0 0.0
    private int id(int i) {                                  0.0 1.0 0.5 0.1
        return i;                                            0.0 1.0 0.9 0.3
    }                                                        0.0 1.0 0.8 0.0
                                                             0.0 1.0 1.0 0.0
    private int id(Object o) {                               0.0 1.0 0.5 0.1
        return -1;                                           0.0 1.0 0.9 0.4
    }                                                        0.0 1.0 0.8 0.0
                                                             0.0 1.0 1.0 0.0
    private static void check(int a, int e) {                0.0 1.0 0.4 0.1
        if (a != e) {                                        0.0 1.0 0.6 0.1
            throw new AssertionError();                      0.0 1.0 0.0 0.0
        }                                                    0.0 1.0 0.0 0.0
    }                                                        0.0 1.0 0.0 0.0
                                                             0.0 1.0 0.0 0.0
    public interface I {                                     0.0 1.0 0.0 0.0
        public int g();                                      0.0 1.0 0.0 0.0
    }                                                        0.0 1.0 0.0 0.0
}                                                            0.0 0.0 0.0 0.0
```

Figure 8: Java input used to reproduce the Checkstyle failure reported in issue #8634 (Checkstyle Contributors, 2020b). Failure-relevant input regions identified by DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX are underlined in gray, orange, teal, and magenta respectively. To the right of each line of input, the ratio between the total number of characters on that line and the number of characters on that line marked as failure-relevant by DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX is displayed in gray, orange, teal, and magenta respectively. This ratio includes newline characters.

As shown in Figure 8, DATA-ONLY did not mark any portions of the input as failure-relevant. By contrast, BASIC-CONTROL reported the entire input text except the final closing bracket as failure-relevant. Both SCD and CONFLUX report large portions of the `switch` expression that triggered the failure including the `switch` keyword. However, both of these policies also report other regions of the input that are not likely to be helpful to developers trying to debug the failure. CONFLUX reports fewer of these regions than SCD.

Google Closure Compiler

The Google Closure Compiler is a tool for compiling and optimizing JavaScript code (Closure Compiler Authors, 2014c). The failure we selected from the Google Closure Compiler was reported in issue #652 (Closure Compiler Authors, 2014b). In the fix for this failure, a Closure developer noted that Closure should "report a parse error if there is a throw followed by a semicolon or newline" (Closure Compiler Authors, 2014a). This developer continued by noting that according to grammar for JavaScript, "'throw;' or 'throw \n expr;' are illegal" (Closure Compiler Authors, 2014a). These developer comments indicate that the failure manifests when Closure compiles code containing a malformed throw statement where the `throw` keyword is followed by a semicolon or newline and not an expression.

We reproduced the failure reported in issue #652 using JavaScript source code based on the JavaScript source code provided in the original issue. Figure 9a shows the input JavaScript source code in its entirety along with the portions of the input identified as failure-relevant by each propagation policy. Listing 3.6 depicts the simplified, failure-inducing input produced for the failure-inducing input in Figure 9a. The simplified, failure-inducing input consists of just the keyword `throw`. Both the developer-identified failure conditions and the simplified, failure-inducing input indicate that the failure described in issue #652 is triggered by the malformed throw statement on line 9 of Figure 9a.

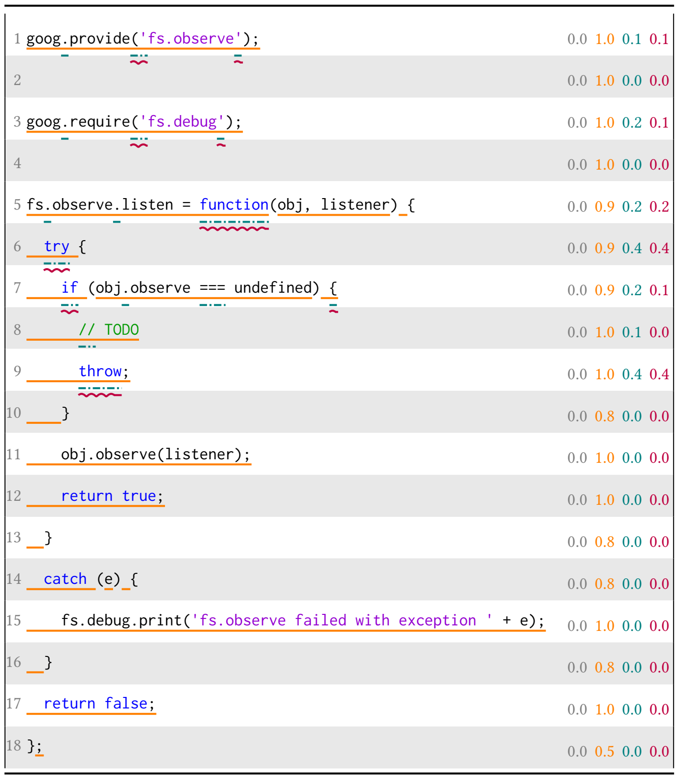
As in the Checkstyle failure, DATA-ONLY did not mark any portions of the input as failure-relevant. BASIC-CONTROL, SCD, and CONFLUX all marked the malformed throw statement as failure-relevant. However, BASIC-CONTROL also marked almost every other input character as failure-relevant. SCD and CONFLUX report some additional regions of the input that may obfuscate the cause of the failure. Once again, CONFLUX reports fewer of these regions than SCD.

Mozilla Rhino

Mozilla Rhino is an implementation of JavaScript written in Java (MDN Contributors, 2019c). Rhino includes a compiler for translating JavaScript source code into Java class files. Issue #539 (MDN Contributors, 2019b) describes the failure that we examined. In the fix for this failure, a Rhino developer noted that the failure "cropped up when generators were used in a function that had a try..catch..finally block and a yield after the finally" (MDN Contributors, 2019a). This comment indicates that the failure manifests when Rhino tries to compile JavaScript source code that contains a generator function or legacy generator function in which a `yield` expression is present after a `finally` block.

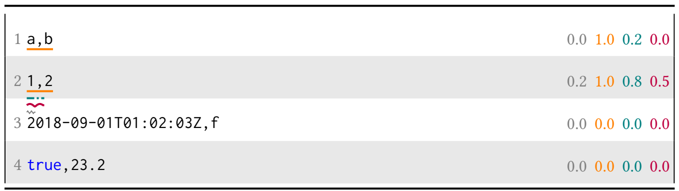
We reproduced the failure reported in issue #539 using JavaScript source code based on a test case that was added in the commit that fixed the failure. Figure 9c shows the input JavaScript source code in its entirety along with the portions of the input identified as failure-relevant by each propagation policy. Listing 3.9 depicts the simplified, failure-inducing input produced for the failure-inducing input in Figure 9c. The simplified, failure-inducing input contains a legacy generator function with a `yield` expression following a `try-finally` statement. Both the developer-identified failure conditions and the simplified, failure-inducing input indicate that the failure described in issue #539 is related to the `finally` on line 6 and the `yield` on line 11 of Figure 9c.

DATA-ONLY
BASIC-CONTROL
SCD
CONFLUX



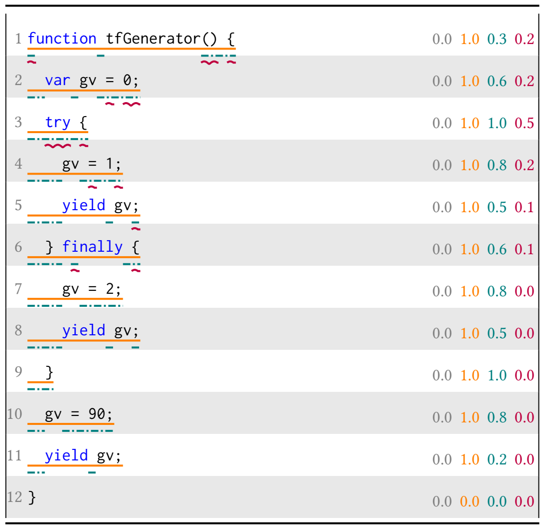
```
goog.provide('fs.observe');                                        0.0 1.0 0.1 0.1
                                                                   0.0 1.0 0.0 0.0
goog.require('fs.debug');                                          0.0 1.0 0.2 0.1
                                                                   0.0 1.0 0.0 0.0
fs.observe.listen = function(obj, listener) {                      0.0 0.9 0.2 0.2
  try {                                                            0.0 0.9 0.4 0.4
    if (obj.observe === undefined) {                               0.0 0.9 0.2 0.1
      // TODO                                                      0.0 1.0 0.1 0.0
      throw;                                                       0.0 1.0 0.4 0.4
    }                                                              0.0 0.8 0.0 0.0
    obj.observe(listener);                                         0.0 1.0 0.0 0.0
    return true;                                                   0.0 1.0 0.0 0.0
  }                                                                0.0 0.8 0.0 0.0
  catch (e) {                                                      0.0 0.8 0.0 0.0
    fs.debug.print('fs.observe failed with exception ' + e);       0.0 1.0 0.0 0.0
  }                                                                0.0 0.8 0.0 0.0
  return false;                                                    0.0 1.0 0.0 0.0
};                                                                 0.0 0.5 0.0 0.0
```

(a) JavaScript input used to reproduce the Closure failure reported in issue #652 (Closure Compiler Authors, 2014b).



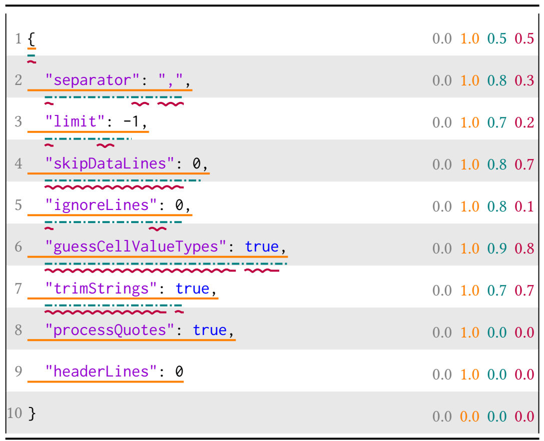
```
a,b                            0.0 1.0 0.2 0.0
1,2                            0.2 1.0 0.8 0.5
2018-09-01T01:02:03Z,f         0.0 0.0 0.0 0.0
true,23.2                      0.0 0.0 0.0 0.0
```

(b) CSV input used to reproduce the OpenRefine failure reported in issue #2584 (OpenRefine Contributors, 2020b).



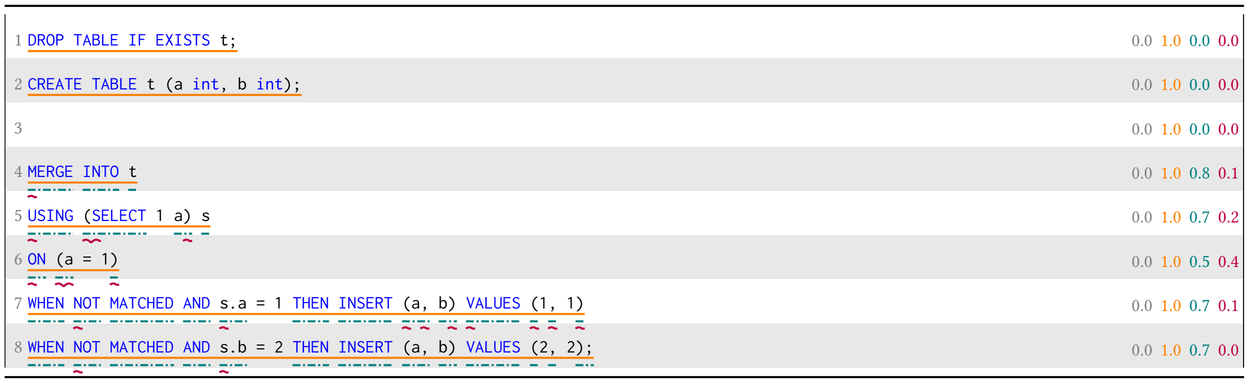
```
function tfGenerator() {       0.0 1.0 0.3 0.2
  var gv = 0;                  0.0 1.0 0.6 0.2
  try {                        0.0 1.0 1.0 0.5
    gv = 1;                    0.0 1.0 0.8 0.2
    yield gv;                  0.0 1.0 0.5 0.1
  } finally {                  0.0 1.0 0.6 0.1
    gv = 2;                    0.0 1.0 0.8 0.0
    yield gv;                  0.0 1.0 0.5 0.0
  }                            0.0 1.0 1.0 0.0
  gv = 90;                     0.0 1.0 0.8 0.0
  yield gv;                    0.0 1.0 0.2 0.0
}                              0.0 0.0 0.0 0.0
```

(c) JavaScript input used to reproduce the Rhino failure reported in issue #539 (MDN Contributors, 2019b).



```
{                                    0.0 1.0 0.5 0.5
  "separator": ",",                  0.0 1.0 0.8 0.3
  "limit": -1,                       0.0 1.0 0.7 0.2
  "skipDataLines": 0,                0.0 1.0 0.8 0.7
  "ignoreLines": 0,                  0.0 1.0 0.8 0.1
  "guessCellValueTypes": true,       0.0 1.0 0.9 0.8
  "trimStrings": true,               0.0 1.0 0.7 0.7
  "processQuotes": true,             0.0 1.0 0.0 0.0
  "headerLines": 0                   0.0 1.0 0.0 0.0
}                                    0.0 0.0 0.0 0.0
```

(d) JSON input used to reproduce the OpenRefine failure reported in issue #2584 (OpenRefine Contributors, 2020b).



```
DROP TABLE IF EXISTS t;                                                      0.0 1.0 0.0 0.0
CREATE TABLE t (a int, b int);                                               0.0 1.0 0.0 0.0
                                                                             0.0 1.0 0.0 0.0
MERGE INTO t                                                                 0.0 1.0 0.8 0.1
USING (SELECT 1 a) s                                                         0.0 1.0 0.7 0.2
ON (a = 1)                                                                   0.0 1.0 0.5 0.4
WHEN NOT MATCHED AND s.a = 1 THEN INSERT (a, b) VALUES (1, 1)                0.0 1.0 0.7 0.1
WHEN NOT MATCHED AND s.b = 2 THEN INSERT (a, b) VALUES (2, 2);               0.0 1.0 0.7 0.0
```

(e) SQL input used to reproduce the H2 failure reported in issue #2550 (H2 Contributors, 2020b).

Figure 9: Failure-relevant input regions identified by DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX are underlined in gray, orange, teal, and magenta respectively. To the right of each line of input, the ratio between the total number of characters on that line and the number of characters on that line marked as failure-relevant by DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX is displayed in gray, orange, teal, and magenta respectively. This ratio includes newline characters.

```java
L: switch (s.length()) {
   case 2: c=s.charAt(1);
   if (c=='f') { if (s.charAt(0)=='i') {id=Id_if; break L0;} }
   else if (c=='n') { if (s.charAt(0)=='i') {id=Id_in; break L0;} }
   else if (c=='o') { if (s.charAt(0)=='d') {id=Id_do; break L0;} }
   break L;
   case 7: switch (s.charAt(1)) {
       case 'a': X="package";id=Id_package; break L;
       case 'e': X="default";id=Id_default; break L;
       case 'i': X="finally";id=Id_finally; break L;
       case 'o': X="boolean";id=Id_boolean; break L;
       case 'r': X="private";id=Id_private; break L;
       case 'x': X="extends";id=Id_extends; break L;
   } break L;
```

Listing 3.12: Part of the Java code used by Mozilla Rhino for lexing JavaScript source code (MDN Contributors, 2019c).

As shown in Figure 8, DATA-ONLY did not mark any portions of the input as failure-relevant and BASIC-CONTROL marked almost every input character as failure-relevant. Unexpectedly, SCD and CONFLUX often marked only part of a keyword as failure relevant, for example, both policies marked only the "i" in `finally` as failure relevant. This was due to the structure of the code that Rhino uses for lexing, converting characters sequences into tokens, JavaScript inputs (MDN Contributors, 2019c). Part of this code is displayed in Listing 3.12. When processing the the input string "finally' with this code, there is a control flow from the input character "i" to the produced token. However, there is not a control flow relationship between the other input characters and the produced token.

OpenRefine

OpenRefine is a tool for manipulating, managing, and visualizing data (OpenRefine contributors, 2020). We studied the OpenRefine failure reported in issue #2584 (OpenRefine Contributors, 2020b). A developer for OpenRefine described this failure as occurring "when both trim and autodetect are enabled in tabular parser" (OpenRefine Contributors, 2020a). In this and other comments, the developer notes that if both the "trimStrings" and "guessCellValueTypes" configuration options are enabled, OpenRefine will crash when importing a numeric value using an importer that subclasses `TabularImportingParserBase` (OpenRefine Contributors, 2020a,b).

Two separate inputs were needed to reproduce the failure reported in issue #2584 (OpenRefine Contributors, 2020b) The first input was a comma-separated values (CSV) file containing input data to be imported that was provided in issue #2584 (OpenRefine Contributors, 2020b). The other input was a JavaScript object notation (JSON) configuration file that was based on a test case that was added in the commit that fixed that the failure. Figure 9b shows the CSV input in its entirety and Figure 9d shows the JSON input in its entirety. Both figures show which portions of the input were identified as failure-relevant by each propagation policy. Listing 3.7 depicts the simplified, failure-inducing input produced for the failure-inducing, CSV input in Figure 9b. The simplified, failure-inducing input contains two rows: an empty header row and a row containing a single numeric value. Listing 3.8 depicts the simplified, failure-inducing input produced for the failure-inducing, JSON input in Figure 9d. This input is a JSON object with two properties: "guessCellValueTypes" and "trimStrings". The value of both of these properties is the boolean `true`. Both the developer-identified failure conditions and the simplified, failure-inducing input suggest that the JSON properties "guessCellValueTypes" and "trimStrings", and their values are relevant to the failure. Additionally, both the developer-identified failure conditions and the simplified, failure-inducing input indicate that a numeric value in the CSV input is relevant to the failure. The fixing commit suggests that first numeric value in the CSV input that is not in a header row (the number "1" on line 2 of Figure 9b) triggers the failure (OpenRefine Contributors, 2020a).

As depicted in Figure 9b, all the propagation policies, even DATA-ONLY, marked the numeric value that triggered the failure (the number "1" on line 2 of Figure 9b) as failure-relevant. This is the only input value ever marked as failure-relevant by DATA-ONLY in any of the failures we examined. BASIC-CONTROL marked additional portions of the CSV input as failure-relevant likely obfuscating the true cause of the failure. For the JSON input depicted in Figure 9d, DATA-ONLY did not mark any portions of the input as failure-relevant and BASIC-CONTROL marked almost every input character as failure-relevant. SCD and CONFLUX reported the properties "guessCellValueTypes" and "trimStrings", and their values as failure-relevant. However, they both reported additional regions of the JSON input that are unlikely to be helpful to a developer trying to debug the failure. CONFLUX reports fewer of these regions than SCD.

H2 Database Engine

H2 is a Relational Database Management System (RDBMS) implemented in Java that supports a subset of SQL (H2 Contributors, 2020c). H2 issue #2550 (H2 Contributors, 2020b) details the failure that we selected. In the fixing commit for this failure, an H2 developer described the failure as a "NullPointerException with MERGE containing unknown column in AND condition of WHEN" (H2 Contributors, 2020a). This comment indicates that the failure occurs when H2 tries to execute a `MERGE` statement containing a `WHEN NOT MATCHED` clause with an `AND` expression that refers to an unknown column (H2 Contributors, 2020a,b).

We reproduced the failure reported in issue #2550 using the SQL statements provided in issue #2550 (H2 Contributors, 2020b). Figure 9e shows the input SQL statements in their entirety along with the portions of the input identified as failure-relevant by each propagation policy. Listing 3.10 depicts the simplified, failure-inducing input produced for the failure-inducing input in Figure 9e. The simplified, failure-inducing input contains a `CREATE TABLE` statement and a `MERGE` statement. Both the developer-identified failure conditions and the simplified, failure-inducing input indicate that the command `MERGE`, the clause `WHEN NOT MATCHED`, the keyword `AND`, and the unknown reference, "s.b", that appears on line 8 of Figure 9e are relevant to the failure.

As depicted in Figure 9e, DATA-ONLY did not mark any portions of the input as failure-relevant. Conversely, BASIC-CONTROL reported the entire input as failure-relevant. SCD and CONFLUX only marked portions of the `MERGE` statement as failure-relevant. SCD marked almost the entire `MERGE` statement as failure-relevant. CONFLUX only marked small portions of the `MERGE` statement as relevant; it is unclear whether these smaller portions better illuminate the cause of the failure.

Summary

When considering the input values marked as failure-relevant by each propagation policy, it is clear that DATA-ONLY and BASIC-CONTROL are unlikely to be useful to a developer attempting to debug a failure. This underscores the need for alternative taint tag propagation semantics. The portions of each input marked as failure-relevant by SCD and CONFLUX appear to be more useful for analyzing the failures. SCD tended to mark more characters as failure-relevant than CONFLUX and, in some cases, marked most of the input as failure-relevant.

3.6.6 Threats to Validity

One threat to the validity of our experiments stems from our selection of evaluation subjects for the benchmark. Our benchmark tests a limited number of methods from only a handful of projects. As discussed in Section 3.6, it is challenging to determine which taint tags should propagate to which values for an arbitrary, real-world program. Thus, only methods that met certain criteria (detailed in Section 3.6.2) could be included in the benchmark. As a result, the benchmark is not necessarily representative of all Java programs. However, we selected these methods based on a search for popular Java libraries.

Additionally, the ground truth expected label sets we used for the benchmark may not be appropriate for every application of taint tracking. For example, in some applications it could be desirable for propagated labels to reflect looser or stronger relationships than those reflected in our ground truth. Nonetheless, we feel that our ground truth selection follows best practices of state-of-the-art taint tracking evaluations (McCamant and Ernst, 2008; Pauck et al., 2018).

Unlike the benchmark, the case study evaluation that we performed did not require a manually specified ground truth. However, the case study evaluation was limited to a single application of taint tracking and examined a limited number of subjects and failures. Therefore, it may not generalize to other applications of taint tracking or other subjects.

3.7 CONCLUSION

Techniques that require high-precision, dynamic taint tracking are highly prevalent in software engineering research (Mera, 2019; Shen et al., 2015; Attariyan and Flinn, 2010; Huo and Clause, 2014; Rawat et al., 2017; Wang et al., 2010; Ganesh et al., 2009; Saoji et al., 2017; Enck et al., 2010a; Hough et al., 2020a). Many of these techniques would greatly benefit from accurately propagated control flow relationships. However, the standard control flow propagation semantics are mismatched with the standard data flow semantics. This mismatch often results in severe over-tainting making the standard control flow propagation semantics impractical for most applications. Prior approaches to mitigate this over-tainting fail to address many of its fundamental sources. Conflux, our alternative control flow propagation semantics, decreases the scope of control flows and leverages a novel heuristic, loop-relative stability, to determine whether a control flow's taint tags should propagate to a particular statement. We compared Conflux to three other control flow propagation policies on a benchmark containing forty-eight tests consisting of programs for encoding and decoding text. Conflux had the highest F1 score on forty-three out of the forty-eight total tests when using test inputs of a fixed size. Additionally, when the size of test inputs scaled, Conflux's F1 score remained constant on all but two of the forty-eight tests indicating that Conflux helped to mitigate taint explosions associated with large inputs. We also examined the impact of Conflux on a concrete application of taint tracking, automated debugging.

3.8 STATUS

This work was published in *Transactions on Software Engineering and Methodology* (TOSEM) in 2021 (Hough and Bell, 2021b). It was presented at the *37th IEEE/ACM International Conference on Automated Software Engineering* (ASE 2022) as part of the "journal-first" track. The associated artifact for this work is publicly available under a BSD 3-Clause License (Hough and Bell, 2021a).

4 CROSSOVER IN PARAMETRIC FUZZING

4.1 INTRODUCTION

Early identification of software defects is crucial for mitigating their impact and reducing the cost of repairing them. Automated input generation techniques facilitate the identification of defects, thereby supporting the production of reliable, high-quality software. Evolutionary fuzzing is a prominent technique for automatically generating test inputs that leverages information about system executions to bias the exploration of an input space towards promising areas in order to maximize the diversity of explored execution behaviors. Over the course of a fuzzing campaign, inputs are "evolved" by maintaining a population of interesting inputs that have been discovered and creating new inputs by perturbing these interesting inputs. This evolutionary search process depends on the assumption that an input created by making a small change to an interesting input is more likely to reveal new execution behaviors than a purely random input. Evolutionary fuzzing has been shown to be effective at finding defects in real world systems (Michał Zalewski, 2024; OSS-Fuzz Contributors, 2024; LLVM Project, 2024; Padhye et al., 2019b; Holler et al., 2012; Pham et al., 2021).

However, if the system under test has a highly constrained input structure, then even small changes to an input are likely to violate those constraints. Thus, an evolutionary fuzzer that is unaware of the system's input structure may struggle to create valid inputs that exercise the core functionality of the system. Padhye et al. (2019b) proposed and demonstrated the effectiveness of parametric fuzzing, a technique for performing structure-aware, evolutionary fuzzing. Parametric fuzzers use QuickCheck (Claessen and Hughes, 2000)-style generators to achieve structural awareness. In parametric fuzzing, generators are used to map fuzzer-created bit strings, referred to as "parameter sequences", to structures (Padhye et al., 2019b; Kukucka et al., 2022; Nguyen and Grunske, 2022; Reddy et al., 2020). The parametric fuzzer provides a means of splitting the parameter sequence into arbitrary, primitive-typed values. These arbitrary values are then used by one or more generators to build structured test inputs that conform to user-defined constraints.

The strength of parametric fuzzing is that it supports generators of arbitrary types without requiring developers to define type-specific mutation operators (operators that modify a single input) and crossover operators (operators that combine parts from multiple inputs together) to perturb inputs. Since a parameter sequence is a bit string, parametric fuzzers are able to modify parameter sequences using generic mutation and crossover operators. However, existing parametric fuzzers do not support crossover and use only mutation operators to modify parameter sequences (Padhye et al., 2019b; Kukucka et al., 2022; Nguyen and Grunske, 2022; Reddy et al., 2020).

Prior work has demonstrated the effectiveness of crossover in structured fuzzing (Pham et al., 2021; Holler et al., 2012; Aschermann et al., 2019). Furthermore, many unstructured fuzzers, such as AFL (Michał Zalewski, 2024), AFL++ (Fioraldi et al., 2020), and libFuzzer (LLVM Project, 2024), utilize some form of general-purpose crossover operator. Some fuzzers, like AFL++ (Fioraldi et al., 2020) and libFuzzer (LLVM Project, 2024), even support custom, user-defined crossover operators. The efficacy of crossover operators stems from their ability to produce children that inherit advantageous traits from multiple parent inputs—a property referred to as heritability (Rothlauf, 2006; Raidl and Gottlieb, 2005).

Unfortunately, parametric fuzzing lacks an explicit tree structure rendering tree-based crossovers operators like the ones proposed by Pham et al. (2021), Holler et al. (2012), and Aschermann et al. (2019) inapplicable. Furthermore, the nature of parametric fuzzing limits the effectiveness of unstructured crossover operators like the ones found in AFL and libFuzzer. Our key insight is that a tree structure can be extracted from the hierarchy of method calls made by generators in parametric fuzzing. Using this insight, we designed a new crossover operator for parametric fuzzing—"linked crossover". Empirically comparing our approach to traditional crossover operators, we found that linked crossover produces new inputs that inherit more desirable traits from their parents. In a comparison against three state-of-the-art parametric fuzzers, we found that applying linked crossover increased branch coverage and defect detection rates. Overall, chapter work makes the following contributions:

- A description of linked crossover, a novel crossover operator for parametric fuzzing that leverages call tree information to intelligently combine inputs.
- An open-source implementation of linked crossover for Java.
- An empirical comparison of the heritability and effectiveness of different crossover operators in parametric fuzzing.
- An evaluation of the effectiveness of linked crossover on seven Java projects against three state-of-the-art parametric fuzzers: Zest (Padhye et al., 2019b), BeDivFuzz (Nguyen and Grunske, 2022), and RLCheck (Reddy et al., 2020).

4.2 BACKGROUND

4.2.1 Evolutionary Fuzzing

Algorithm 6 depicts a generic, evolutionary fuzzing algorithm that is similar to the ones used by AFL (Michał Zalewski, 2024) and libFuzzer (LLVM Project, 2024) The evolutionary fuzzer maintains a population of interesting inputs. The fuzzer repeatedly creates and executes new inputs. These inputs are created by selecting a parent input from the population and perturbing that input by applying a number of mutation and crossover operators to produce a child. This child is then executed as an input to the fuzzing target and, the fuzzer observes execution feedback. The type of execution feedback depends on the fuzzer; branch coverage is a common choice (Padhye et al., 2019b; Michał Zalewski, 2024; LLVM Project, 2024). If the child exercises new coverage, it is saved to a corpus of coverage-revealing inputs. If the child induced a new failure, it is saved to a directory of failure-inducing inputs. Lastly, the population may be updated, typically to include this new child input if the child revealed new system behavior.

4.2.2 Crossover

Crossover (sometimes also referred to as recombination or splicing) is an evolutionary operator that produces new child inputs by combining multiple parent inputs with the goal of passing on desirable traits from the parents to the children (Mitchell et al., 1992). Typically, crossover operators exchange segments from two parents between a number of "crossover points". For example, the crossover operator used by AFL and AFL++ is a one-point crossover—it combines two inputs by splicing them together at a randomly selected midpoint (Michał Zalewski, 2024; Fioraldi et al., 2020). Evolutionary search approaches

Algorithm 6 A generic, evolutionary fuzzer.

```
1:  $failures \leftarrow \{\}$
2:  $totalCoverage \leftarrow \{\}$
3:  $population \leftarrow \{\}$
4:  while there is time remaining in the campaign do
5:      if $population$ is empty then
6:          $child \leftarrow$ a new random input
7:      else
8:          $parent \leftarrow \text{select}(population)$
9:          $child \leftarrow \text{modify}(parent, population)$
10:     end if
11:     $coverage, feedback, failure \leftarrow \text{execute}(child)$
12:     if $\exists x \in coverage : x \notin totalCoverage$ then
13:         save $child$ to the corpus
14:         $totalCoverage \leftarrow totalCoverage \cup coverage$
15:     end if
16:     if $failure \neq \square \wedge failure \notin failures$ then
17:         save $child$ to the failures directory
18:         $failures \leftarrow failures \cup \{failure\}$
19:     end if
20:     $population \leftarrow \text{update}(population, feedback, child)$
21: end while
```

commonly use one- or two-point crossover since performance may degrade as the number of crossover points increases (De Jong and Spears, 1991; De Jong, 1975). A crossover operator is most effective when it is able to recombine high-fitness, interesting subcomponents from separate parents into a single child (Mitchell et al., 1992; Holland, 2000; Watson and Jansen, 2007).

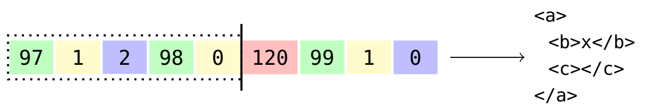
4.2.3 Parametric Fuzzing

When a system under test has a highly constrained input structure, small modifications to an input are likely to produce invalid inputs preventing the fuzzer from exercising the core functionality of the system. Parametric fuzzing overcomes this limitation by using QuickCheck-style generators to achieve structural awareness (Padhye et al., 2019b; Kukucka et al., 2022; Nguyen and Grunske, 2022). The parametric fuzzer provides a means of splitting parts of fuzzer-created bit strings, known as "parameter sequences", into arbitrary, primitive-typed values. QuickCheck-style generators create complex input structures using these arbitrary values, thereby creating a "parametric generator" that maps parameter sequences to generated structures. For example, consider the `generate` method in Listing 4.1.

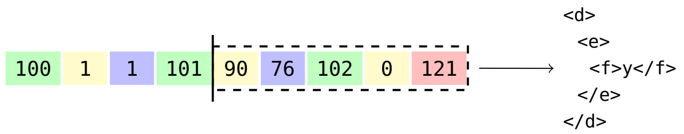
The method `nextByte` is provided by the parametric fuzzer; it consumes and returns the next byte of the parameter sequence. The `generate` method recursively creates an XML element. The call to `nextByte` on line 2 selects a tag name for the XML element. The value returned by the call to `nextByte` on line 4 determines whether the element should have child elements or text content. If the element has children, the call to `nextByte` on line 5 determines the number of children. Otherwise, the call to `nextByte` on line 11 selects the text content of the element.

```
 1 public String generate() {
 2   char n = (char) nextByte();
 3   String c = "";
 4   if (nextByte() > 0) {
 5     int x = nextByte() % 5;
 6     for (int i = 0;
 7         i < x; i++) {
 8       c += generate();
 9     }
10   } else {
11     c += (char) nextByte();
12   }
13   return "<" + n + ">" + c +
14     "</" + n + ">";
15 }
```

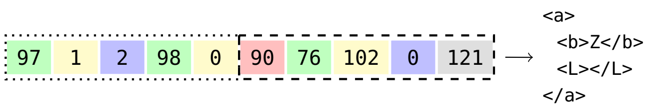
Listing 4.1: A simple Extensible Markup Language (XML) document generator.



(a) Parent A. The solid line marks the crossover point. The dotted line encloses the prefix contributed to the child produced from the crossover operation.



(b) Parent B. The solid line marks the crossover point. The dashed line encloses the suffix contributed to the child produced from the crossover operation.



(c) Child produced by performing one-point crossover on Parents A and B. The dotted and dashed lines enclose the portions of the child's parameter sequence that were transferred from Parents A and B, respectively.

Figure 10: A generator (left) and associated inputs (right). For each input, we depict the bytes of the parameter sequence (left of the arrow) and the XML structure (right of the arrow) generated when the input is applied to the generator. Each parameter is colored based on how it used by the generator. Parameters used to construct a value returned by the call to `nextByte` on lines 2, 4, 5, and 11 are colored green, yellow, blue, and red, respectively. Unused parameters are colored gray. Whitespace has been added to the XML structures for readability.

When the parameter sequence in Figure 10a is applied to the generator in Listing 4.1, the generator produces the string `"<a><b>x</b><c></c></a>"`. The parameter at index 0 is used to construct the root element's tag name, `'a'`. Next, the parameter at index 1 determines that the root element should have child elements, and the parameter at index 2 determines that there should be two children. The parameter at index 3 determines the first child's tag name, `'b'`. The parameter at index 4 determines that the first child should have text content, and the parameter at index 5 determines that the text content should be `'x'`. The parameter at index 6 determines the second child's tag name, `'c'`. The parameter at index 7 determines that the second child should have children. Finally, the parameter at index 8 determines that the second child should have zero children. Following a similar process, the generator produces the string `"<d><e><f>y</f></e></d>"` from the parameter sequence in Figure 10b.

Notice that the effect of each parameter depends upon how it is used by the generator. For example, the parameter at index 5 of Figure 10a is used by the generator to create the text content `'x'` for the first child of the root element. Whereas, the parameter at index 5 of Figure 10b is used by the generator to determine that the first child of the root element should have one child.

Since a parameter sequence is a bit string, it can be perturbed using generic mutation and crossover operators to create a new child sequence. Regardless of how these operators change the parent, the structure generated from the child sequence will still conform to any constraints imposed by the generator. For instance, the generator in Listing 4.1 will always create opening and closing tags. This allows the parametric fuzzer to produce valid inputs of various types without the need for type-specific operators. However, since the effect of a parameter value depends on the context in which it is used, unmodified parameters in the child sequence may be interpreted differently than they were for the parent sequence. This can limit the effectiveness of traditional crossover operators because naively chosen crossover points are likely to cause the parameters from one parent to be placed into a position in the other parent that corresponds to an entirely different context.

Consider a one-point crossover of the inputs in Figures 10a and 10b that produces the child parameter sequence displayed Figure 10c. This child parameter sequence generates the string `"<a><b>Z</b><L></L></a>"` when applied to the generator in Listing 4.1. Even though the child parameter sequence was constructed from part of the sequence in Figure 10b, the structure generated from the child does not resemble the structure generated for Figure 10b, because the portion of the sequence in Figure 10b that was transferred to the child was interpreted in a different context. For example, the parameter at index 5 of input sequence in Figure 10b, 76, was originally used by the call to `nextByte` on line 4. In this context, the value 76 meant that the element should have children. However, in the child parameter sequence, the parameter value 76 was used by the call to `nextByte` on line 2. In this context, the value 76 meant that the element should have the tag name `'L'` (ASCII character 76).

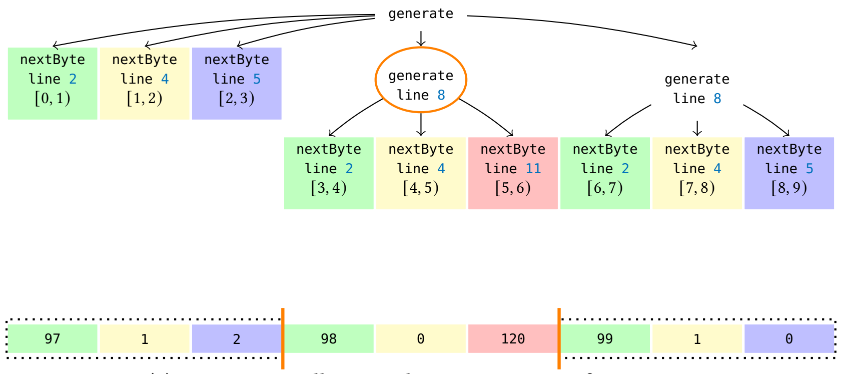
These context changes limit the number of crossover points that produce children that inherit advantageous traits from both parents—negatively impacting the heritability of traditional crossover operators. This effect is even more pronounced as the length of parameter sequences and the complexity of generators increase.

4.3 APPROACH

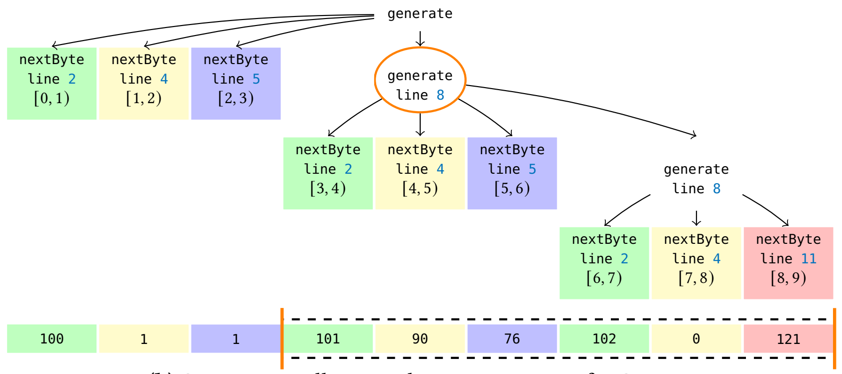
Our approach to crossover in parametric fuzzing, linked crossover, leverages information about the dynamic execution behavior of parametric generators to intelligently select crossover points. Linked crossover aims to identify and exchange portions of parameter sequences that are interpreted similarly by the parametric generator. These subsequences are identified using "parametric call trees". A parametric call tree records caller-callee relationships between method calls and the portion of the parameter sequence that was used by each method call. Linked crossover is a variant of two-point crossover—two crossover points are chosen for each parent and the values between those points are swapped between the parents. Unlike traditional two-point crossover, which chooses crossover points at random (De Jong and Spears, 1991), linked crossover computes crossover points based on the parametric call trees of the parent inputs. This links the choice of crossover points to the parametric generator's execution behavior, thereby preserving logical boundaries in the input and increasing the chance that the crossover produces a child that inherits traits from both parents.

4.3.1 Parametric Call Tree

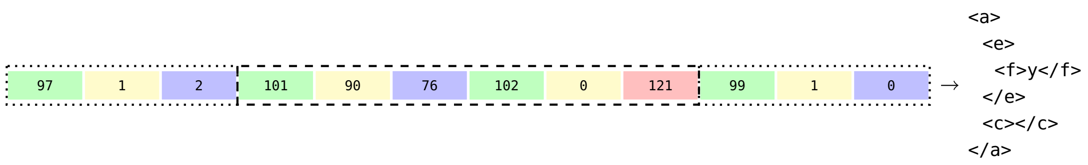
The parametric call tree for a parameter sequence represents an execution of the `generate` method, the method responsible for constructing arguments for the fuzzing target from a parameter sequence using one or more parametric generators, when `generate` is supplied with the parameter sequence. The parametric call tree consists of a set of vertices representing method calls and edges representing caller-callee relationships. Each vertex has exactly one parent, its caller, except the root of the tree, which has no parent and represents the initial call to `generate`. Each vertex v has zero or more child vertices representing method calls made directly by v, i.e., its callees. A vertex v is a descendant of a vertex u if the simple path from the root of the tree to v contains u; every vertex is a descendant of itself. Therefore, a vertex v is a descendant of a vertex u if the method call represented by v was made during the execution of the method call represented by u or $v = u$.



(a) Parametric call tree and crossover points for Parent A.



(b) Parametric call tree and crossover points for Parent B.



```
<a>
  <e>
    <f>y</f>
  </e>
  <c></c>
</a>
```

(c) Child sequence (left) and the XML structure generated when the child is applied to the generator in Listing 4.1 (right).

Figure 11: Parent parametric call trees (top), crossover points (middle), and the child parameter sequence (bottom) for a linked crossover. Parametric call trees are depicted for the generator executions produced when applying the parent parameter sequences in Figures 10a and 10b to the generator in Listing 4.1. Each vertex contains the name of the method called and the line on which it was called. Parameters requests consumed by calls to `nextByte` on lines 2, 4, 5, and 11 are colored green, yellow, blue, and red, respectively. The third line of each parameter request indicates the parameter sequence interval consumed by the request. Arrows represent caller-callee relationships. Below each tree, we show the parent parameter sequence and the crossover points linked to the vertex circled in the tree. Crossover points are marked with solid orange lines. Below the parents, we show the child parameter sequence produced from crossing over the parents at the marked points. The dotted and dashed lines enclose the portions of the child that were transferred from Parents A and B, respectively.

Vertices corresponding to calls to methods provided by the parametric fuzzer that directly consume bytes from the parameter sequence (e.g., the `nextByte` method used in Listing 4.1) are annotated with the interval of the parameter sequence that was consumed by the call. We refer to these annotated vertices as "parameter requests". A parameter request has no children by construction. A method call is represented in the parametric call tree if and only if it is a parameter request or at least one parameter request occurred during the execution of that method call. Therefore, for every vertex v in a parametric call tree, there exists some parameter request u that is a descendant of v.

In theory, the parametric call tree could be defined per thread of execution. However, since parametric generators are single-threaded (regardless of whether the application under test is multi-threaded), parameter requests only occur in a single thread. Thus, we will discuss only a single parametric call tree for each input parameter sequence.

As an example, consider the generator in Listing 4.1 and the parametric input depicted in Figure 10a. The parametric call tree for the execution of `generate` induced by the parameter sequence in Figure 10a is shown in Figure 11a. The root vertex represents the call to `generate`. The leftmost child of the root represents the first call to `nextByte` on line 2. This vertex is a parameter request and is associated with the interval $[0, 1)$ because the first call to `nextByte` consumed the first byte of the parameter sequence. The next two leftmost children of the root vertex represent the calls to `nextByte` on lines 4 and 5. The two rightmost children of the root vertex represent recursive calls to `generate` made on line 8. Each of these two vertices has three children, each representing a call to `nextByte`.

4.3.2 Linked Crossover

Computing Crossover Points

When performing linked crossover, the crossover points for a parent input are computed based on a vertex selected from the parent's parametric call tree using Algorithm 7. The computed crossover points split the parent input before and after the portion of the parameter sequence that was used by the parameter requests made during the execution of the method call represented by the vertex. This preserves boundaries corresponding to method calls in the input increasing the chance that high-fitness subsequences in parent inputs appear in their children.

Algorithm 7 Computing crossover points for a vertex.

```
Input: parametric call tree vertex $v$
Output: a pair of crossover points
1: $D \leftarrow$ the set of parameter requests that are a descendant of $v$
2: $S \leftarrow \{\}$
3: for each $d \in D$ do
4:     $S \leftarrow S \cup$ interval of parameter sequence consumed by $d$
5: end for
6: return $min(S), max(S) + 1$
```

For example, consider the circled vertex in Figure 11a. There are three parameter requests that are a descendant of this vertex. The union of the intervals associated with these requests is $[3, 6)$ corresponding to the crossover points depicted in Figure 11a.

Selecting Vertices

In order to leverage vertex-based crossover points to combine inputs from a population, linked crossover begins by selecting two vertices: a "recipient" and a "donor". Given a parent input, referred to as the "primary" parent, linked crossover begins by selecting a recipient vertex at random from eligible vertices in the primary parent's parametric call tree. A vertex v is eligible to be a recipient if the following is true:

1. There does not exist some vertex u, such that the set of parameter requests descended from v is equal to those descended from u and the method call represented by u happened before the method call represented by v.

2. v is not the root of the parametric call tree.

3. There are at least two different parameter requests that are a descendant of v.

The first criterion is violated if a vertex does not have a sibling. A vertex does not have a sibling if every parameter request that occurred during the caller of the method represented by the vertex occurred during execution of the method represented by the vertex. The first criterion ensures that there is only one eligible vertex corresponding to each distinct pair of crossover points. If two vertices, u and v, have the same set of descendant parameter requests, then the crossover points for v and u are the same. Therefore, when two or more vertices have the same set of descendant parameter requests, only the vertex corresponding to the earliest method call is eligible to be a recipient. The second criterion guards against producing a child that is overly dissimilar to the primary parent: if v is the root of the parametric call tree, then selecting v as the recipient will replace the entire parameter sequence for the primary parent. The third criterion is a heuristic that guards against producing a child that is overly similar to the primary parent: if only one parameter request is a descendant of v, then selecting v as the recipient will replace only a small portion of the parameter sequence for the primary parent. Future work could explore other heuristics for selecting recipient vertices.

Next, linked crossover selects a secondary parent at random from the set of eligible members of the population. A member of the population is eligible to be the secondary parent if its call tree contains at least one eligible vertex. Every vertex that represents a call to the same method as the recipient is eligible to act as the donor vertex, even vertices that do not satisfy the recipient eligibility requirements. This criterion increases the chance that the donated subsequence is interpreted similarly by the generator for the child as it was for the secondary parent. To reduce the performance impact of selecting secondary parents, the fuzzer can maintain a mapping from each method to the set of members of the current population that are eligible to act as the secondary parent for a linked crossover targeting a recipient vertex representing that method. Once a secondary parent is selected, a donor vertex is selected at random from eligible vertices in the secondary parent's parametric call tree.

Application

Algorithm 8 describes how to apply a linked crossover between a primary and secondary input based on a selected recipient vertex and donor vertex. Crossover points are computed based on the recipient and donor vertex, as described above. Standard two-point crossover is then performed replacing the portion of the primary parent's parameter sequence that lies between the crossover points computed for the recipient vertex with the portion of the secondary parent's parameter sequence that lies between the crossover points computed for the donor.

Figure 11 depicts a linked crossover between the two inputs in Figure 10. The recipient vertex is the call to the method `generate` circled in orange in Figure 11a. The donor vertex is the call to the method `generate` circled in orange in Figure 11b. The union of the interval of parameter requests descended from the recipient vertex is $[3, 6)$ and from the donor vertex is $[3, 9)$. These intervals correspond to the crossover points marked with orange lines in Figures 11a and 11b. The portion of the sequence in Figure 11a between the marked crossover points is replaced with the portion of the sequence in Figure 11b between the marked crossover points producing the child sequence depicted in Figure 11c.

Algorithm 8 Applying a linked crossover.

```
Input: primary parent $x \leftarrow \langle x_0, x_1, \ldots, x_{n-1} \rangle$, secondary parent $y \leftarrow \langle y_0, y_1, \ldots, y_{m-1} \rangle$, recipient vertex $r$,
    donor vertex $d$
Output: child sequence
1: $i, j \leftarrow$ crossover points computed for $r$ using Algorithm 7
2: $k, l \leftarrow$ crossover points computed for $d$ using Algorithm 7
3: return $\langle x_0, x_1, \ldots, x_{i-1} \rangle + \langle y_k, y_{k+1}, \ldots, y_{l-1} \rangle + \langle x_j, x_{j+1}, \ldots, x_{n-1} \rangle$
```

When applying multiple linked crossover operations to the same primary parent input additional considerations must be made because an operation may shift subsequent portions of the input if the size of the donated subsequence is not equal to the size of the replaced subsequence. Additionally, if two operations impact non-disjoint intervals of the primary parent's parameter sequence, only one of the operations can be applied because the same parameter should only be replaced once.

When applying multiple linked crossover operations on the same primary parent, begin by sorting the operations into non-increasing order by the start of the interval they impact (as determined by the recipient vertex of the application). Then, process each operation in order. If an operation targets an interval that is non-disjoint with an interval targeted by an operation that has already been applied, skip it. Otherwise, apply the operation as normal replacing the portion of the primary parent within the interval targeted by the operation with the operation's donated subsequence.

Applying the linked crossover operations in non-increasing order by the start of the interval they impact ensures that parameters in positions before the start of the interval targeted by the last operation applied remain in their original positions. When a crossover operation is about to be applied, the interval that it targets must be disjoint with the interval targeted by the last operation applied, otherwise the operation would have been skipped. Furthermore, the start of the interval targeted by the operation cannot be greater than the start of the interval targeted by the last operation applied. Therefore, when an operation is applied, the interval that it targets must start and end before the start of the interval targeted by the last operation applied. Thus, the operation can be applied normally without adjusting the replacement interval.

4.4 IMPLEMENTATION

Although we believe that linked crossover is suitable for many languages, we implemented linked crossover as part of ZEUGMA, a new parametric fuzzer for Java. For the sake of simplicity, we chose to use branch coverage feedback for ZEUGMA. However, linked crossover could be used with other forms of feedback such as Padhye et al. (2019b)'s input-validity feedback. ZEUGMA collects branch coverage and method call information using the ASM instrumentation and analysis framework to rewrite Java bytecode (OW2 Consortium, 2024). Like JQF (Padhye et al., 2019a) (the parametric fuzzing framework used to create ZEST, BEDIVFUZZ, and RLCHECK), ZEUGMA is implemented on top of JUNIT-QUICKCHECK (Holser, 2023), a property-testing library inspired by QUICKCHECK (Claessen and Hughes, 2000). JUNIT-QUICKCHECK leverages user-defined generators to create random test inputs. These generators use a high-level API provided by JUNIT-QUICKCHECK to create arbitrary values of common types. ZEUGMA integrates into JUNIT-QUICKCHECK by using fuzzer-derived parameter sequences to determine these arbitrary values.

4.4.1 Updating the Population

ZEUGMA implements the generic fuzzing algorithm described in Section 4.2.1. Branch coverage feedback is used to determine which inputs should be included in the population. For each branch that has been covered by at least one input, ZEUGMA tracks the shortest input that covered that branch. The set of tracked inputs form the population.

4.4.2 Modifying Inputs

When creating a child input, ZEUGMA selects the primary parent from the population at random. Then, ZEUGMA chooses the total number of mutation and crossover operations to apply to the primary parent. This number is chosen from a shifted geometric distribution with a success probability of 0.25 (corresponding to an expected value of four). This value can be fine-tuned; preliminary experiments that we conducted suggested that a probability of 0.25 was an effective choice across all subjects. ZEUGMA uses the same replacement-based mutation operator described by Padhye et al. (2019b) with a mean mutation length of eight. The mean mutation length can be fine-tuned; preliminary experiments that we conducted suggested that eight was an effective choice across all subjects.

ZEUGMA can be configured to use mutation only, to use traditional one-point crossover, to use traditional two-point crossover, or to use our novel linked crossover. We describe our evaluation of these options in Section 4.6. For one- and two-point crossover, the second parent is selected at random from the population, and crossover points are selected at random. Linked crossover is performed as described in Section 4.3.2.

If ZEUGMA has been configured to use mutation only, then all the operations are mutation. Otherwise, ZEUGMA chooses between mutation and crossover at random with an equal likelihood of selecting either option. If ZEUGMA has been configured to use linked crossover, then all linked crossover operations are applied first in the manner described in Section 4.3.2, then the mutation operations are applied. Otherwise, operations are applied in the order they are selected.

4.4.3 Building the Parametric Call Tree

ZEUGMA uses bytecode instrumentation to build parametric call trees allowing linked crossover to work on unmodified JUNIT-QUICKCHECK generators. ZEUGMA adds code at the start of methods that records that the method was entered and before method returns that records that the method was exited. ZEUGMA's integration with JUNIT-QUICKCHECK records when a portion of the parameter sequence is consumed. These recorded messages are ignored by ZEUGMA unless it is actively building a parametric call tree.

Parametric call trees are only needed when ZEUGMA is configured to use linked crossover and for inputs that will be saved to the population. Hence, before ZEUGMA saves an input to the population, it re-executes the `generate` method with the input and observes messages recorded about method entries, method exits, and parameter consumptions. A stack is used to track the call stack. When a method is entered, a new vertex is created for the method call and pushed onto the stack. When a parameter is consumed, the index of that parameter is then associated with the vertex at the top of the stack and the top vertex is marked as a parameter request. When a method is exited, the top vertex is popped off of the stack. If the popped vertex is not associated with the index of at least one parameter, and it has no children, then the vertex is not a parameter request and no parameter requests occurred during the execution of the method call represented by the vertex. As noted in Section 4.3.1, a method call is included in the parametric call tree only if it is a parameter request or at least one parameter request occurred during the execution of that method call.

Therefore, the popped vertex is discarded. If the popped vertex is not discarded, then the vertex on the top of the stack is marked as the parent of the popped vertex. If there is no vertex on the top of the stack, i.e, the stack is empty, then the popped vertex is recorded as the root of the tree.

4.5 LIMITATIONS

Linked crossover is a heuristic approach; its efficacy is dependent on the structure of the generators. If the generators are not split into methods or the methods do not correspond to logical boundaries, then linked crossover will be ineffective. This limitation only applies to the structure of the generators and not the entire system under test. However, we do not believe that this is a significant limitation, as best practices for writing QuickCheck-style generators rely on composition. Claessen and Hughes (2000) explain that combinators can be used to combine simple generators into complex generators. This compositional style results in method calls that are responsible for creating a single subcomponent, and, therefore, likely correspond to reasonable logical boundaries. Linked crossover's dependency on methods could be mitigated by using additional dynamic execution information or techniques such as method call sites, dynamic slices, and dynamic information flows. This is an interesting direction for future research.

4.6 EVALUATION

For our evaluation, we examined linked crossover's impact on overall fuzzer performance and its ability to produce children that preserve desirable traits from their parents—a property referred to by Raidl and Gottlieb (2005) as heritability. We created two novel heritability metrics for evolutionary fuzzing, hybrid proportion and inheritance rate, which we describe in Section 4.6.1. We also examined the impact of linked crossover on overall fuzzer performance using traditional metrics. Our evaluation of linked crossover focused on answering the following research questions:

RQ1: How does linked crossover compare to other crossover operators with respect to heritability?

RQ2: How does Zeugma with linked crossover's ability to discover coverage-revealing inputs compare to state-of-the-art parametric fuzzers?

RQ3: How does Zeugma with linked crossover's ability to detect defects compare to state-of-the-art parametric fuzzers?

4.6.1 Methodology

We evaluated Zeugma on benchmark suite of seven real-world Java projects consisting of the five subjects used by Padhye et al. (2019b) in their evaluation of Zest (Ant, BCEL, Closure, Maven, and Rhino) and the two additional subjects used by Nguyen and Grunske (2022) in their evaluation of BeDivFuzz (Nashorn and Tomcat). We list these subjects in Table 6. We used the latest stable release of each subject available in the Maven Central Repository. Minor modifications were made to the fuzzing targets used by Padhye et al. (2019b) and Nguyen and Grunske (2022) to ensure compatibility with the newer subject versions. We used the junit-quickcheck generators included with JQF (version 2.0) (Rohan Padhye and JQF Contributors, 2023) for XML, JavaScript, and Java classes. We changed the configuration for the XML generator to increase the maximum depth of generated XML trees to ten as Kukucka et al. (2022) found that deeper trees were necessary to exercise certain functionality in Maven. No changes were made to the generator itself.

Table 6: Evaluation Subjects. For each subject, we list the project name and version (Project), the format of the input (Format), and the number of branches as reported by JaCoCo (Branches).

Project		Format	Branches
Apache Ant (1.10.13)	[9]	XML	24,626
Apache BCEL (6.7.0)	[12]	Java class	5,975
Google Closure (v20230502)	[50]	JavaScript	129,376
Apache Maven (3.9.2)	[14]	XML	14,886
OpenJDK Nashorn (11.0.19)	[164]	JavaScript	28,191
Mozilla Rhino (1.7.14)	[126]	JavaScript	26,690
Apache Tomcat (10.1.9)	[15]	XML	39,020

In order to compare linked crossover against other crossover operators, we created four variants of Zeugma. The first variant, Zeugma-X, does not use crossover at all. The other variants, Zeugma-Link, Zeugma-1PT, and Zeugma-2PT, use linked, one-point and, two-point crossover, respectively. These variants differ from each other only with respect to the application of crossover as described in Section 4.4.

RQ1: Heritability

In an evolutionary search, an effective crossover operator produces children that preserve desirable traits from their parents—a property referred to by Raidl and Gottlieb (2005) as heritability. For an evolutionary fuzzer, the primary trait of interest for an input is its ability to cover program features (typically branches or statements). We propose two coverage-based heritability metrics for evolutionary fuzzing: inheritance rate and hybrid proportion. Inheritance rate considers the percentage of program features covered by at least one of an input's parents that were also covered by the input for a typical input produced by a crossover operator. Hybrid proportion measures the likelihood that a crossover operator produces a child that covers at least one feature exclusively covered by each of its parents.

Inheritance rate and hybrid proportion aim to measure commonalities between a child and its parents—they do not try to measure whether the child covers new program features. Although additional coverage is generally positive in fuzzing, it is not necessarily indicative of a high-quality crossover. The additional coverage could represent an undesirable deviation from the parents' behavior, or it could be a positive effect of combining parts of the parents' behavior. Therefore, program features covered by the child but not its parents are neither penalized nor rewarded when computing inheritance rates and hybrid proportions.

Consider a child c produced by applying a crossover operator to parents p_1 and p_2. Let X be a set of "common" features—program features that are covered by a high percentage of random inputs. Common features are excluded when computing inheritance rate and hybrid proportion because covering a common feature does not necessarily represent a unique, desirable property of a particular input. Let P_1, P_2, and C be the set of program features not in X covered by p_1, p_2, and c, respectively. We define the *inheritance rate*, denoted IR, of a crossover as the percentage of program features covered by either parent that were also covered by the child:

$$\mathrm{IR}(P_1, P_2, C) = \frac{|(P_1 \cup P_2) \cap C|}{|P_1 \cup P_2|}$$

We say that a crossover is a *hybrid*, denoted HY, if the child covers at least one feature that is covered by the first parent but not the second parent, and the child covers at least one feature that is covered by the second parent but not the first parent:

$$\begin{aligned} \mathrm{HY}(P_1, P_2, C) = {} & (\exists x \in C : x \in P_1 \land x \notin P_2) \\ & \land (\exists y \in C : y \in P_2 \land y \notin P_1) \end{aligned}$$

These metrics are extended to the operator itself by considering the distribution of inheritance rates and the proportion of children that are hybrids for some sample of parents.

In order to collect a representative sample of parent inputs, we performed twenty fuzzing campaigns using Zeugma-X on the subjects listed in Table 6 and recorded the state of the corpus after five minutes. We chose to use the state of the corpus after five minutes because we found that, on average, over half of the inputs saved to corpus after three hours were saved in the first five minutes (mean = 57.8%, median = 61.8%, minimum = 29.8%, maximum = 78.2%). We collected branch coverage using Zeugma's instrumentation, and considered all coverage including system classes. To ensure that non-repeatable coverage due to class loading did not impact the results, if a class was loaded during the execution of an input, the input was re-executed, and the coverage recording from the second execution was used. We identified the set of common features, X, by executing one thousand random inputs for each subject. Any branch that was covered by a majority of these random inputs was marked as common.

To compute the heritability metrics, we sampled one thousand pairs of parents for each subject. Each sample was selected by choosing two parents at random from a randomly selected corpus produced for the subject. Samples were re-selected until both parents covered at least one branch not in the common feature set and not covered by the other parent to ensure that it was possible to produce a hybrid child from the pairing. For each sample, we produced one child for each of the three crossover operators—linked, one point and two point—and recorded whether the child was a hybrid and its inheritance rate.

RQ2 and RQ3: Coverage and Defects

Our second and third research questions evaluate the impact of linked crossover on branch coverage and defect detection ability. In addition to the three variants of Zeugma without linked crossover, we also compared our approach against Zest (Padhye et al., 2019b), BeDivFuzz (Nguyen and Grunske, 2022), and RLCheck (Reddy et al., 2020). We used the latest releases at time of writing of Zest (version 2.0) and BeDivFuzz (commit c06eaca) which include improvements to the coverage instrumentation. For BeDivFuzz, we evaluated both of the configurations described by Nguyen and Grunske (2022): BeDiv-Struct and BeDiv-Simple. Because our reported values are based only on saved inputs, we modified BeDivFuzz to ensure that all inputs that reveal new coverage are saved—not just "valid" inputs. For the same reason, we choose to use the "grey-box" version of RLCheck because the grey-box version saves coverage-revealing inputs to a corpus. We did not compare Zeugma against Confetti (Kukucka et al., 2022) because Confetti only supports Java version 8, and the latest release of Zest requires Java version 9 or greater.

BeDivFuzz and RLCheck cannot use junit-quickcheck generators out of the box; they require the generators to be manually modified. For the XML and JavaScript generators, we used the generators created by Nguyen and Grunske (2022) and Reddy et al. (2020) to evaluate BeDivFuzz and RLCheck, respectively. Neither Nguyen and Grunske (2022)'s evaluation of BeDivFuzz nor Reddy et al. (2020) evaluation of RLCheck included a Java class generator. Therefore, we created a modified version of the Java class generator included with JQF for BeDivFuzz. Unfortunately, the documentation for RLCheck did not provide sufficient detail for us to create a modified Java class generator for RLCheck; therefore, we do not include results for RLCheck on BCEL.

Following best practices suggested by Metzman et al. (2021), we used an independent code coverage metric—branch coverage collected with JaCoCo (version 0.8.7) (Mountainminds GmbH & Co. KG and Contributors, 2021). In order to calculate branch coverage, we reran inputs saved by the fuzzer in a JaCoCo-instrumented JVM after the campaign finished. Coverage was measured only in application classes (those found in the Java archive (JAR) files associated with the subject). For Nashorn, we further limited coverage to only include classes related to Nashorn, those with the package prefix `jdk.nashorn`. Nashorn is part of the Java Class Library (JCL) and including all JCL coverage would bias results in favor of fuzzers that heavily depend on parts the JCL (e.g., `java.lang.String` and `java.util.HashMap`) inflating their coverage.

In order to measure defect detection ability, we collected the failures detected for each campaign by rerunning inputs saved by the fuzzer in a new JVM after the campaign finished. If a saved input induced a failure, we recorded the type (e.g., `java.lang.RuntimeException`) and stack trace of the failure induced by the input. Failures with the same type and top five stack frames were initially marked as the same failure. We then manually inspected the set of distinct failures to map the failures to a set of unique defects. All the identified defects were reported and confirmed by a developer for the associated project.

Twenty trials were conducted for each fuzzer on each subject in accordance with current best practices (Klees et al., 2018). Each campaign was performed on its own virtual machine with four 2.6 GHz AMD EPYC 7H12 vCPUs, with sixteen GB of RAM, running Ubuntu 20.04.3, and using the Oracle Java Development Kit (JDK) version 11.0.19. No seeds were provided for any subject. The original dictionaries created by Padhye et al. (2019b) and Nguyen and Grunske (2022) were used for the XML subjects. Similar to the evaluation performed by Padhye et al. (2019b) and Kukucka et al. (2022), generators were limited to producing inputs that used 10,240 or less raw input bytes. . Because parametric fuzzing is typically used for property-based testing, the effectiveness a parametric fuzzer on relatively short campaigns is of particular interest (Reddy et al., 2020; Nguyen and Grunske, 2022). However, longer campaigns may be more indicative of general performance trends (Klees et al., 2018). Therefore, we chose to evaluate the effectiveness of linked crossover on both short (five minute) campaigns like Reddy et al. (2020) and long (three hour) campaigns like Padhye et al. (2019b).

4.6.2 Results

RQ1: Heritability

Table 7: Heritability Metrics. For each crossover operator, we report the proportion of samples that were hybrids (HY) and the median inheritance rate (IR) on each subject. The largest value for each metric on each subject is highlighted in blue. Values that differ significantly from that of linked crossover are colored red.

	Linked		One Point		Two Point	
Subject	HY	IR	HY	IR	HY	IR
Ant	0.561	0.923	0.459	0.124	0.493	0.069
BCEL	0.283	0.512	0.660	0.347	0.756	0.286
Closure	0.742	0.717	0.661	0.101	0.712	0.094
Maven	0.446	0.589	0.404	0.497	0.399	0.453
Nashorn	0.622	0.646	0.548	0.117	0.591	0.132
Rhino	0.611	0.502	0.599	0.263	0.643	0.255
Tomcat	0.322	0.775	0.350	0.276	0.328	0.279

Link	0.633	0.667
$8.18 \cdot 10^{-25}$	1PT	0.534
$3.45 \cdot 10^{-38}$	$8.94 \cdot 10^{-3}$	2PT

Ant

Link	0.643	0.689
$1.38 \cdot 10^{-28}$	1PT	0.551
$1.43 \cdot 10^{-48}$	$7.00 \cdot 10^{-5}$	2PT

BCEL

Link	0.659	0.684
$7.15 \cdot 10^{-35}$	1PT	0.522
$2.91 \cdot 10^{-46}$	$9.42 \cdot 10^{-2}$	2PT

Closure

Link	0.584	0.616
$8.11 \cdot 10^{-11}$	1PT	0.528
$2.95 \cdot 10^{-19}$	$3.30 \cdot 10^{-2}$	2PT

Maven

Link	0.644	0.643
$6.11 \cdot 10^{-29}$	1PT	0.501
$1.57 \cdot 10^{-28}$	$9.33 \cdot 10^{-1}$	2PT

Nashorn

Link	0.624	0.639
$1.06 \cdot 10^{-21}$	1PT	0.517
$3.50 \cdot 10^{-27}$	$1.79 \cdot 10^{-1}$	2PT

Rhino

Link	0.670	0.677
$1.23 \cdot 10^{-39}$	1PT	0.501
$1.41 \cdot 10^{-42}$	$9.31 \cdot 10^{-1}$	2PT

Tomcat

(a) Pairwise Inheritance Rates. Mann-Whitney U test two-tailed p-values are placed in the entries below the main diagonal. Vargha-Delaney $\hat{A}_{12}$ effect sizes are placed in the entries above the main diagonal.

Link	1.506	1.314
$6.14 \cdot 10^{-6}$	1PT	1.146
$2.68 \cdot 10^{-3}$	$1.40 \cdot 10^{-1}$	2PT

Ant

Link	4.918	7.850
$3.21 \cdot 10^{-65}$	1PT	1.596
$4.74 \cdot 10^{-103}$	$2.88 \cdot 10^{-6}$	2PT

BCEL

Link	1.475	1.163
$9.10 \cdot 10^{-5}$	1PT	1.268
$1.45 \cdot 10^{-1}$	$1.59 \cdot 10^{-2}$	2PT

Closure

Link	1.188	1.213
$6.36 \cdot 10^{-2}$	1PT	1.021
$3.73 \cdot 10^{-2}$	$8.55 \cdot 10^{-1}$	2PT

Maven

Link	1.357	1.139
$9.18 \cdot 10^{-4}$	1PT	1.192
$1.70 \cdot 10^{-1}$	$5.78 \cdot 10^{-2}$	2PT

Nashorn

Link	1.051	1.147
$6.15 \cdot 10^{-1}$	1PT	1.206
$1.52 \cdot 10^{-1}$	$4.75 \cdot 10^{-2}$	2PT

Rhino

Link	1.134	1.028
$2.01 \cdot 10^{-1}$	1PT	1.103
$8.11 \cdot 10^{-1}$	$3.21 \cdot 10^{-1}$	2PT

Tomcat

(b) Pairwise Hybrid Proportions. Fisher's exact test two-tailed p-values are placed in the entries below the main diagonal. Odds ratios are placed in the entries above the main diagonal. Each reported odds ratio is the ratio of the higher odds to the lower odds for the compared pair.

Figure 12: Pairwise comparisons of heritability metrics of for one-point (1PT), two-point (2PT), and linked (Link) crossover. The main diagonal of each matrix consists of the crossover operators. Each matrix entry not on the main diagonal at some row r and column c compares the crossover operator in row r to the crossover operator in column c. Statistically significant p-values are colored blue. Larger effect sizes are colored darker shades of red.

Table 7 summarizes the results of our heritability experiment. We performed pairwise comparisons of the inheritance rates and hybrid proportions measured on each subject for the different crossover operators using two-tailed Mann-Whitney U tests and Fisher's exact tests, respectively. Following current best practices as described by Arcuri and Briand (2014), a base significance level of 0.05 was adjusted for three comparisons resulting in a Bonferroni-corrected significance level of $\frac{0.05}{3} = 0.0167$ per test. Inheritance rates for linked crossover were statistically significantly greater than for one- and two-point crossover on all subjects. Linked crossover had a significantly greater hybrid proportion compared to one-point crossover on Ant, Closure, and Nashorn; and compared to two-point crossover on Ant. However, linked crossover had a significantly lower hybrid proportion compared to one- and two-crossover on BCEL. All other differences between linked crossover and the other operators were not significant. Full results of the pairwise comparisons are shown in Figure 12. Overall, linked crossover produced children that inherited more desirable traits from their parents (i.e., covered a higher percentage of the branches covered by their parents) while still combining traits from both parents.

RQ2: Coverage

Figure 13 and Table 8 summarize the results of our coverage experiment. Figure 13 shows median, minimum, and maximum branch coverage across the twenty trials over time for each fuzzer on each subject. Table 8 shows median coverage values for each fuzzer after five minutes (short duration) and three hours (long duration). On each subject, we performed pairwise comparisons of the branch coverage for short and long campaigns for the different fuzzers using Mann-Whitney U tests. A base significance level of 0.05 was adjusted for twenty-eight comparisons resulting in a Bonferroni-corrected significance level of $\frac{0.05}{28} = 0.00179$ per test except on BCEL. For BCEL, the significance level was adjusted for twenty-one comparisons (due to the exclusion of RLCHECK) resulting in a corrected significance level $\frac{0.05}{21} \approx 0.00238$. Statistically significant differences between the branch coverage of ZEUGMA-LINK and the other fuzzers are colored red in Table 8.

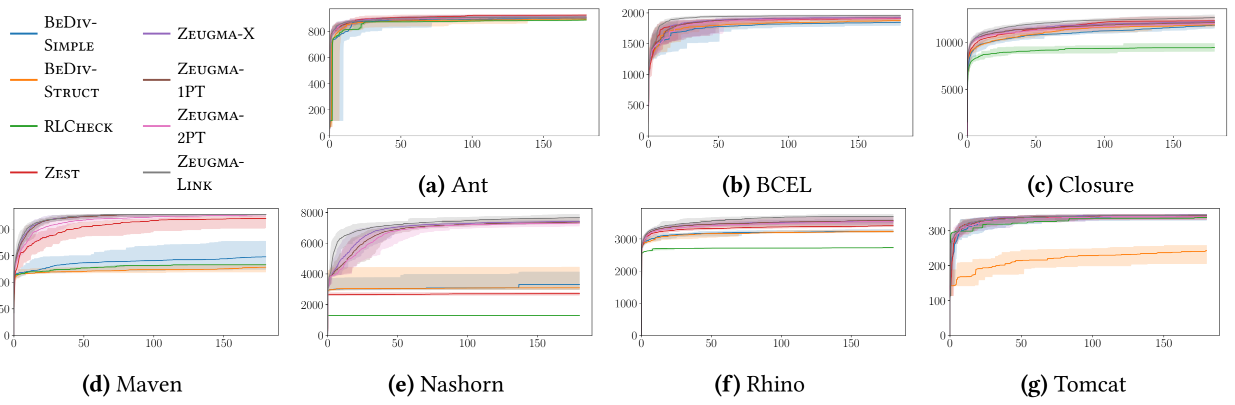


Figure 13: Branch Coverage Over Time. Each x-axis is time in minutes and each y-axis is the number of covered branches. Each plot depicts the median number of covered branches (solid line) and the range of covered branches (filled area) across the twenty trials over the duration of the fuzzing campaign for each of the fuzzers.

Table 8: Branch Coverage. For each fuzzer, we report the median branch coverage in application classes for each subject across twenty fuzzing campaigns after five minutes (5M) and three hours (3H). The largest median or medians (in the case of a tie) for each time and subject is highlighted in blue. Branch coverage values that differ significantly from Zeugma-Link's are colored red.

Subject / Fuzzer	Ant 5M	Ant 3H	BCEL 5M	BCEL 3H	Closure 5M	Closure 3H	Maven 5M	Maven 3H	Nashorn 5M	Nashorn 3H	Rhino 5M	Rhino 3H	Tomcat 5M	Tomcat 3H
BeDiv-Simple	755.0	899.0	1435.5	1846.5	9328.0	11863.5	590.5	738.0	3008.5	3319.5	2952.0	3235.5	274.5	341.0
BeDiv-Struct	786.5	896.5	1412.5	1876.5	9336.5	11904.5	578.5	641.5	2993.5	3092.0	2915.5	3237.0	161.0	242.5
RLCheck	769.0	889.0	—	—	8262.5	9480.5	579.0	663.0	1298.0	1298.0	2627.0	2730.0	299.0	338.0
Zest	820.0	927.0	1516.5	1909.5	9782.5	12352.0	778.5	1098.5	2654.5	2717.0	3108.5	3408.0	295.0	340.0
Zeugma-X	835.5	911.0	1480.5	1927.0	10274.5	12251.0	873.0	1138.0	4259.0	7411.0	3169.0	3551.5	297.5	345.0
Zeugma-1PT	828.0	909.5	1489.0	1917.0	10237.5	12153.5	855.5	1138.0	4166.0	7369.5	3169.0	3572.5	294.0	344.0
Zeugma-2PT	819.5	910.0	1472.0	1915.0	10284.0	12038.0	797.0	1134.0	3962.5	7310.0	3143.0	3549.0	291.0	341.5
Zeugma-Link	845.5	911.5	1541.5	1959.0	10395.5	12709.0	906.0	1138.0	5568.5	7654.0	3233.5	3703.0	295.5	345.0

We also used the Vargha-Delaney $\hat{A}_{12}$ statistic (Vargha and Delaney, 2000) to quantify effect sizes for these comparisons. Full results of these test are shown in Figure 14.

Zeugma-Link had the highest median coverage on all subjects except Tomcat after five minutes and on all subjects except Ant after three hours. On the three JavaScript subjects (Closure, Nashorn, and Rhino), Zeugma-Link's branch coverage was significantly greater than that of all other fuzzers on both long and short campaigns, except Zeugma-X on five-minute Closure campaigns. On the only Java class subject (BCEL), Zeugma-Link outperformed BeDiv-Simple, BeDiv-Struct, RLCheck, and Zeugma-2PT on long and short campaigns. However, Zeugma-Link only performed significantly better than Zest, Zeugma, and Zeugma-1PT on long BCEL campaigns. We carefully examined this fuzzing target and believe that, in order to achieve further improvements in coverage, the generator should be improved to be more likely to generate valid Java method bodies. Results on the three XML subjects (Ant, Maven, and Tomcat) were more mixed. As depicted in Figure 13, coverage for most of the fuzzers plateaued on these three subjects. Our analysis of these fuzzing targets revealed that limited coverage is reachable from the drivers for these subjects. Zeugma-Link performed as well as or better than the other fuzzers on all the XML subjects except on long Ant campaigns where Zeugma-Link's branch coverage was significantly less than that of Zest. For all subjects, we found that the effect size was large ($\hat{A}_{12} \geq 0.71$) for all comparisons in which the performance of a baseline fuzzer differed significantly from that of Zeugma-Link.

BeDivFuzz, Zest, and RLCheck performed notably poorly on Nashorn. These fuzzers are all built using JQF which does not add and cannot be configured to add coverage instrumentation to classes with the package prefix `jdk`. Since the classes related to Nashorn are found in `jdk.nashorn`, the JQF-based fuzzers did not receive critical coverage feedback for Nashorn. Given that our primary goal is to evaluate the efficacy of linked crossover (as opposed to comparing all variants of Zeugma against BeDivFuzz, Zest, and RLCheck), we did not find it necessary to make the invasive changes to JQF necessary to collect coverage in these packages.

In general, linked crossover was demonstrably effective at discovering coverage-revealing inputs in both long and short campaigns. It consistently performed as well as or better than other forms of crossover and against state-of-the-art parametric fuzzers. Linked crossover was generally more effective on subjects using the JavaScript or Java class generator, possibly indicating that its efficacy may be impacted by either the nature of the generator or the input type itself. However, this could also be a product of driver limitations for the XML fuzzing targets. Future work may study how to measure and improve the quality of fuzzing targets.

RQ3: Defects

Table 9: Defect Detection Rates. For each fuzzer, we report the defect detection rate of each discovered defect across twenty fuzzing campaigns after five minutes (5M) and three hours (3H). The largest detection rate or rates (in the case of a tie) for each time and defect is highlighted in blue. Detection rates that differ significantly from Zeugma-Link's are colored red.

Defect / Fuzzer	B0 [10]		B1 [11]		C0 [48]		C1 [49]		N0 [155]		N1 [153]		N2 [154]		R0 [121]		R1 [120]		R2 [118]		R3 [119]		R4 [125]	
	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H
BeDiv-Simple	0.00	0.65	0.00	0.00	0.00	0.05	0.15	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.45	0.00	0.00	1.00	1.00	0.00	0.00
BeDiv-Struct	0.05	0.70	0.00	0.00	0.00	0.15	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.20	0.65	0.00	0.00	1.00	1.00	0.00	0.00
RLCheck	—	—	—	—	0.00	0.10	0.20	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00
Zest	0.00	1.00	0.00	0.00	0.00	0.40	0.10	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.10	0.05	0.95	0.00	0.20	1.00	1.00	0.00	0.05
Zeugma-X	0.05	1.00	0.00	0.00	0.05	0.25	0.60	1.00	0.00	0.75	0.00	0.00	0.00	0.05	0.00	0.75	0.15	0.80	0.00	0.95	1.00	1.00	0.00	0.90
Zeugma-1PT	0.00	1.00	0.00	0.10	0.00	0.40	0.50	1.00	0.00	0.30	0.00	0.00	0.00	0.00	0.00	0.70	0.10	0.75	0.00	0.85	1.00	1.00	0.00	0.85
Zeugma-2PT	0.25	1.00	0.00	0.10	0.00	0.40	0.80	1.00	0.00	0.45	0.00	0.05	0.00	0.05	0.00	0.60	0.10	0.90	0.00	0.70	1.00	1.00	0.00	0.85
Zeugma-Link	0.25	1.00	0.00	0.00	0.00	0.65	0.60	1.00	0.05	1.00	0.00	0.00	0.00	0.50	0.00	0.80	0.10	0.70	0.00	0.85	1.00	1.00	0.00	0.95

Across all the campaigns, a total of twelve unique defects were detected: two in BCEL (B0–B1), two in Closure (C0–C1), three in Nashorn (N0–N2), and five in Rhino (R0–R4). Table 9 lists the percentage of campaigns for each fuzzer in which each of these defects was discovered (the detection rate) within the first five minutes and after the full three hours of the campaign. For each defect, we performed pairwise comparisons of the detection rate for short and long campaigns for the different fuzzers using Fisher's exact tests with the same Bonferroni-adjusted significance levels used in Section 4.6.2. Statistically significant differences between Zeugma-Link and other fuzzers are colored red in Table 9. Full results of these tests are shown in Figures 15 and 16.

Zeugma-Link had the highest detection rate for nine of the twelve defects within the first five minutes (short campaigns) and eight out of the twelve defects after the full three hours (long campaigns) of the campaign. This suggests that linked crossover may positively impact a fuzzer's ability to detect defects. For the short campaigns, the Fisher's exact tests indicated that most of the differences in detection rate between the fuzzers were not significant. For long campaigns, Zeugma-Link's detection rate for five defects (N0, N2, R0, R2, R4) was significantly higher than that of BeDiv-Simple, BeDiv-Struct, RLCheck, and Zest.

Interestingly, Zeugma-Link's detection rate never differed significantly from that of Zeugma-X, although, in some cases, it was superior to that of Zeugma-1PT and Zeugma-2PT.

4.6.3 Threats to Validity

We evaluated linked crossover on only seven subjects. These subject may not be representative of all programs. However, these subjects are all mature, well-established projects. Furthermore, we included all the Java subjects evaluated by Padhye et al. (2019b), Reddy et al. (2020), Kukucka et al. (2022), or Nguyen and Grunske (2022).

Our evaluation featured generators for only three different input types: JavaScript, XML, and Java class. We did not evaluate the impact of generator quality or style on the effectiveness of linked crossover. Generator quality is likely to impact any generator-based technique—not just linked crossover. To avoid potential bias, we used the generators included with JQF without modification.

We used branch coverage as a metric for evaluating fuzzer effectiveness in Section 4.6.2. Coverage is only weakly correlated with defect detection ability (Inozemtseva and Holmes, 2014). However, as noted by Metzman et al. (2021) the sparsity of bugs in programs makes it difficult to evaluate a fuzzer by analyzing detected defects alone. Therefore, we analyzed both branch coverage and defect detection rate.

4.7 CONCLUSION

This work demonstrates that crossover point selection can have a significant impact on overall fuzzer performance and that dynamic execution information can be effectively used to inform the selection of crossover points in evolutionary fuzzing. Linked crossover, our approach for using dynamic execution information to select crossover points, produced children that inherited more desirable traits from their parents than traditional one- and two-point crossover. Our evaluation of linked crossover's impact on fuzzer performance found that linked crossover was effective at discovering coverage-revealing inputs and defects in both long and short campaigns. Based on these results, we believe that linked crossover could potentially be adapted for use in unstructured fuzzing in cases where the input is read in a stream-like or piecewise manner by the application.

4.8 STATUS

This work was published in the *Proceedings of the ACM/IEEE 46th International Conference on Software Engineering* (ICSE 2024) (Hough and Bell, 2024a). It was presented at ICSE 2024. The associated artifact for this work was evaluated and awarded the "Available" and "Reusable" badges (Hough and Bell, 2024b).

Figure 14: Pairwise Branch Coverage. Pairwise comparisons of branch coverage per subject on long (three hour) and short (five minute) campaigns for BeDiv-Simple (BeDiv-Simple), BeDiv-Struct (BeDiv-Struct), RLCheck, Zest, Zeugma-X, Zeugma-1PT, Zeugma-2PT, and Zeugma-Link. The main diagonal of each matrix consists of the fuzzers. Each matrix entry not on the main diagonal at some row r and column c compares the fuzzer in row r to the fuzzer in column c. Mann-Whitney U test two-tailed p-values are placed in the entries below the main diagonal. Vargha-Delaney $\hat{A}_{12}$ effect sizes are placed in the entries above the main diagonal. Statistically significant p-values are colored blue. Larger effect sizes are colored darker shades of red.

Ant After 5 Minutes

BD-Simple	0.718	0.627	0.909	0.927	0.914	0.965	0.949
$1.93 \cdot 10^{-2}$	BD-Struct	0.900	0.858	0.863	0.869	0.907	0.896
$1.69 \cdot 10^{-1}$	$1.43 \cdot 10^{-5}$	RLCheck	1.000	1.000	1.000	1.000	1.000
$1.02 \cdot 10^{-5}$	$1.14 \cdot 10^{-4}$	$5.60 \cdot 10^{-8}$	Zest	0.559	0.550	0.756	0.698
$3.96 \cdot 10^{-6}$	$9.24 \cdot 10^{-5}$	$5.68 \cdot 10^{-8}$	$5.33 \cdot 10^{-1}$	Zeugma-1PT	0.530	0.748	0.656
$8.04 \cdot 10^{-6}$	$6.97 \cdot 10^{-5}$	$5.68 \cdot 10^{-8}$	$5.97 \cdot 10^{-1}$	$7.56 \cdot 10^{-1}$	Zeugma-2PT	0.774	0.696
$5.23 \cdot 10^{-7}$	$1.10 \cdot 10^{-5}$	$5.70 \cdot 10^{-8}$	$5.75 \cdot 10^{-3}$	$7.69 \cdot 10^{-3}$	$3.18 \cdot 10^{-3}$	Zeugma-Link	0.603
$1.28 \cdot 10^{-6}$	$1.91 \cdot 10^{-5}$	$5.68 \cdot 10^{-8}$	$3.36 \cdot 10^{-2}$	$9.32 \cdot 10^{-2}$	$3.47 \cdot 10^{-2}$	$2.73 \cdot 10^{-1}$	Zeugma-X

Ant After 3 Hours

BD-Simple	0.704	0.996	0.968	0.764	0.818	0.823	0.834
$2.80 \cdot 10^{-2}$	BD-Struct	0.912	0.998	0.912	0.931	0.939	0.934
$6.11 \cdot 10^{-8}$	$6.70 \cdot 10^{-6}$	RLCheck	1.000	1.000	1.000	1.000	1.000
$4.06 \cdot 10^{-7}$	$7.04 \cdot 10^{-8}$	$4.46 \cdot 10^{-8}$	Zest	0.943	0.875	0.896	0.861
$4.45 \cdot 10^{-3}$	$8.28 \cdot 10^{-6}$	$4.83 \cdot 10^{-8}$	$1.65 \cdot 10^{-6}$	Zeugma-1PT	0.574	0.583	0.601
$6.10 \cdot 10^{-4}$	$3.17 \cdot 10^{-6}$	$4.84 \cdot 10^{-8}$	$4.95 \cdot 10^{-5}$	$4.31 \cdot 10^{-1}$	Zeugma-2PT	0.525	0.536
$4.97 \cdot 10^{-4}$	$2.14 \cdot 10^{-6}$	$4.85 \cdot 10^{-8}$	$1.76 \cdot 10^{-5}$	$3.78 \cdot 10^{-1}$	$7.97 \cdot 10^{-1}$	Zeugma-Link	0.565
$3.17 \cdot 10^{-4}$	$2.78 \cdot 10^{-6}$	$4.85 \cdot 10^{-8}$	$9.13 \cdot 10^{-5}$	$2.78 \cdot 10^{-1}$	$7.04 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$	Zeugma-X

BCEL After 5 Minutes

BD-Simple	0.557	0.647	0.610	0.555	0.785	0.674
$5.43 \cdot 10^{-1}$	BD-Struct	0.677	0.666	0.605	0.820	0.722
$1.14 \cdot 10^{-1}$	$5.65 \cdot 10^{-2}$	Zest	0.557	0.646	0.640	0.537
$2.39 \cdot 10^{-1}$	$7.42 \cdot 10^{-2}$	$5.43 \cdot 10^{-1}$	Zeugma-1PT	0.626	0.732	0.500
$5.61 \cdot 10^{-1}$	$2.62 \cdot 10^{-1}$	$1.17 \cdot 10^{-1}$	$1.76 \cdot 10^{-1}$	Zeugma-2PT	0.904	0.583
$2.14 \cdot 10^{-3}$	$5.63 \cdot 10^{-4}$	$1.33 \cdot 10^{-1}$	$1.23 \cdot 10^{-2}$	$1.33 \cdot 10^{-5}$	Zeugma-Link	0.770
$6.20 \cdot 10^{-2}$	$1.67 \cdot 10^{-2}$	$6.95 \cdot 10^{-1}$	1.00	$3.79 \cdot 10^{-1}$	$3.63 \cdot 10^{-3}$	Zeugma-X

BCEL After 3 Hours

BD-Simple	0.807	1.000	1.000	1.000	1.000	1.000
$9.18 \cdot 10^{-4}$	BD-Struct	0.941	0.985	0.979	1.000	0.994
$6.74 \cdot 10^{-8}$	$1.92 \cdot 10^{-6}$	Zest	0.591	0.585	1.000	0.779
$6.66 \cdot 10^{-8}$	$1.63 \cdot 10^{-7}$	$3.29 \cdot 10^{-1}$	Zeugma-1PT	0.537	0.999	0.764
$6.73 \cdot 10^{-8}$	$2.37 \cdot 10^{-7}$	$3.64 \cdot 10^{-1}$	$6.94 \cdot 10^{-1}$	Zeugma-2PT	1.000	0.776
$6.62 \cdot 10^{-8}$	$6.63 \cdot 10^{-8}$	$6.62 \cdot 10^{-8}$	$7.04 \cdot 10^{-8}$	$6.61 \cdot 10^{-8}$	Zeugma-Link	0.989
$6.72 \cdot 10^{-8}$	$9.79 \cdot 10^{-8}$	$2.65 \cdot 10^{-3}$	$4.47 \cdot 10^{-3}$	$2.90 \cdot 10^{-3}$	$1.29 \cdot 10^{-7}$	Zeugma-X

Closure After 5 Minutes

BD-Simple	0.541	0.995	0.922	1.000	1.000	1.000	1.000
$6.65 \cdot 10^{-1}$	BD-Struct	1.000	0.932	1.000	1.000	1.000	1.000
$9.16 \cdot 10^{-8}$	$6.79 \cdot 10^{-8}$	RLCheck	1.000	1.000	1.000	1.000	1.000
$5.17 \cdot 10^{-6}$	$3.07 \cdot 10^{-6}$	$6.79 \cdot 10^{-8}$	Zest	0.965	0.948	0.985	0.968
$6.79 \cdot 10^{-8}$	$6.79 \cdot 10^{-8}$	$6.78 \cdot 10^{-8}$	$5.22 \cdot 10^{-7}$	Zeugma-1PT	0.552	0.804	0.560
$6.78 \cdot 10^{-8}$	$6.78 \cdot 10^{-8}$	$6.77 \cdot 10^{-8}$	$1.37 \cdot 10^{-6}$	$5.79 \cdot 10^{-1}$	Zeugma-2PT	0.845	0.586
$6.80 \cdot 10^{-8}$	$6.80 \cdot 10^{-8}$	$6.79 \cdot 10^{-8}$	$1.66 \cdot 10^{-7}$	$1.06 \cdot 10^{-3}$	$1.99 \cdot 10^{-4}$	Zeugma-Link	0.738
$6.78 \cdot 10^{-8}$	$6.78 \cdot 10^{-8}$	$6.77 \cdot 10^{-8}$	$4.53 \cdot 10^{-7}$	$5.25 \cdot 10^{-1}$	$3.58 \cdot 10^{-1}$	$1.06 \cdot 10^{-2}$	Zeugma-X

Closure After 3 Hours

BD-Simple	0.575	1.000	1.000	0.887	0.789	1.000	0.953
$4.25 \cdot 10^{-1}$	BD-Struct	1.000	0.995	0.871	0.816	1.000	0.938
$6.80 \cdot 10^{-8}$	$6.80 \cdot 10^{-8}$	RLCheck	1.000	1.000	1.000	1.000	1.000
$6.79 \cdot 10^{-8}$	$9.16 \cdot 10^{-8}$	$6.79 \cdot 10^{-8}$	Zest	0.946	0.988	0.892	0.874
$2.92 \cdot 10^{-5}$	$6.24 \cdot 10^{-5}$	$6.80 \cdot 10^{-8}$	$1.47 \cdot 10^{-6}$	Zeugma-1PT	0.718	1.000	0.701
$1.86 \cdot 10^{-3}$	$6.53 \cdot 10^{-4}$	$6.79 \cdot 10^{-8}$	$1.43 \cdot 10^{-7}$	$1.93 \cdot 10^{-2}$	Zeugma-2PT	1.000	0.860
$6.80 \cdot 10^{-8}$	$6.80 \cdot 10^{-8}$	$6.80 \cdot 10^{-8}$	$2.30 \cdot 10^{-5}$	$6.80 \cdot 10^{-8}$	$6.79 \cdot 10^{-8}$	Zeugma-Link	1.000
$1.05 \cdot 10^{-6}$	$2.36 \cdot 10^{-6}$	$6.80 \cdot 10^{-8}$	$5.56 \cdot 10^{-5}$	$3.05 \cdot 10^{-2}$	$1.04 \cdot 10^{-4}$	$6.80 \cdot 10^{-8}$	Zeugma-X

Maven After 5 Minutes

BD-Simple	0.724	0.938	1.000	1.000	1.000	1.000	1.000
$1.60 \cdot 10^{-2}$	BD-Struct	0.502	1.000	1.000	1.000	1.000	1.000
$1.71 \cdot 10^{-6}$	$9.89 \cdot 10^{-1}$	RLCheck	1.000	1.000	1.000	1.000	1.000
$6.72 \cdot 10^{-8}$	$6.77 \cdot 10^{-8}$	$4.54 \cdot 10^{-8}$	Zest	0.968	0.720	1.000	0.993
$6.72 \cdot 10^{-8}$	$6.77 \cdot 10^{-8}$	$4.54 \cdot 10^{-8}$	$4.53 \cdot 10^{-7}$	Zeugma-1PT	0.921	0.751	0.619
$6.71 \cdot 10^{-8}$	$6.76 \cdot 10^{-8}$	$4.54 \cdot 10^{-8}$	$1.79 \cdot 10^{-2}$	$5.49 \cdot 10^{-6}$	Zeugma-2PT	0.968	0.948
$6.72 \cdot 10^{-8}$	$6.77 \cdot 10^{-8}$	$4.54 \cdot 10^{-8}$	$6.78 \cdot 10^{-8}$	$6.82 \cdot 10^{-3}$	$4.51 \cdot 10^{-7}$	Zeugma-Link	0.694
$6.68 \cdot 10^{-8}$	$6.73 \cdot 10^{-8}$	$4.52 \cdot 10^{-8}$	$1.06 \cdot 10^{-7}$	$2.03 \cdot 10^{-1}$	$1.36 \cdot 10^{-6}$	$3.72 \cdot 10^{-2}$	Zeugma-X

Maven After 3 Hours

BD-Simple	0.968	0.950	1.000	1.000	1.000	1.000	1.000
$4.52 \cdot 10^{-7}$	BD-Struct	0.800	1.000	1.000	1.000	1.000	1.000
$2.10 \cdot 10^{-7}$	$5.49 \cdot 10^{-4}$	RLCheck	1.000	1.000	1.000	1.000	1.000
$6.76 \cdot 10^{-8}$	$6.77 \cdot 10^{-8}$	$7.98 \cdot 10^{-9}$	Zest	1.000	0.988	1.000	1.000
$3.78 \cdot 10^{-8}$	$3.78 \cdot 10^{-8}$	$3.73 \cdot 10^{-9}$	$3.78 \cdot 10^{-8}$	Zeugma-1PT	0.906	0.604	0.627
$6.65 \cdot 10^{-8}$	$6.66 \cdot 10^{-8}$	$7.82 \cdot 10^{-9}$	$1.40 \cdot 10^{-7}$	$6.57 \cdot 10^{-6}$	Zeugma-2PT	0.960	0.965
$1.94 \cdot 10^{-8}$	$1.94 \cdot 10^{-8}$	$1.54 \cdot 10^{-9}$	$1.94 \cdot 10^{-8}$	$1.41 \cdot 10^{-1}$	$1.81 \cdot 10^{-7}$	Zeugma-Link	0.525
$1.51 \cdot 10^{-8}$	$1.51 \cdot 10^{-8}$	$1.10 \cdot 10^{-9}$	$1.51 \cdot 10^{-8}$	$5.98 \cdot 10^{-2}$	$1.05 \cdot 10^{-7}$	$6.54 \cdot 10^{-1}$	Zeugma-X

Nashorn After 5 Minutes

BD-Simple	0.569	1.000	1.000	1.000	1.000	1.000	1.000
$4.65 \cdot 10^{-1}$	BD-Struct	1.000	1.000	0.959	0.950	0.995	0.968
$8.01 \cdot 10^{-9}$	$7.99 \cdot 10^{-9}$	RLCheck	1.000	1.000	1.000	1.000	1.000
$6.79 \cdot 10^{-8}$	$6.78 \cdot 10^{-8}$	$7.99 \cdot 10^{-9}$	Zest	1.000	1.000	1.000	1.000
$6.80 \cdot 10^{-8}$	$7.40 \cdot 10^{-7}$	$8.01 \cdot 10^{-9}$	$6.79 \cdot 10^{-8}$	Zeugma-1PT	0.733	0.943	0.617
$6.79 \cdot 10^{-8}$	$1.20 \cdot 10^{-6}$	$7.99 \cdot 10^{-9}$	$6.78 \cdot 10^{-8}$	$1.23 \cdot 10^{-2}$	Zeugma-2PT	0.973	0.805
$6.80 \cdot 10^{-8}$	$9.16 \cdot 10^{-8}$	$8.01 \cdot 10^{-9}$	$6.79 \cdot 10^{-8}$	$1.80 \cdot 10^{-6}$	$3.41 \cdot 10^{-7}$	Zeugma-Link	0.907
$6.79 \cdot 10^{-8}$	$4.53 \cdot 10^{-7}$	$7.99 \cdot 10^{-9}$	$6.78 \cdot 10^{-8}$	$2.08 \cdot 10^{-1}$	$1.01 \cdot 10^{-3}$	$1.10 \cdot 10^{-5}$	Zeugma-X

Nashorn After 3 Hours

BD-Simple	0.512	1.000	1.000	1.000	1.000	1.000	1.000
$9.03 \cdot 10^{-1}$	BD-Struct	1.000	1.000	1.000	1.000	1.000	1.000
$7.98 \cdot 10^{-9}$	$8.01 \cdot 10^{-9}$	RLCheck	1.000	1.000	1.000	1.000	1.000
$6.77 \cdot 10^{-8}$	$6.79 \cdot 10^{-8}$	$7.99 \cdot 10^{-9}$	Zest	1.000	1.000	1.000	1.000
$6.77 \cdot 10^{-8}$	$6.79 \cdot 10^{-8}$	$7.99 \cdot 10^{-9}$	$6.78 \cdot 10^{-8}$	Zeugma-1PT	0.715	0.953	0.705
$6.78 \cdot 10^{-8}$	$6.80 \cdot 10^{-8}$	$8.01 \cdot 10^{-9}$	$6.79 \cdot 10^{-8}$	$2.07 \cdot 10^{-2}$	Zeugma-2PT	1.000	0.887
$6.76 \cdot 10^{-8}$	$6.78 \cdot 10^{-8}$	$7.98 \cdot 10^{-9}$	$6.77 \cdot 10^{-8}$	$1.04 \cdot 10^{-6}$	$6.78 \cdot 10^{-8}$	Zeugma-Link	0.993
$6.76 \cdot 10^{-8}$	$6.78 \cdot 10^{-8}$	$7.98 \cdot 10^{-9}$	$6.77 \cdot 10^{-8}$	$2.74 \cdot 10^{-2}$	$2.92 \cdot 10^{-5}$	$1.06 \cdot 10^{-7}$	Zeugma-X

Rhino After 5 Minutes

BD-Simple	0.738	1.000	0.935	0.995	0.950	1.000	0.990
$1.06 \cdot 10^{-2}$	BD-Struct	1.000	0.975	1.000	0.986	1.000	1.000
$4.89 \cdot 10^{-8}$	$4.90 \cdot 10^{-8}$	RLCheck	1.000	1.000	1.000	1.000	1.000
$2.68 \cdot 10^{-6}$	$2.94 \cdot 10^{-7}$	$4.87 \cdot 10^{-8}$	Zest	0.945	0.664	1.000	0.848
$9.15 \cdot 10^{-8}$	$6.79 \cdot 10^{-8}$	$4.89 \cdot 10^{-8}$	$1.57 \cdot 10^{-6}$	Zeugma-1PT	0.796	0.932	0.547
$1.20 \cdot 10^{-6}$	$1.53 \cdot 10^{-7}$	$4.89 \cdot 10^{-8}$	$7.86 \cdot 10^{-2}$	$1.41 \cdot 10^{-3}$	Zeugma-2PT	0.980	0.716
$6.77 \cdot 10^{-8}$	$6.78 \cdot 10^{-8}$	$4.89 \cdot 10^{-8}$	$6.74 \cdot 10^{-8}$	$3.05 \cdot 10^{-6}$	$2.21 \cdot 10^{-7}$	Zeugma-Link	0.945
$1.23 \cdot 10^{-7}$	$6.78 \cdot 10^{-8}$	$4.89 \cdot 10^{-8}$	$1.79 \cdot 10^{-4}$	$6.17 \cdot 10^{-1}$	$2.00 \cdot 10^{-2}$	$1.57 \cdot 10^{-6}$	Zeugma-X

Rhino After 3 Hours

BD-Simple	0.575	1.000	1.000	1.000	1.000	1.000	1.000
$4.24 \cdot 10^{-1}$	BD-Struct	1.000	1.000	1.000	1.000	1.000	1.000
$1.50 \cdot 10^{-8}$	$1.51 \cdot 10^{-8}$	RLCheck	1.000	1.000	1.000	1.000	1.000
$6.72 \cdot 10^{-8}$	$6.76 \cdot 10^{-8}$	$1.51 \cdot 10^{-8}$	Zest	0.958	0.973	0.995	0.975
$6.72 \cdot 10^{-8}$	$6.76 \cdot 10^{-8}$	$1.51 \cdot 10^{-8}$	$7.93 \cdot 10^{-7}$	Zeugma-1PT	0.652	0.829	0.637
$6.73 \cdot 10^{-8}$	$6.77 \cdot 10^{-8}$	$1.51 \cdot 10^{-8}$	$3.41 \cdot 10^{-7}$	$1.02 \cdot 10^{-1}$	Zeugma-2PT	0.890	0.546
$6.72 \cdot 10^{-8}$	$6.76 \cdot 10^{-8}$	$1.51 \cdot 10^{-8}$	$9.15 \cdot 10^{-8}$	$3.94 \cdot 10^{-4}$	$2.59 \cdot 10^{-5}$	Zeugma-Link	0.895
$6.73 \cdot 10^{-8}$	$6.77 \cdot 10^{-8}$	$1.51 \cdot 10^{-8}$	$2.96 \cdot 10^{-7}$	$1.40 \cdot 10^{-1}$	$6.26 \cdot 10^{-1}$	$2.04 \cdot 10^{-5}$	Zeugma-X

Tomcat After 5 Minutes

BD-Simple	1.000	0.914	0.849	0.850	0.861	0.896	0.884
$6.19 \cdot 10^{-8}$	BD-Struct	1.000	1.000	1.000	1.000	1.000	1.000
$5.29 \cdot 10^{-6}$	$3.81 \cdot 10^{-8}$	RLCheck	0.666	0.627	0.650	0.589	0.552
$1.67 \cdot 10^{-4}$	$6.23 \cdot 10^{-8}$	$6.73 \cdot 10^{-2}$	Zest	0.517	0.557	0.559	0.561
$1.57 \cdot 10^{-4}$	$6.14 \cdot 10^{-8}$	$1.62 \cdot 10^{-1}$	$8.60 \cdot 10^{-1}$	Zeugma-1PT	0.530	0.569	0.580
$9.66 \cdot 10^{-5}$	$6.26 \cdot 10^{-8}$	$1.02 \cdot 10^{-1}$	$5.42 \cdot 10^{-1}$	$7.55 \cdot 10^{-1}$	Zeugma-2PT	0.614	0.605
$1.87 \cdot 10^{-5}$	$6.14 \cdot 10^{-8}$	$3.35 \cdot 10^{-1}$	$5.33 \cdot 10^{-1}$	$4.64 \cdot 10^{-1}$	$2.23 \cdot 10^{-1}$	Zeugma-Link	0.521
$3.43 \cdot 10^{-5}$	$6.26 \cdot 10^{-8}$	$5.71 \cdot 10^{-1}$	$5.16 \cdot 10^{-1}$	$3.93 \cdot 10^{-1}$	$2.61 \cdot 10^{-1}$	$8.28 \cdot 10^{-1}$	Zeugma-X

Tomcat After 3 Hours

BD-Simple	1.000	0.824	0.501	0.752	0.512	0.782	0.805
$6.48 \cdot 10^{-8}$	BD-Struct	1.000	1.000	1.000	1.000	1.000	1.000
$3.21 \cdot 10^{-4}$	$3.77 \cdot 10^{-8}$	RLCheck	0.807	0.999	0.879	0.965	0.965
1.00	$6.62 \cdot 10^{-8}$	$6.84 \cdot 10^{-4}$	Zest	0.735	0.521	0.756	0.797
$6.19 \cdot 10^{-3}$	$6.50 \cdot 10^{-8}$	$3.93 \cdot 10^{-8}$	$1.09 \cdot 10^{-2}$	Zeugma-1PT	0.776	0.561	0.621
$9.03 \cdot 10^{-1}$	$6.58 \cdot 10^{-8}$	$2.83 \cdot 10^{-5}$	$8.28 \cdot 10^{-1}$	$2.73 \cdot 10^{-3}$	Zeugma-2PT	0.806	0.838
$2.12 \cdot 10^{-3}$	$5.92 \cdot 10^{-8}$	$2.78 \cdot 10^{-7}$	$5.17 \cdot 10^{-3}$	$5.09 \cdot 10^{-1}$	$8.81 \cdot 10^{-4}$	Zeugma-Link	0.585
$9.19 \cdot 10^{-4}$	$6.30 \cdot 10^{-8}$	$2.95 \cdot 10^{-7}$	$1.24 \cdot 10^{-3}$	$1.89 \cdot 10^{-1}$	$2.52 \cdot 10^{-4}$	$3.54 \cdot 10^{-1}$	Zeugma-X

Figure 15: Pairwise Defect Detection Rates. Pairwise comparisons of defect detection rates per defect on long (three hour) and short (five minute) campaigns for BeDiv-Simple (BeDiv-Simple), BeDiv-Struct (BeDiv-Struct), RLCheck, Zest, Zeugma-X, Zeugma-1PT, Zeugma-2PT, and Zeugma-Link. The main diagonal of each matrix consists of the fuzzers. Each matrix entry not on the main diagonal at some row *r* and column *c* compares the fuzzer in row *r* to the fuzzer in column *c*. Fisher's exact test two-tailed p-values are placed in the entries below the main diagonal. Odds ratios are placed in the entries above the main diagonal. Each reported odds ratio is the ratio of the higher odds to the lower odds for the compared pair. Statistically significant p-values are colored blue. Larger effect sizes are colored darker shades of red.

B0 After 5 Minutes

BD-Simple	3.154	1.000	1.000	14.548	14.548	3.154
1.00	BD-Struct	3.154	3.154	6.333	6.333	1.000
1.00	1.00	Zest	1.000	14.548	14.548	3.154
1.00	1.00	1.00	Zeugma-1PT	14.548	14.548	3.154
$4.71 \cdot 10^{-2}$	$1.82 \cdot 10^{-1}$	$4.71 \cdot 10^{-2}$	$4.71 \cdot 10^{-2}$	Zeugma-2PT	1.000	6.333
$4.71 \cdot 10^{-2}$	$1.82 \cdot 10^{-1}$	$4.71 \cdot 10^{-2}$	$4.71 \cdot 10^{-2}$	1.00	Zeugma-Link	6.333
1.00	1.00	1.00	1.00	$1.82 \cdot 10^{-1}$	$1.82 \cdot 10^{-1}$	Zeugma-X

B0 After 3 Hours

BD-Simple	1.256	22.778	22.778	22.778	22.778	22.778
1.00	BD-Struct	18.379	18.379	18.379	18.379	18.379
$8.32 \cdot 10^{-3}$	$2.02 \cdot 10^{-2}$	Zest	1.000	1.000	1.000	1.000
$8.32 \cdot 10^{-3}$	$2.02 \cdot 10^{-2}$	1.00	Zeugma-1PT	1.000	1.000	1.000
$8.32 \cdot 10^{-3}$	$2.02 \cdot 10^{-2}$	1.00	1.00	Zeugma-2PT	1.000	1.000
$8.32 \cdot 10^{-3}$	$2.02 \cdot 10^{-2}$	1.00	1.00	1.00	Zeugma-Link	1.000
$8.32 \cdot 10^{-3}$	$2.02 \cdot 10^{-2}$	1.00	1.00	1.00	1.00	Zeugma-X

B1 After 5 Minutes

BD-Simple	1.000	1.000	1.000	1.000	1.000	1.000
1.00	BD-Struct	1.000	1.000	1.000	1.000	1.000
1.00	1.00	Zest	1.000	1.000	1.000	1.000
1.00	1.00	1.00	Zeugma-1PT	1.000	1.000	1.000
1.00	1.00	1.00	1.00	Zeugma-2PT	1.000	1.000
1.00	1.00	1.00	1.00	1.00	Zeugma-Link	1.000
1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-X

B1 After 3 Hours

BD-Simple	1.000	1.000	5.541	5.541	1.000	1.000
1.00	BD-Struct	1.000	5.541	5.541	1.000	1.000
1.00	1.00	Zest	5.541	5.541	1.000	1.000
$4.87 \cdot 10^{-1}$	$4.87 \cdot 10^{-1}$	$4.87 \cdot 10^{-1}$	Zeugma-1PT	1.000	5.541	5.541
$4.87 \cdot 10^{-1}$	$4.87 \cdot 10^{-1}$	$4.87 \cdot 10^{-1}$	1.00	Zeugma-2PT	5.541	5.541
1.00	1.00	1.00	$4.87 \cdot 10^{-1}$	$4.87 \cdot 10^{-1}$	Zeugma-Link	1.000
1.00	1.00	1.00	$4.87 \cdot 10^{-1}$	$4.87 \cdot 10^{-1}$	1.00	Zeugma-X

C0 After 5 Minutes

BD-Simple	1.000	1.000	1.000	1.000	1.000	1.000	3.154
1.00	BD-Struct	1.000	1.000	1.000	1.000	1.000	3.154
1.00	1.00	RLCheck	1.000	1.000	1.000	1.000	3.154
1.00	1.00	1.00	Zest	1.000	1.000	1.000	3.154
1.00	1.00	1.00	1.00	Zeugma-1PT	1.000	1.000	3.154
1.00	1.00	1.00	1.00	1.00	Zeugma-2PT	1.000	3.154
1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-Link	3.154
1.00	1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-X

C0 After 3 Hours

BD-Simple	3.353	2.111	12.667	12.667	12.667	35.286	6.333
$6.05 \cdot 10^{-1}$	BD-Struct	1.588	3.778	3.778	3.778	10.524	1.889
1.00	1.00	RLCheck	6.000	6.000	6.000	16.714	3.000
$1.97 \cdot 10^{-2}$	$1.55 \cdot 10^{-1}$	$6.48 \cdot 10^{-2}$	Zest	1.000	1.000	2.786	2.000
$1.97 \cdot 10^{-2}$	$1.55 \cdot 10^{-1}$	$6.48 \cdot 10^{-2}$	1.00	Zeugma-1PT	1.000	2.786	2.000
$1.97 \cdot 10^{-2}$	$1.55 \cdot 10^{-1}$	$6.48 \cdot 10^{-2}$	1.00	1.00	Zeugma-2PT	2.786	2.000
$1.37 \cdot 10^{-4}$	$3.06 \cdot 10^{-3}$	$7.72 \cdot 10^{-4}$	$2.05 \cdot 10^{-1}$	$2.05 \cdot 10^{-1}$	$2.05 \cdot 10^{-1}$	Zeugma-Link	5.571
$1.82 \cdot 10^{-1}$	$6.95 \cdot 10^{-1}$	$4.07 \cdot 10^{-1}$	$5.01 \cdot 10^{-1}$	$5.01 \cdot 10^{-1}$	$5.01 \cdot 10^{-1}$	$2.48 \cdot 10^{-2}$	Zeugma-X

C1 After 5 Minutes

BD-Simple	8.200	1.417	1.588	5.667	22.667	8.500	8.500
$2.31 \cdot 10^{-1}$	BD-Struct	11.182	5.541	41.000	150.333	60.294	60.294
1.00	$1.06 \cdot 10^{-1}$	RLCheck	2.250	4.000	16.000	6.000	6.000
1.00	$4.87 \cdot 10^{-1}$	$6.61 \cdot 10^{-1}$	Zest	9.000	36.000	13.500	13.500
$4.07 \cdot 10^{-2}$	$4.36 \cdot 10^{-4}$	$9.58 \cdot 10^{-2}$	$1.38 \cdot 10^{-2}$	Zeugma-1PT	4.000	1.500	1.500
$8.75 \cdot 10^{-5}$	$1.54 \cdot 10^{-7}$	$3.60 \cdot 10^{-4}$	$1.66 \cdot 10^{-5}$	$9.58 \cdot 10^{-2}$	Zeugma-2PT	2.667	2.667
$7.91 \cdot 10^{-3}$	$4.51 \cdot 10^{-5}$	$2.25 \cdot 10^{-2}$	$2.20 \cdot 10^{-3}$	$7.51 \cdot 10^{-1}$	$3.01 \cdot 10^{-1}$	Zeugma-Link	1.000
$7.91 \cdot 10^{-3}$	$4.51 \cdot 10^{-5}$	$2.25 \cdot 10^{-2}$	$2.20 \cdot 10^{-3}$	$7.51 \cdot 10^{-1}$	$3.01 \cdot 10^{-1}$	1.00	Zeugma-X

C1 After 3 Hours

BD-Simple	1.000	1.000	1.000	1.000	1.000	1.000	1.000
1.00	BD-Struct	1.000	1.000	1.000	1.000	1.000	1.000
1.00	1.00	RLCheck	1.000	1.000	1.000	1.000	1.000
1.00	1.00	1.00	Zest	1.000	1.000	1.000	1.000
1.00	1.00	1.00	1.00	Zeugma-1PT	1.000	1.000	1.000
1.00	1.00	1.00	1.00	1.00	Zeugma-2PT	1.000	1.000
1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-Link	1.000
1.00	1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-X

N0 After 5 Minutes

BD-Simple	1.000	1.000	1.000	1.000	1.000	3.154	1.000
1.00	BD-Struct	1.000	1.000	1.000	1.000	3.154	1.000
1.00	1.00	RLCheck	1.000	1.000	1.000	3.154	1.000
1.00	1.00	1.00	Zest	1.000	1.000	3.154	1.000
1.00	1.00	1.00	1.00	Zeugma-1PT	1.000	3.154	1.000
1.00	1.00	1.00	1.00	1.00	Zeugma-2PT	3.154	1.000
1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-Link	3.154
1.00	1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-X

N0 After 3 Hours

BD-Simple	1.000	1.000	1.000	18.379	33.870	1681.000	115.5
1.00	BD-Struct	1.000	1.000	18.379	33.870	1681.000	115.5
1.00	1.00	RLCheck	1.000	18.379	33.870	1681.000	115.5
1.00	1.00	1.00	Zest	18.379	33.870	1681.000	115.5
$2.02 \cdot 10^{-2}$	$2.02 \cdot 10^{-2}$	$2.02 \cdot 10^{-2}$	$2.02 \cdot 10^{-2}$	Zeugma-1PT	1.909	91.462	7.0
$1.23 \cdot 10^{-3}$	$1.23 \cdot 10^{-3}$	$1.23 \cdot 10^{-3}$	$1.23 \cdot 10^{-3}$	$5.14 \cdot 10^{-1}$	Zeugma-2PT	49.632	3.6
$1.45 \cdot 10^{-11}$	$1.45 \cdot 10^{-11}$	$1.45 \cdot 10^{-11}$	$1.45 \cdot 10^{-11}$	$3.34 \cdot 10^{-6}$	$1.45 \cdot 10^{-4}$	Zeugma-Link	14.5
$7.71 \cdot 10^{-7}$	$7.71 \cdot 10^{-7}$	$7.71 \cdot 10^{-7}$	$7.71 \cdot 10^{-7}$	$1.04 \cdot 10^{-2}$	$1.05 \cdot 10^{-1}$	$4.71 \cdot 10^{-2}$	Zeugma

N1 After 5 Minutes

BD-Simple	1.000	1.000	1.000	1.000	1.000	1.000	1.000
1.00	BD-Struct	1.000	1.000	1.000	1.000	1.000	1.000
1.00	1.00	RLCheck	1.000	1.000	1.000	1.000	1.000
1.00	1.00	1.00	Zest	1.000	1.000	1.000	1.000
1.00	1.00	1.00	1.00	Zeugma-1PT	1.000	1.000	1.000
1.00	1.00	1.00	1.00	1.00	Zeugma-2PT	1.000	1.000
1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-Link	1.000
1.00	1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-X

N1 After 3 Hours

BD-Simple	1.000	1.000	1.000	1.000	3.154	1.000	1.000
1.00	BD-Struct	1.000	1.000	1.000	3.154	1.000	1.000
1.00	1.00	RLCheck	1.000	1.000	3.154	1.000	1.000
1.00	1.00	1.00	Zest	1.000	3.154	1.000	1.000
1.00	1.00	1.00	1.00	Zeugma-1PT	3.154	1.000	1.000
1.00	1.00	1.00	1.00	1.00	Zeugma-2PT	3.154	3.154
1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-Link	1.000
1.00	1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-X

N2 After 5 Minutes

BD-Simple	1.000	1.000	1.000	1.000	1.000	1.000	1.000
1.00	BD-Struct	1.000	1.000	1.000	1.000	1.000	1.000
1.00	1.00	RLCheck	1.000	1.000	1.000	1.000	1.000
1.00	1.00	1.00	Zest	1.000	1.000	1.000	1.000
1.00	1.00	1.00	1.00	Zeugma-1PT	1.000	1.000	1.000
1.00	1.00	1.00	1.00	1.00	Zeugma-2PT	1.000	1.000
1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-Link	1.000
1.00	1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-X

N2 After 3 Hours

BD-Simple	1.000	1.000	1.000	1.000	3.154	41.000	3.154
1.00	BD-Struct	1.000	1.000	1.000	3.154	41.000	3.154
1.00	1.00	RLCheck	1.000	1.000	3.154	41.000	3.154
1.00	1.00	1.00	Zest	1.000	3.154	41.000	3.154
1.00	1.00	1.00	1.00	Zeugma-1PT	3.154	41.000	3.154
1.00	1.00	1.00	1.00	1.00	Zeugma-2PT	19.000	1.000
$4.36 \cdot 10^{-4}$	$4.36 \cdot 10^{-4}$	$4.36 \cdot 10^{-4}$	$4.36 \cdot 10^{-4}$	$4.36 \cdot 10^{-4}$	$3.34 \cdot 10^{-3}$	Zeugma-Link	19.000
1.00	1.00	1.00	1.00	1.00	1.00	$3.34 \cdot 10^{-3}$	Zeugma-X

Figure 16: Pairwise Defect Detection Rates Continued.

R0 After 5 Minutes

BD-Simple	1.000	1.000	1.000	1.000	1.000	1.000	1.000
1.00	BD-Struct	1.000	1.000	1.000	1.000	1.000	1.000
1.00	1.00	RLCheck	1.000	1.000	1.000	1.000	1.000
1.00	1.00	1.00	Zest	1.000	1.000	1.000	1.000
1.00	1.00	1.00	1.00	Zeugma-1PT	1.000	1.000	1.000
1.00	1.00	1.00	1.00	1.00	Zeugma-2PT	1.000	1.000
1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-Link	1.000
1.00	1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-X

R0 After 3 Hours

BD-Simple	1.000	1.000	5.541	91.462	60.294	150.333	115.545
1.00	BD-Struct	1.000	5.541	91.462	60.294	150.333	115.545
1.00	1.00	RLCheck	5.541	91.462	60.294	150.333	115.545
$4.87 \cdot 10^{-1}$	$4.87 \cdot 10^{-1}$	$4.87 \cdot 10^{-1}$	Zest	21.000	13.500	36.000	27.000
$3.34 \cdot 10^{-6}$	$3.34 \cdot 10^{-6}$	$3.34 \cdot 10^{-6}$	$2.44 \cdot 10^{-4}$	Zeugma-1PT	1.556	1.714	1.286
$4.51 \cdot 10^{-5}$	$4.51 \cdot 10^{-5}$	$4.51 \cdot 10^{-5}$	$2.20 \cdot 10^{-3}$	$7.41 \cdot 10^{-1}$	Zeugma-2PT	2.667	2.000
$1.54 \cdot 10^{-7}$	$1.54 \cdot 10^{-7}$	$1.54 \cdot 10^{-7}$	$1.66 \cdot 10^{-5}$	$7.16 \cdot 10^{-1}$	$3.01 \cdot 10^{-1}$	Zeugma-Link	1.333
$7.71 \cdot 10^{-7}$	$7.71 \cdot 10^{-7}$	$7.71 \cdot 10^{-7}$	$6.86 \cdot 10^{-5}$	1.00	$5.01 \cdot 10^{-1}$	1.00	Zeugma-X

R1 After 5 Minutes

BD-Simple	4.750	3.154	1.000	2.111	2.111	2.111	3.353
$3.42 \cdot 10^{-1}$	BD-Struct	11.182	4.750	2.250	2.250	2.250	1.417
1.00	$1.06 \cdot 10^{-1}$	RLCheck	3.154	5.541	5.541	5.541	8.200
1.00	$3.42 \cdot 10^{-1}$	1.00	Zest	2.111	2.111	2.111	3.353
1.00	$6.61 \cdot 10^{-1}$	$4.87 \cdot 10^{-1}$	1.00	Zeugma-1PT	1.000	1.000	1.588
1.00	$6.61 \cdot 10^{-1}$	$4.87 \cdot 10^{-1}$	1.00	1.00	Zeugma-2PT	1.000	1.588
1.00	$6.61 \cdot 10^{-1}$	$4.87 \cdot 10^{-1}$	1.00	1.00	1.00	Zeugma-Link	1.588
$6.05 \cdot 10^{-1}$	1.00	$2.31 \cdot 10^{-1}$	$6.05 \cdot 10^{-1}$	1.00	1.00	1.00	Zeugma-X

R1 After 3 Hours

BD-Simple	2.270	33.870	23.222	3.667	11.000	2.852	4.889
$3.41 \cdot 10^{-1}$	BD-Struct	73.800	10.231	1.615	4.846	1.256	2.154
$1.23 \cdot 10^{-3}$	$1.29 \cdot 10^{-5}$	RLCheck	533.000	115.545	303.400	91.462	150.333
$1.25 \cdot 10^{-3}$	$4.36 \cdot 10^{-2}$	$3.05 \cdot 10^{-10}$	Zest	6.333	2.111	8.143	4.750
$1.05 \cdot 10^{-1}$	$7.31 \cdot 10^{-1}$	$7.71 \cdot 10^{-7}$	$1.82 \cdot 10^{-1}$	Zeugma-1PT	3.000	1.286	1.333
$5.74 \cdot 10^{-3}$	$1.27 \cdot 10^{-1}$	$3.35 \cdot 10^{-9}$	1.00	$4.07 \cdot 10^{-1}$	Zeugma-2PT	3.857	2.250
$2.00 \cdot 10^{-1}$	1.00	$3.34 \cdot 10^{-6}$	$9.15 \cdot 10^{-2}$	1.00	$2.35 \cdot 10^{-1}$	Zeugma-Link	1.714
$4.84 \cdot 10^{-2}$	$4.80 \cdot 10^{-1}$	$1.54 \cdot 10^{-7}$	$3.42 \cdot 10^{-1}$	1.00	$6.61 \cdot 10^{-1}$	$7.16 \cdot 10^{-1}$	Zeugma-X

R2 After 5 Minutes

BD-Simple	1.000	1.000	1.000	1.000	1.000	1.000	1.000
1.00	BD-Struct	1.000	1.000	1.000	1.000	1.000	1.000
1.00	1.00	RLCheck	1.000	1.000	1.000	1.000	1.000
1.00	1.00	1.00	Zest	1.000	1.000	1.000	1.000
1.00	1.00	1.00	1.00	Zeugma-1PT	1.000	1.000	1.000
1.00	1.00	1.00	1.00	1.00	Zeugma-2PT	1.000	1.000
1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-Link	1.000
1.00	1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-X

R2 After 3 Hours

BD-Simple	1.000	1.000	11.182	205.000	91.462	205.000	533.00
1.00	BD-Struct	1.000	11.182	205.000	91.462	205.000	533.00
1.00	1.00	RLCheck	11.182	205.000	91.462	205.000	533.00
$1.06 \cdot 10^{-1}$	$1.06 \cdot 10^{-1}$	$1.06 \cdot 10^{-1}$	Zest	22.667	9.333	22.667	76.00
$2.57 \cdot 10^{-8}$	$2.57 \cdot 10^{-8}$	$2.57 \cdot 10^{-8}$	$8.75 \cdot 10^{-5}$	Zeugma-1PT	2.429	1.000	3.35
$3.34 \cdot 10^{-6}$	$3.34 \cdot 10^{-6}$	$3.34 \cdot 10^{-6}$	$3.64 \cdot 10^{-3}$	$4.51 \cdot 10^{-1}$	Zeugma-2PT	2.429	8.14
$2.57 \cdot 10^{-8}$	$2.57 \cdot 10^{-8}$	$2.57 \cdot 10^{-8}$	$8.75 \cdot 10^{-5}$	1.00	$4.51 \cdot 10^{-1}$	Zeugma-Link	3.35
$3.05 \cdot 10^{-10}$	$3.05 \cdot 10^{-10}$	$3.05 \cdot 10^{-10}$	$2.21 \cdot 10^{-6}$	$6.05 \cdot 10^{-1}$	$9.15 \cdot 10^{-2}$	$6.05 \cdot 10^{-1}$	Zeugma-

R3 After 5 Minutes

BD-Simple	1.000	1.000	1.000	1.000	1.000	1.000	1.000
1.00	BD-Struct	1.000	1.000	1.000	1.000	1.000	1.000
1.00	1.00	RLCheck	1.000	1.000	1.000	1.000	1.000
1.00	1.00	1.00	Zest	1.000	1.000	1.000	1.000
1.00	1.00	1.00	1.00	Zeugma-1PT	1.000	1.000	1.000
1.00	1.00	1.00	1.00	1.00	Zeugma-2PT	1.000	1.000
1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-Link	1.000
1.00	1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-X

R3 After 3 Hours

BD-Simple	1.000	1.000	1.000	1.000	1.000	1.000	1.000
1.00	BD-Struct	1.000	1.000	1.000	1.000	1.000	1.000
1.00	1.00	RLCheck	1.000	1.000	1.000	1.000	1.000
1.00	1.00	1.00	Zest	1.000	1.000	1.000	1.000
1.00	1.00	1.00	1.00	Zeugma-1PT	1.000	1.000	1.000
1.00	1.00	1.00	1.00	1.00	Zeugma-2PT	1.000	1.000
1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-Link	1.000
1.00	1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-X

R4 After 5 Minutes

BD-Simple	1.000	1.000	1.000	1.000	1.000	1.000	1.000
1.00	BD-Struct	1.000	1.000	1.000	1.000	1.000	1.000
1.00	1.00	RLCheck	1.000	1.000	1.000	1.000	1.000
1.00	1.00	1.00	Zest	1.000	1.000	1.000	1.000
1.00	1.00	1.00	1.00	Zeugma-1PT	1.000	1.000	1.000
1.00	1.00	1.00	1.00	1.00	Zeugma-2PT	1.000	1.000
1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-Link	1.000
1.00	1.00	1.00	1.00	1.00	1.00	1.00	Zeugma-X

R4 After 3 Hours

BD-Simple	1.000	1.000	3.154	205.000	205.000	533.000	303.
1.00	BD-Struct	1.000	3.154	205.000	205.000	533.000	303.
1.00	1.00	RLCheck	3.154	205.000	205.000	533.000	303.
1.00	1.00	1.00	Zest	107.667	107.667	361.000	171.
$2.57 \cdot 10^{-8}$	$2.57 \cdot 10^{-8}$	$2.57 \cdot 10^{-8}$	$4.06 \cdot 10^{-7}$	Zeugma-1PT	1.000	3.353	1.
$2.57 \cdot 10^{-8}$	$2.57 \cdot 10^{-8}$	$2.57 \cdot 10^{-8}$	$4.06 \cdot 10^{-7}$	1.00	Zeugma-2PT	3.353	1.
$3.05 \cdot 10^{-10}$	$3.05 \cdot 10^{-10}$	$3.05 \cdot 10^{-10}$	$5.82 \cdot 10^{-9}$	$6.05 \cdot 10^{-1}$	$6.05 \cdot 10^{-1}$	Zeugma-Link	2.
$3.35 \cdot 10^{-9}$	$3.35 \cdot 10^{-9}$	$3.35 \cdot 10^{-9}$	$5.82 \cdot 10^{-8}$	1.00	1.00	1.00	Zeugm

5 DYNAMIC TAINT TRACKING FOR MODERN JAVA VIRTUAL MACHINES

5.1 INTRODUCTION

Dynamic taint tracking traces the flow of information through a program by associating labels, also called "taint tags", with program data and propagating these labels as the program executes. By tracing different types of information, dynamic taint tracking has been used to prevent confidential information from being leaked to public channels (McCamant and Ernst, 2008; Cox et al., 2014), detect security vulnerabilities (Halfond et al., 2006; Hough et al., 2020a; Kieżun et al., 2009; Qin et al., 2006), assist in automated input generation systems (Rawat et al., 2017; Ganesh et al., 2009; Wang et al., 2010; Kukucka et al., 2022), provide debugging guidance (Clause and Orso, 2009; Attariyan and Flinn, 2010), and reason about configurable systems (Velez et al., 2021; Attariyan and Flinn, 2010).

Java is the second most popular programming language according to the PYPL PopularitY of Programming Language Index as of January 2024 (Carbonnelle, 2024) and the fourth most popular programming language according to the 2024 TIOBE Programming Community Index (TIOBE Software BV, 2024). Due to the popularity of Java and other languages that compile to Java bytecode (e.g., Kotlin, Scala, and Clojure), researchers have explored different techniques for performing dynamic taint tracking in the JVM (Bell and Kaiser, 2014; Perkins et al., 2020; Nair et al., 2008; Halfond et al., 2006; Franz et al., 2005; Ouyang et al., 2023; Chandra and Franz, 2007). However, due to changes to the Java platform, these techniques perform poorly on newer Java versions—inducing deviations from an application's normal behavior, inaccurately propagating taint tags, and even causing the JVM to crash in some cases.

Since its release in 1995, several major versions of the Java platform have been released to add new features and better support the needs of the Java community (Oracle Corporation, 2023c). As of August 2024, there are four long-term support (LTS) versions of Java that are officially supported by Oracle: 8, 11, 17, and 21 (Oracle Corporation, 2023c). The most recent of these, Java 21, was released in September 2023 (Oracle Corporation, 2023c). The Java community has widely adopted these modern LTS Java versions. New Relic, Inc. (2023) found that 56%, 33%, and 9%, of the applications that reported to them used Java 11, 8, and 21 in production in 2023, respectively.

Given the prevalence of newer Java versions, it is critical that dynamic taint tracking systems support modern Java features. However, existing taint tracking tools fail to address two major features introduced in newer Java versions: *signature polymorphism* and *modules*. The challenges introduced by these features impact other dynamic analyses, e.g., dynamic invariant detectors (Ernst et al., 2007), and have yet to be addressed by existing work. In our empirical evaluation, we found that state-of-the-art dynamic taint tracking systems were unreliable and inaccurate when analyzing newer Java applications. To support dynamic taint tracking in modern JVMs, this work makes the following contributions:

- A technique for instrumenting Java classes used in the initialization of the JVM without violating constraints imposed by Java's module system.
- A technique for reasoning about the flow of information through signature polymorphic methods.
- A precise, robust technique for performing full-system dynamic taint tracking that minimizes deviations from an application's normal behavior.

- An open-source dynamic taint tracking system for modern JVMs that uses these techniques: Galette.
- An evaluation of Galette's accuracy, performance, and ability to preserve program semantics compared against two state-of-the-art dynamic taint tracking systems for the JVM: Phosphor (Bell and Kaiser, 2014) and MirrorTaint (Ouyang et al., 2023).

5.2 BACKGROUND

5.2.1 The Java Virtual Machine

The JVM is an abstract computing machine that runs Java bytecode programs (Lindholm et al., 2013). Java bytecode instructions operate on two kinds of values: primitive and reference. A reference value is a pointer to an object—a class instance or an array. Primitive values represent raw integral, floating-point, or boolean values. The JVM supports multiple, concurrent threads of execution. Each thread of execution has its own program counter and JVM stack. The JVM stack consists of JVM frames. Each of these frames contains an array of local variables and an operand stack which are used to store values for a particular method invocation. There is one active or "current" frame for each thread; this frame stores values for the method invocation that is currently executing on that thread (Lindholm et al., 2013). Java bytecode instructions can manipulate the current frame's local variables and operand stack to store, load, and perform operations on values.

When a method is invoked, a new frame is created for the callee method. The JVM passes any parameters to the callee method by loading them from the top elements of the caller's operand stack and storing them into the new frame's local variables. Then, the new frame becomes the current frame and control is transferred from the call site in the caller method to the entry point of the callee method. When the callee method completes, the current frame passes the return value of the method invocation, if any, to the top of the previous frame's operand stack. The current frame is then destroyed, the previous frame is made current, and control is transferred back to the caller.

In addition to local variables and the operand stack, the "heap" is also used to store values. The heap is shared between all threads and contains objects and arrays. Java bytecode instructions can manipulate heap data by loading and storing elements from arrays, loading and storing values from objects' fields, or by accessing the length of an array.

5.2.2 Dynamic Taint Tracking for the Java Virtual Machine

Dynamic taint tracking systems have three primary responsibilities: associating taint tags with values, passing taint tags between procedure calls, and propagating taint tags in response to operations. Instrumentation-based taint tracking systems fulfill these responsibilities by transforming Java bytecode. There are two main approaches to instrumentation-based taint tracking in the JVM: "shadowing" (Bell and Kaiser, 2014; Perkins et al., 2020) and "mirroring" (Ouyang et al., 2023).

Shadowing

Shadowing was proposed by Bell and Kaiser (2014) and Perkins et al. (2020). Shadowing stores taint tags alongside original values. The taint tag associated with a local variable is stored in a new "shadow local variable" in the same JVM stack frame as the original local variable. The operand stack is modified to store

```java
class X {
  static int i = 7;
  static void set(int b) {
    X.i = b;
  }

  static int run(int[] arr, int y) {
    int a = arr[y];
    set(a);
    return a;
  }
}
```

Listing 5.1: Original Java class.

```java
class X {
  static int i = 7;
  static Tag i$$TAG;
  static void set(int b, Tag b$$TAG) {
    X.i$$TAG = b$$TAG;
    X.i = b;
  }

  static int run(TaggedIntArray arr, Tag arr$$TAG,
     int y, Tag y$$TAG, RetVal ret) {
    Tag a$$TAG = Tag.union(arr.tags[y], y$$TAG);
    int a = arr.values[y];
    set(a, a$$TAG);
    ret.tag = a$$TAG;
    return a;
  }
}
```

Listing 5.2: Instrumentation applied by shadowing.

```java
class X {
  static int i = 7;
  static int set(int b) {
    MirrorFrame frame = MirrorStack.get(
      "set",
      "(I)V"
    );
    frame.loadLocal(0);
    frame.putStaticField(X.class, "i", "I");
    X.i = b;
    frame.prepareReturn();
  }

  static int run(int[] arr, int y) {
    MirrorFrame frame = MirrorStack.get(
      "run",
      "([II)I"
    );
    frame.loadLocal(0);
    frame.loadLocal(1);
    frame.getArrayElement();
    frame.storeLocal(2);
    int a = arr[y];
    frame.loadLocal(2);
    frame.prepareCall("set", "(I)V");
    set(a);
    frame.postCall();
    frame.loadLocal(2);
    frame.prepareReturn();
    return a;
  }
}
```

Listing 5.3: Instrumentation applied by mirroring.

the taint tag of each operand, its corresponding “shadow operand”, immediately next to the original operand in the same operand stack. Classes are instrumented to add a “shadow field” to store the tag associated with each original field. To store the taint tags associated with the lengths and elements of arrays, each array is replaced with a wrapper object which stores the original array and its taint tags.

Shadowing uses “shadow parameters” to pass taint tags between method calls. For each original method, a corresponding “shadow method” is created. The shadow method’s signature contains the original method’s parameters plus a corresponding shadow parameter for each original parameter to store the taint tag associated with the original parameter. An extra shadow parameter is added to the method’s signature to store the return value’s taint tag if the method has a return value. Each original method call is replaced with a call to the corresponding shadow method.

Shadowing adds logic to propagate taint tags directly into the body of methods. Operations on taint tags happen in the same method call as the original program logic using the same stack frame.

Listing 5.2 shows the instrumentation applied by shadowing to the class in Listing 5.1. The name of each shadow field or shadow local variable is the name of the original value that it shadows followed by the suffix “`$$Tag`”. The class `TaggedIntArray` is used a wrapper that stores an original array and its taint tags together.

Because shadows are stored in the same fashion as original values and tag propagation occurs in the original method call, shadowing is able to match the runtime semantics of the JVM resulting in precise taint tag propagation. However, the invasiveness of the changes made by shadowing may cause divergences between an application’s normal behavior and its behavior when performing the taint analysis including unexpected crashes (Ouyang et al., 2023). We observed these divergences in our empirical evaluation of dynamic taint tracking systems for the JVM (discussed in Section 5.6).

Mirroring

Mirroring was proposed by Ouyang et al. (2023). Mirroring stores the taint tags associated with values in "mirrored JVM stack frames" and a "mirrored heap". The taint tag associated with a local variable is stored in the corresponding local variable slot in the mirrored frame for a method invocation. Similarly, the taint tags associated with the operand stack are stored in an operand stack within the mirrored frame. The mirrored heap uses a mapping to associate an object with a mapping from the names of its fields to taint tags. This mapping is also used to store the taint tags associated with arrays.

Mirroring uses a "mirrored JVM stack" to pass taint tags between method calls. Before a method call, mirroring creates a mirrored frame for that call. This frame is passed the taint tags of the arguments passed to the call. The frame is then pushed to a thread-local mirrored JVM stack. When control is transferred to the callee, this frame is accessed from the stack. The frame is also used to pass the taint tag of the return value of the method invocation, if any, to the caller method.

Mirroring does not add logic to propagate taint tags directly into method bodies. Instead, executed instructions are forwarded to the active mirrored frame which manipulates the mirrored heap and frame to model the effects of the operation.

Listing 5.3 shows the instrumentation applied by mirroring to the class in Listing 5.1. Methods have been modified to forward executed instructions to a `MirrorFrame` instance retrieved from a thread-local mirrored stack. The actual logic for propagating taint tags is contained in the `MirrorFrame` class.

Mirroring is less invasive than shadowing; it does not require changes to be made to classes' field, methods' signatures, or the class hierarchy. However, mirroring does not correctly model the runtime semantics of the JVM. As shown in our empirical evaluation (discussed in Section 5.6), incorrect modeling of JVM semantics can result in inaccurate taint tag propagation. For example, assume that there is a class `B` that directly extends a class `A` and `A` declares a static field `i`. A field reference `B.i` may refer to the same field as the reference `A.i` or a different field depending on whether `B` also declares a field `i` (Lindholm et al., 2013). To correctly model the JVM's field resolution semantics for field accesses, mirroring would need the name of the field being accessed, the descriptor of the field, the class name used to access the field, the fields declared by the class referenced by the specified class name, and the fields declared by the supertypes of the class referenced by the specified class name (Lindholm et al., 2013).

Furthermore, mirroring may associate mirrored frames with the wrong method call. JVM implementations are allowed to make "up-calls" into the Java runtime between the execution of Java bytecode instructions (Lindholm et al., 2013). In some cases, these calls are required by the specification, for example, to initialize a class, load a class, or create an implicitly thrown exception (Lindholm et al., 2013). These calls can occur between the execution of an instruction inserted by the instrumentation to push a mirrored frame and the execution of the instruction inserted by the instrumentation to retrieve the pushed frame in the callee method. This can result in a mirrored frame being associated with the wrong method call causing incorrect taint tag propagation.

5.2.3 Signature Polymorphism

Signature polymorphism was introduced to support dynamic language features in the JVM (Oracle Corporation, 2024f; Lindholm et al., 2013). When linking and invoking signature polymorphic method calls, the

JVM follows alternative semantics to allow for greater flexibility in how these methods can be used (Lindholm et al., 2013). As of Java 21, there are two classes that can contain signature polymorphic methods: `java.lang.invoke.MethodHandle` (released in Java 7) and `java.lang.invoke.VarHandle` (released in Java 9) (Oracle Corporation, 2023b; Lindholm et al., 2013). Method handles provide a type-safe, performant way to dynamically access methods, constructors, and fields (Oracle Corporation, 2023b). Method handles are used to implement the bytecode instruction `invokedynamic` which was added in Java 7 and can be used to invoke a dynamically computed call site (Oracle Corporation, 2024f, 2023b; Lindholm et al., 2013). The `invokedynamic` instruction has been used to facilitate the implementation of dynamically typed languages for the JVM (released in Java 7) (Oracle Corporation, 2024f), to support lambda expressions in Java (released in Java 8) (Oracle Corporation, 2024g), to improve string concatenation in Java (released in Java 9) (Oracle Corporation, 2024c), and to add record types to Java (released in Java 16) (Oracle Corporation, 2024d). Variable handles provide type-safe, reflective access to variables (Oracle Corporation, 2023b). Variable handles were proposed to replace uses of the `java.util.concurrent.atomic` and `sun.misc.Unsafe` APIs by allowing programmers to directly use a variety of access-consistency policies (Oracle Corporation, 2024b). The alternative semantics used by the JVM to link and invoke signature polymorphic methods are what enables method and variable handles to support these use cases.

Before a Java bytecode instruction used to call an instance method (`invokevirtual`) can be executed, it must be linked by the JVM to resolve the symbolic method reference used by the instruction (Lindholm et al., 2013). Normally, this resolution succeeds only if the JVM is able to find a method with the name and type descriptor, a string indicating a method's parameters and return type, specified by the symbolic reference in or inherited by the class specified by the symbolic reference (Lindholm et al., 2013). However, calls to signature polymorphic methods can be resolved regardless of their type descriptor (Oracle Corporation, 2023b). This allows signature polymorphic methods to be called with a variety of signatures and return types (Oracle Corporation, 2023b; Lindholm et al., 2013).

When a signature polymorphic method is invoked, the symbolic type descriptor at the call site is compared against the type descriptor expected by the receiver of the method call (the method or variable handle) to ensure that the call is type safe (Oracle Corporation, 2023b; Lindholm et al., 2013). If this comparison succeeds, then the handle's underlying behavior or variable is directly invoked or accessed (Oracle Corporation, 2023b). The specifics of how this is achieved are left largely up to the JVM implementation (Oracle Corporation, 2023b).

Consider an execution of the program in Listing 5.4 on an OpenJDK Corretto (version 11.0.22.7.1) runtime. The program begins by creating a method handle for the `add` method (lines 5–6). On line 7, a call to the signature polymorphic method `MethodHandle#invokeExact` appears. When the JVM goes to execute this instruction, it first makes an up-call into application code to the method `java.lang.invoke.MethodHandleNatives#findMethodHandleType` to obtain a `java.lang.invoke.MethodType` instance to represent the symbolic type descriptor for this call site. Next, the obtained `MethodType` instance is passed along with additional information about the call site to `MethodHandleNatives#linkMethod`. The method `linkMethod` computes a pointer to an adapter method and an appendix value for the call site; these results are stored into the constant pool cache (Rose, 2013). Subsequent executions of this call to `invokeExact` can use the stored method pointer and appendix directly without re-resolving them (Rose, 2013). In this example, the adapter method is dynamically generated by the JVM. Decompiled source code for this adapter method is shown in Listing 5.5. Once the adapter method pointer and appendix value are resolved, the adapter method is called. This call is passed the receiver of the original call to `invokeExact` (the `MethodHandle` instance), followed by the original arguments passed to `invokeExact`, and trailed by the resolved appendix value (Rose, 2013).

```java
1 import java.lang.invoke.*;
2
3 public class Example {
4   public static void main(String[] args) throws
       Throwable {
5     MethodType mt = MethodType.methodType(int.
       class, int.class, int.class);
6     MethodHandle mh = MethodHandles.lookup().
       findStatic(Example.class, "add", mt);
7     int j = (int) mh.invokeExact(23, 4);
8   }
9
10  static int add(int x, int y) {return x + y;}
11 }
```

Listing 5.4: A program that uses a `MethodHandle`.

```java
static int invokeExact_MT(Object o, int i, int j, Object
    mt) {
  MethodHandle mh = (MethodHandle) o;
  Invokers.checkExactType(mh, (MethodType) mt);
  Invokers.checkCustomized(mh);
  return mh.invokeBasic(i, j);
}
```

Listing 5.5: A generated adaptor method.

```java
static int invokeStatic(Object mh, int i, int j) {
  Object mn = DirectMethodHandle.internalMemberName(mh);
  return MethodHandle.linkToStatic(i, j, (MemberName) mn);
}
```

Listing 5.6: `Holder#invokeStatic`.

The adapter checks whether the `MethodHandle`'s type matches the symbolic type descriptor for the call site. This check succeeds. Next, the adapter calls another signature polymorphic method, `MethodHandle #invokeBasic`. This call is handled specially by the JVM, which intrinsically links it to the method `Holder#invokeStatic` shown in Listing 5.6. The method `invokeStatic` obtains a `java.lang.invoke. MemberName` for the passed `MethodHandle` and calls another signature polymorphic method, `MethodHandle #linkToStatic`. This call is also handled specially by the JVM, which pops the trailing `MemberName` instance passed to `linkToStatic` and links the call to the target method indicated by the `MemberName`, in this case `Example#add`. Finally, `add` executes normally returning the value `27` which is passed up the call stack, to the call to `invokeExact`, and assigned to the local variable `j` on line 7 of `Example#main`.

As demonstrated in this example, the invocation semantics for signature polymorphic methods allow the JVM to "intrinsically" modify the arguments passed to signature polymorphic method calls. As a result, the arguments passed at a signature polymorphic method call site may not match the arguments received by the callee method. We experimented with a variety of signature polymorphic method calls on eight different Java Development Kit (JDK)s: OpenJDK Temurin (versions 1.8.0_402-b06, 11.0.22+7, 17.0.10+7, and 21.0.2+13) and OpenJDK Corretto (versions 8.402.08.1, 11.0.22.7.1, 17.0.10.8.1, and 21.0.2.14.1). In these experiments, we observed the following intrinsic modifications: addition of trailing reference-typed values; removal of trailing reference-typed values; and conversion of `boolean`, `byte`, `short`, or `char` primitive values to type `int`.

Unfortunately, the flexibility that makes signature polymorphic methods useful also presents problems for dynamic taint tracking systems. Dynamic taint tracking systems need to reason about the flow of information between method calls. As described above, mirroring pass tags between method calls using mirrored frames and shadowing uses shadow parameters. Before executing a signature polymorphic method, the JVM can make calls into the Java runtime to resolve symbolic references used by the method invocation instruction. These calls can prevent mirroring from correctly associating the mirrored frame with the actual callee method. Shadowing is unaffected by any calls made by the JVM into the Java runtime because it explicitly passes taint tags from the caller to the callee as arguments. However, the modifications that shadowing makes to the type descriptors of method calls to add shadow parameters causes a mismatch between the type descriptor at the call site and the type descriptor expected by the method or variable handle. Additionally, if the JVM intrinsically modifies the arguments to the call, then the corresponding shadow arguments will not be modified.

5.2.4 The Java Platform Module System

The JPMS was introduced in Java 9 (Oracle Corporation, 2024h). The JPMS adds support for modules, groupings of related packages and resources. Each Java module contains a module declaration which specifies the name of the module, the other modules on which it depends (requires), the packages it exports to other modules (exports), the packages it makes available to other modules via reflection (opens), the services it provides (provides), and the services it consumes (uses) (Lindholm et al., 2013). The dependencies and constraints defined in a module declaration are enforced by the JVM at runtime. Making use of the JPMS, the JCL, the standard library included with the Java runtime, was decomposed into several modules starting in Java 9 (Oracle Corporation, 2024h). One of these modules, `java.base`, is the primordial module; every other module depends on `java.base` (Lindholm et al., 2013).

Instrumentation-based taint tracking systems modify classes' bytecode often inserting references to classes that are part of the taint tracking system (e.g., the class that represents a taint tag). If the instrumentation inserts a reference to a class that is not part of a module on which the module containing the class being instrumented declares a dependency, then the JVM will fail to resolve the inserted reference (Lindholm et al., 2013). Fortunately, Java provides support for instrumenting classes at runtime as they are loaded through the use of a Java agent, and this support includes considerations for modules. If a class is instrumented at runtime using a Java agent, the JVM will transform the class' module to add a dependency on the unnamed module of the bootstrap class loader and the class loader that loads the main agent class (Oracle Corporation, 2023b).

Unfortunately, not all classes can be instrumented as they are loaded using a Java agent; certain classes are loaded during the initialization of the JVM before a Java agent can be attached (e.g., `java.lang.String`). In theory, these classes can be instrumented using the method `java.lang.instrument.Instrumentation #retransformClasses`, which allows classes that have already been loaded to be instrumented (Oracle Corporation, 2023b). However, in some Java versions, there are restrictions on the types of changes that can be made when re-transforming a loaded class. For example, you cannot add or remove methods when re-transforming a class in Java 11 (Oracle Corporation, 2023a). These restrictions can prohibit necessary bytecode transformations, particularly for shadowing. Therefore, classes that are loaded during the initialization of the JVM need to be instrumented statically before the Java runtime is started. However, this static instrumentation will introduce dependencies on classes that are part of the taint tracking system violating constraints imposed by the module system. Standard approaches for dynamically adding a dependency between modules cannot be used to correct this. Some classes are loaded before the Java command line option `--add-reads` is processed and before `java.lang.Module#addReads` can be called. If these core Java classes are not instrumented by a taint tracking system, it will result in taint tags being lost in critical classes like `java.lang.String`. Furthermore, approaches like shadowing assume the presence of instrumentation in classes and may crash in its absence.

5.3 APPROACH

Galette, our approach for dynamic taint tracking in modern JVM's, incorporates elements of both shadowing and mirroring to provide precise, robust taint tag propagation while minimizing deviations from an application's normal behavior. By strategically using elements of both approaches, we are able to preserve the precision of shadowing without the fragility introduced by certain bytecode modifications. This added robustness allows Galette to accommodate modern Java features. We also present techniques for reasoning about the flow of information through signature polymorphic methods and for statically instrumenting Java classes that are loaded before a Java agent can be attached without violating constraints imposed by the module system. These techniques enable GALETTE to track taint tags through the entirety of a Java runtime preventing taint tags from being lost due to missing propagation logic.

5.3.1 Instrumentation Application

In order to associate taint tags with values and propagate those taint tags in response to operations, GALETTE instruments Java classes, modifying their bytecode. Because taint tags will not be propagated through code that is not instrumented, all classes being analyzed need to be instrumented to ensure correctness. GALETTE uses a Java agent to dynamically instrument classes as they are loaded. However, as noted in Section 5.2.4, certain JCL classes are loaded before a Java agent can be attached and need to be instrumented statically before the Java runtime is started. Therefore, GALETTE statically instruments the entirety of the JCL.

Unfortunately, GALETTE's static instrumentation introduces an undeclared dependency between the module of the code being instrumented and GALETTE, thereby violating the constraints imposed by Java's module system. GALETTE circumvents this issue by packing its classes and packages into the primordial module, `java.base`. Every module other than `java.base` has an implicit dependency on `java.base` (Lindholm et al., 2013). Therefore, any class can reference a class in a package exported by `java.base` without violating the constraints imposed by Java's module system. In addition to packing its classes into `java.base`, GALETTE updates the module declaration for `java.base` to export the packages of GALETTE's classes so that they can be accessed from other modules.

5.3.2 Associating Taint Tags with Values

GALETTE stores the taint tags associated with values using a combination of shadow variables and mirrored tag stores.

Local Variables

Similar to shadowing, GALETTE stores the taint tag associated with each original local variable in a corresponding "shadow local variable" in the local variable array of the same JVM stack frame as the original local variable. For example, in the original runtime shown in Figure 17a, there are three local variables: a reference to an instance of type `X`, an `int` 0, and a reference to an array of `int` values. In the corresponding instrumented runtime (Figure 17b), local variables 4–6 are used to store the taint tags associated with the three original local variables.

The Operand Stack

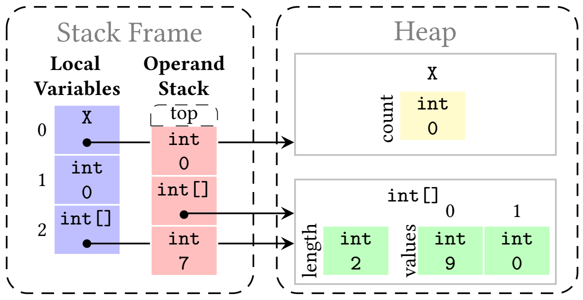
GALETTE stores the taint tag associated with each original operand in the operand stack in a "shadow operand stack". The shadow operand stack is placed at the end of the local variable array in the same JVM stack frame as the original operand stack. The top of the shadow operand stack is always located in the last local variable index. For example, in the original runtime shown in Figure 17a, the operand stack has three operands: an `int` 0 (on the top), a reference to an array of `int` values, and another `int` 7. In the corresponding instrumented runtime (Figure 17b), local variables 7–9 are used to store the taint tags

```java
public class X {
    int count = 7;
    int sum(int i, int[]
      a) {
        return this.count
       + a[i];
    }
}
```

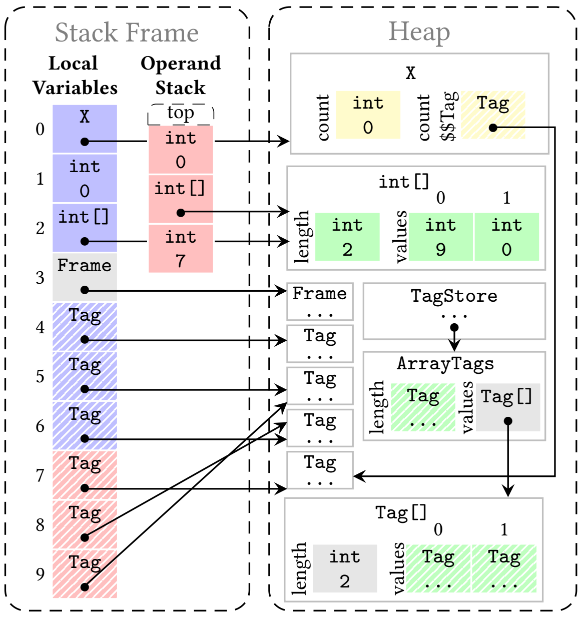
Listing 5.7: A Java class

```
aload_0
getfield X#count:I
aload_2
iload_1
iaload
iadd
ireturn
```

Listing 5.8: Bytecode for the `sum` method



(a) Original runtime memory.



(b) GALETTE-instrumented runtime memory.

Figure 17: Figures 17a and 17b show the active JVM stack frame and heap before executing an array access, the `iaload` instruction in Listing 5.8, for an original runtime and a GALETTE-instrumented runtime, respectively. Locations containing original values found in the local variable array, the operand stack, fields, and arrays are colored solid blue, red, yellow, and green, respectively. Locations containing the taint tags associated with original values found in the local variable array, the operand stack, fields, and arrays are colored with hatched lines in blue, red, yellow, and green, respectively.

associated with the original operands. The taint tag of the top operand, the `int` 0, is stored in index 9, the last local variable.

Fields

GALETTE stores the taint tag associated with each field by adding a corresponding "shadow field" to the declaring class. This shadow field uses the same name as the original field with a consistent, known suffix (e.g., `$$Tag`) appended to its end. For example, in the original runtime shown in Figure 17a, the class `X` declares a single field, `count`. The first local variable contains a reference to an instance of type `X`; this instance has a `count` value of 0. In the corresponding instrumented runtime (Figure 17b), the class `X` declares two fields: `count` and its shadow, `count$$Tag`. The first local variable still contains a reference to an instance of type `X`. However, this instance now has a `count` value of 0 and a `count$$Tag` value which points to a `Tag` object in the heap.

Shadow fields are declared with the same modifiers as the original field, on the same class, and with a consistent, known suffix. This enables a shadow field to be accessed in the same fashion as the corresponding original field causing the reference to the shadow field to be resolved by the JVM in the same fashion as the original field. This removes the need to model the JVM's field resolution semantics, which, as noted in Section 5.2.2, is a limitation of mirroring fields.

```java
1  class X {
2    static int i = 7;
3    static Tag i$$TAG;
4    static void set(int b, Frame frame) {
5      Tag b$$TAG = frame.get(0);
6      X.i$$TAG = b$$TAG;
7      X.i = b;
8    }
9
10   static int run(int[] arr, int y, Frame frame) {
11     Tag y$$TAG = frame.get(0);
12     Tag arr$$TAG = frame.get(1);
13     Tag a$$TAG = TagStore.getTag(arr, y, arr$$TAG,
        y$$TAG);
14     int a = arr[y];
15     Frame frame2 = new Frame(a$$Tag);
16     set(a, frame2);
17     frame.setReturnTag(a$$TAG);
18     return a;
19   }
20 }
```

Listing 5.9: Instrumentation applied by Galette to the class in Listing 5.1.

Arrays

Unlike shadowing, Galette does not create shadow wrappers for arrays. Instead, Galette maintains a global array tag store similar to the mirrored heap described in Section 5.2.2. This tag store maps each array to a record containing the taint tags associated with the array's length and elements. For example, in the original runtime shown in Figure 17a, the local variable in index 2 refers to an array of `int` values. In the corresponding instrumented runtime (Figure 17b), the `ArrayTags` instance mapped to this array can be retrieved from a global `TagStore`. This `ArrayTags` instance contains a `Tag` representing the taint tag associated with the array's length and an array containing the taint tags associated with the array's elements.

Unlike field accesses, the semantics of array accesses do not depend on the type hierarchy. Thus, a global tag store can be used to precisely store the taint tags associated with an array's length and elements. The use of a global tag store over array wrappers removes a source of brittleness that affects shadowing. In particular, if an array wrapper is passed into code that is not instrumented (e.g., native methods), it may cause the program to behave unexpectedly or even crash.

5.3.3 Propagating Taint Tags

Galette modifies the bodies of methods to insert logic for propagating taint tags in response to operations. This propagation happens in the same method call as the original program logic. Listing 5.9 shows the instrumentation applied by Galette to the class in Listing 5.1.

When an original instruction modifies the operand stack, Galette performs the same modification to the shadow operand stack. When an original instruction accesses a local variable, Galette accesses the corresponding shadow local variable. For example, on line 14 of Listing 5.9, the local variable `a` is declared and defined. On line 13, Galette's instrumentation declares and defines a shadow local variable `a$$TAG` to store the taint tag associated with `a`. Since the definition of `a` uses the value of the variables `arr` and `y`, the definition of `a$$TAG` directly uses the taint tags associated with `arr` and `y` which are stored in the shadow variables `arr$$TAG` and `y$$TAG`, respectively.

When an original instruction accesses a field, Galette accesses the corresponding shadow field. For example, since a value is stored to the field `i` on line 7 of Listing 5.9, Galette stores a value to the corresponding shadow field `i$$TAG` on line 6.

When an original instruction accesses the length or an element of an array, Galette accesses the taint tags associated with the accessed value from a global array tag store. For instance, on line 14 of Listing 5.9, the element at index `y` of the array `arr` is accessed. On line 13, Galette retrieves the taint tag associated with that element from the tag store by calling `TagStore#getTag`.

5.3.4 Passing Taint Tags Between Methods

Like mirroring, GALETTE uses a “tag frame” to pass taint tags between method calls. This tag frame stores the taint tag associated with each argument passed to a method call and, after the method call finishes, the taint tag of the return value of the call. Unlike mirroring, GALETTE does not use a mirrored JVM stack to pass this tag frame from caller to callee. Instead, GALETTE directly passes a tag frame as an argument to method calls by modifying method descriptors and call sites. This ensures that tag frames are associated with the correct method call even in the presence of up-calls from the JVM into the Java runtime.

For each original method, GALETTE creates a corresponding “shadow method”. The shadow method’s signature contains the original method’s parameters plus a trailing tag frame. Before a method call, GALETTE creates a tag frame containing the taint tags of the arguments to be passed to the call (including the receiver of the method call). This tag frame is passed as the last argument to the call. The original method call is then replaced with a call to the corresponding shadow method. At the beginning of the callee method, the tag frame passed from the caller is used to initialize the taint tags associated with received arguments. At the end of the callee method, GALETTE sets the taint tag associated with the return value of the method invocation on the tag frame passed from the caller. After the method call returns, in the caller method, GALETTE retrieves the taint tag associated with the method invocation’s return value from the passed tag frame.

For example, on line 15 of Listing 5.9, GALETTE creates a new tag frame for a call to the method `set`. This frame is then passed to the call to set on line 16. On line 5 of the shadow method for `set`, the taint tag for the parameter `b` is retrieved from the tag frame and stored to the shadow local variable for `b`.

Wrapper Methods

The original methods defined in a class cannot safely be removed in favor of the corresponding shadow methods, because it is possible that these methods may still be called by native code or an up-call from the JVM. GALETTE keeps all original method definitions in a class, but replaces the bodies of these methods with wrapper code that obtains a tag frame and then calls the corresponding shadow method for that original method.

5.3.5 Signature Polymorphic Methods

As described in Section 5.2.3, signature polymorphic methods do not follow the same invocation semantics as normal methods. As a consequence of these alternative semantics, the type descriptor of signature polymorphic method call cannot be modified to explicitly pass a tag frame as an argument to the call. Such a modification would produce a mismatch between the type descriptor at the call site and the type descriptor expected by the method or variable handle receiver of the call. Therefore, GALETTE indirectly passes the taint tags of arguments to the targets of signature polymorphic method calls using a thread-local “frame store”. The frame store for a thread of execution may be empty or hold a single frame containing the taint tags associated with the arguments of the next target of a signature polymorphic method call to be executed on that thread.

GALETTE’s frame store is impacted by similar limitations as the mirrored JVM stack used in mirroring. In particular, up-calls made by the JVM between when a tag frame is stored and the callee method is executed may cause an indirectly passed frame to be associated with the wrong method call. To minimize the likelihood of these misassociations, GALETTE records additional information about the method call in the frame store. This information is used to distinguish between the actual target of the signature polymorphic method call and up-calls made by the JVM into the runtime.

GALETTE instruments signature polymorphic method call sites to insert code to prepare the frame store for the call and restore the state of the frame store after the call. First, this instrumentation creates a tag frame containing the taint tags associated with each argument passed to the method call (including the receiver of the method call). This is identical to how a tag frame is created for a non-signature polymorphic method call. Next, this frame and the actual arguments to be passed to the method call are stored into the frame store for the current thread of execution. Then, the signature polymorphic method is called normally—without a trailing tag frame. Finally, when this call completes (normally or exceptionally), the frame store for the current thread of execution is cleared.

To receive the frame stored at signature polymorphic method call site, the target callee method must check the frame store. Thus, GALETTE applies instrumentation to any potential target of a signature polymorphic method call to check the frame store. Because GALETTE does not modify method or variable handles, the underlying method referenced by a method or variable handle will always be a method declared in the original, un-instrumented class—not a shadow method added by GALETTE. As described in Section 5.3.4, original methods are converted by GALETTE into wrappers which obtain a tag frame and then call the corresponding shadow method. Since any wrapper can be a potential target of a signature polymorphic method call, when a wrapper attempts to obtain a tag frame, it first checks the frame store for the current thread of execution. If the store is empty, then the wrapper will call its corresponding shadow method with an empty frame, a frame indicating that none of the arguments are tainted. If the store is not empty, the wrapper will retrieve the stored entry and clear the store. Next, the wrapper checks whether the arguments received by the wrapper match the arguments passed at the signature polymorphic call site, the ones stored in the retrieved entry. This matching procedure is relaxed and compensates for the intrinsic modifications discussed in Section 5.2.3. If the wrapper's arguments match the stored arguments, then the wrapper will call its corresponding shadow method with the stored tag frame. Otherwise, the wrapper will call its corresponding shadow method with an empty frame. Before returning, the wrapper restores the state of the frame store to what it was at the start of the execution of the wrapper.

For example, consider the call to the signature polymorphic method `linkToStatic` shown in Listing 5.6. Before this call, GALETTE will create a tag frame containing the taint tags associated with the arguments `i`, `j`, and `mn`. Next it will store this frame and the argument values `23`, `4`, and a `MemberName` instance to the frame store for the current thread of execution. Then call to `linkToStatic` is performed. The JVM pops the trailing `MemberName` instance passed to `linkToStatic` and links the call to the wrapper method for `Example#add`. This wrapper retrieves the stored entry for the current thread of execution from the store and clears the store. Next, the wrapper checks whether the arguments it was passed, `x` and `y`, match the arguments stored by the caller. The value of `x` will be equal to the stored value `23`, and the value of `y` will be equal to the stored value `4`. The relaxed matching algorithm will discard the trailing `MemberName` which was intrinsically removed by the JVM allowing the match to succeed. As a result, the wrapper will call its corresponding shadow method with the stored frame allowing the taint tags stored for the arguments to propagate to the callee.

5.3.6 Native Methods

The Java Native Interface allows Java bytecode programs to interoperate with programs written in other languages through the use of native methods (Oracle Corporation, 2024a). GALETTE does not track the flow of information through these native methods. For each native method, GALETTE creates a special shadow method which discards the passed tag frame and calls the original native method with the remaining arguments.

5.3.7 Reflection and Unsafe

Java's Reflection API allows developers to inspect the attributes of classes at runtime, access fields, manipulate arrays, and invoke methods. Java's Unsafe API allows developers to perform low-level, unsafe operations including accessing array elements and fields. GALETTE inserts special instrumentation to ensure that taint tags are propagated through reflective and unsafe operations. Additionally, GALETTE "masks" calls to reflective methods that expose changes made by GALETTE to an application which could cause deviations from the program's original behavior. For example, to hide shadow fields from reflection, GALETTE masks calls to reflective methods that expose the fields of a class (e.g., `java.lang.Class#getFields`). After a method that exposes the fields of a class is called, GALETTE intercepts the return value to remove added shadow fields.

5.4 IMPLEMENTATION

Our implementation of GALETTE uses the ASM (OW2 Consortium, 2024) Java bytecode manipulation and analysis framework to instrument Java classes. As mentioned in Section 5.3.1, we use a Java agent to dynamically instrument most classes as they are loaded, but we statically instrument the JCL. For Java versions 8 or less, the JCL is statically instrumented by processing the class files contained in a Java installation. For Java versions 9+, we statically instrument the JCL using jlink, a tool included in the JDK starting in Java 9 to build custom runtime images (Oracle Corporation, 2024e). Before running a Java 9+ application with GALETTE, jlink is invoked with two custom plugins to create an instrumented Java runtime. The first plugin applies GALETTE's instrumentation to Java classes. The second plugin is responsible for packing GALETTE's classes into `java.base` and updating the module declaration for `java.base` as described in Section 5.3.1.

To implement the global array tag store described in Section 5.3.2, we used `java.lang.ref.WeakReference` instances to store the array keys. Unlike standard Java references, weak reference objects do not prevent their referents from being reclaimed by JVM's garbage collector. This ensures that when an array is no longer reachable through standard references it can still be reclaimed by JVM's garbage collector even if there is an entry for it the tag store.

To create the thread-local frame stores described in Section 5.3.5, we instrumented `java.lang.Thread`, the class that the Java platform uses to represent a thread of execution, to add an instance field that holds a frame store. The method `Thread#currentThread` can be used to obtain the `Thread` instance representing the current thread of execution (Oracle Corporation, 2023b). Then, the current thread's frame store can be accessed from the added field on the obtained `Thread` instance.

As discussed in Section 5.3.5, GALETTE uses a relaxed matching procedure when indirectly passing tag frames from signature polymorphic method call sites to callee methods. Our implementation of GALETTE uses a matching procedure that compensates for the intrinsic modifications described in Section 5.2.3, but could easily be modified to accommodate different intrinsic modifications.

5.5 LIMITATIONS

As mentioned in Section 5.3.6, GALETTE does not track the flow of information through native methods. More broadly, GALETTE does not track the flow of information outside the Java runtime. This includes through native code, databases, the network, and the file system.

GALETTE approach to tracking the flow of information through signature polymorphic method calls is not guaranteed to be correct. It is still possible that an up-call into the JVM is incorrectly matched as the target of a signature polymorphic method call. It is also possible that the relaxed matching procedure is not sufficiently permissive to ensure that the arguments received by the true target method call are always considered to be a match to the arguments passed at the call site. However, because GALETTE replaces all non-signature polymorphic method calls with calls to the corresponding shadow method, these mismatches can only occur for signature polymorphic method calls and up-calls made by the JVM. Furthermore, mismatches are limited to up-calls with almost identical arguments to the true target method call.

GALETTE currently only supports tracing the flow of information through "explicit" or "data" flows (Sabelfeld and Myers, 2006). GALETTE does not support tracing the flow of information through "implicit" or "control" flows (McCamant and Ernst, 2008). However, GALETTE could be extended to support different propagation logic, e.g., CONFLUX's semantics as described in Chapter 3. This extension could be similar to the extension that we made to PHOSPHOR to add support for custom control flow propagation policies (described in Section 3.4). However, instead of adding a new method argument (i.e., the `ControlFlowStack`) to pass control flow and context information between method boundaries, the existing tag frames used by GALETTE could simply be augmented to include this information.

5.6 EVALUATION

Our evaluation of GALETTE focused on the following research questions:

- **RQ1**: How well does GALETTE preserve program semantics?
- **RQ2**: How accurately does GALETTE track taint tags?
- **RQ3**: What is the runtime overhead of GALETTE?
- **RQ4**: What is the memory overhead of GALETTE?

5.6.1 Methodology

Baselines

To evaluate these questions, we compared GALETTE against two state-of-the-art dynamic taint tracking systems for the JVM: PHOSPHOR (Bell and Kaiser, 2014) and MIRRORTAINT (Ouyang et al., 2023). We used the latest release of PHOSPHOR (version 0.1.0) (Bell and Phosphor Contributors, 2024) and the latest commit of MIRRORTAINT (commit 54fcbfc) (MirrorTaint Contributors, 2023).

PHOSPHOR uses shadowing and requires the JCL to be instrumented (Bell and Kaiser, 2014). For the reasons discussed in Section 5.2.4, this instrumentation must be applied statically to certain parts of the JCL, and this static instrumentation violates constraints imposed by the JPMS. To evaluate PHOSPHOR on

Java 9+ JDKs, we worked with the maintainers of PHOSPHOR to create custom jlink plugins, similar to the ones we created for GALETTE, for PHOSPHOR that implement the approach described in Section 5.3.1. For all experiments, PHOSPHOR was configured with the options "forceUnboxAcmpEq", "withEnumsByValue", and "serialization". These options were selected based on the advice of the maintainers of PHOSPHOR.

MIRRORTAINT uses mirroring and does not support statically instrumenting the JCL. Instead, MIRRORTAINT relies upon manually defined propagation rules for commonly used methods in the JCL (Ouyang et al., 2023). This approach is unsound—taint tags will not be propagated through methods that are not instrumented for which a propagation rule has not been defined. However, it allows MIRRORTAINT to run on Java 9+ without violating constraints imposed by the JPMS. We encountered some defects in the latest commit of MIRRORTAINT which prevented us from using the unmodified release in our evaluation. We made minor changes to MIRRORTAINT to correct these defects. This corrected version of MIRRORTAINT is what we used in our evaluation. We have contacted the authors of MIRRORTAINT about releasing these fixes.

Benchmarks

Our evaluation featured two benchmark suites: a functional and a performance benchmark suite. We used the functional benchmark suite for RQ1 and R2 which evaluate GALETTE's conformance with the functional requirements of a taint tracking system. We used the performance benchmark suite for RQ3 and RQ4 which evaluate GALETTE's runtime performance.

The functional benchmark suite consists of a set of deterministic, synthetic programs designed to exercise both newer and older Java features. To identify deviations from expected program behavior and measure the accuracy of taint tag propagation, we needed a ground truth for both the expected behavior of the program and the expected propagated taint tags for each benchmark program. As noted by Pauck et al. (2018), the ground truth for expected taint tag propagation needs to be manually determined. However, it is challenging to manually determine the ground truths for arbitrary, real-world programs. Therefore, we manually constructed synthetic programs for which we could manually determine the ground truth expected taint tag propagation. We grouped these programs by the type of functionality that they exercise. A detailed listing of these benchmark groups is shown in Table 10. Each synthetic program contains handwritten assertions which check that the program has executed normally; these assertions are intended to always pass on a Java runtime that has not been instrumented. For each program, we manually apply taint tags to selected input values and determine the set of expected taint tags for selected values computed in the program.

The functional benchmarks were compiled using OpenJDK Temurin (version 17.0.10+7). We configured `javac` to generate class files for Java 8 for every benchmark group except for the "String Indy Concat" and "Record Type". These groups evaluate Java features not available for Java 8 class files. The "String Indy Concat" group was compiled to Java 9; this group tests the improved string concatenation introduced in Java 9 which uses the `invokedynamic` (Oracle Corporation, 2024c). The "Record Type" group was compiled to Java 16; this group tests the record types introduced in Java 16 (Oracle Corporation, 2024d).

For the performance benchmark suite, we used the third major release of the DaCapo benchmark suite (version dacapo-23.11-chopin) (DaCapo Project Contributors, 2023b). The DaCapo benchmark suite is a well established suite of JVM performance benchmarks first proposed by Blackburn et al. (2006). The DaCapo benchmark suite (version dacapo-23.11-chopin) consists of non-trivial workloads for 22 real-world, open-source applications (Blackburn et al., 2006; DaCapo Project Contributors, 2023b). We excluded two of these benchmarks, "cassandra" and "kafka", due to failures that we observed when executing these benchmarks on a Java runtime that had not been instrumented. Of the remaining twenty benchmarks,

Table 10: Functional Benchmarks. For each benchmark group (Group), we report the total number of tests in the group (#), and a description of the type of functionality exercised by tests in that group (Description).

Group	#	Description
Array Access	74	Loads and stores elements from arrays of various types, lengths, and shapes. Checks taint tag propagation to and from array elements and indices.
Array Length	5	Creates arrays of various types, lengths, and shapes. Checks taint tag propagation to and from array lengths.
Array Reflection	75	Uses `java.lang.reflect.Array` to create arrays, load array elements, and store array elements. Checks taint tag propagation to and from array elements, indices, and lengths.
Assignment	12	Assigns values of various types to local variables. Checks taint tag propagation to and from local variables.
Boxed Type	8	Creates various boxed primitive type (e.g., `java.lang.Integer`) instances. Checks taint tag propagation through the boxing and unboxing of primitive values.
Class Reflection	7	Checks the correctness of class' attributes inspected using reflection.
Collection	6	Checks taint tag propagation through operations on common JCL collections (e.g., `java.util.ArrayList`).
Conditional	4	Checks taint tag propagation in the presence of conditional branches.
Constructor Reflection	88	Invokes various constructors using reflection. Checks taint tag propagation through the constructors' arguments.
Field	6	Loads and stores values from fields of various types. Checks taint tag propagation to and from the fields.
Field Reflection	176	Loads and stores values from fields of various types using reflection. Checks taint tag propagation to and from the field.
Jdk Unsafe	322	Uses `jdk.internal.misc.Unsafe` to load and store array elements and fields of various types using various access semantics. Checks taint tag propagation to and from the accessed values.
Lambda	7	Invokes various lambda expressions. Checks taint tag propagation through the expressions' arguments and return values.
Loop	4	Checks taint tag propagation in the presence of various looping constructs.
Method Call	14	Invokes various methods. Checks taint tag propagation through the methods' arguments and return values.
Method Handle	40	Uses `java.lang.invoke.MethodHandle` instances to invoke various methods. Checks taint tag propagation through the methods' arguments and return values.
Method Handle 9+	13	Uses `java.lang.invoke.MethodHandle` instances created using methods added to the JCL in Java 9 to invoke various methods. Checks taint tag propagation through the methods' arguments and return values.
Method Reflection	213	Invokes various methods using reflection. Checks taint tag propagation through the methods' arguments and return values.
Record Type	68	Creates record type instances with various types of components. Checks taint tag propagation through the records' components and methods.
Static Initializer	2	Checks taint tag taint tag propagation through static method calls defined in classes that have not yet been initialized.
String	26	Checks taint tag propagation through operations on `java.lang.String`.
String Builder Concat	37	Checks taint tag propagation through pre-Java 9 string concatenation which uses `java.lang.StringBuilder`.
String Indy Concat	37	Checks taint tag propagation through Java 9+ string concatenation which uses `invokedynamic`.
Sun Unsafe	270	Uses `sun.misc.Unsafe` to load and store array elements and fields of various types using various access semantics. Checks taint tag propagation to and from the accessed values.
Throwable	5	Catches explicitly thrown exceptions. Checks taint tag propagation from the thrown exception to the caught exception.
Var Handle	1932	Uses `java.lang.invoke.VarHandle` instances to load and store array elements and fields of various types using various access semantics. Checks taint tag propagation to and from the accessed values.

eight required Java version 11 or greater; the rest are compatible with Java version 8 or greater (DaCapo Project Contributors, 2023a). We used the "small" workload size for each benchmark. The timeouts used in the benchmarks were increased by a factor of one thousand to give adequate time for the slower taint tracking systems to complete tasks. We also modified two of the benchmarks, "tradebeans" and "tradesoap", to remove problematic timeouts. We did not apply taint tags to any values for these benchmarks. Generally, overheads for taint tracking systems will be higher in the presence of tainted data.

Metrics

For RQ1 and RQ2, we ran each functional benchmark program with each taint tracking system on JDKs for the four active LTS versions of Java (8, 11, 17, and 21). If a program uses features that are not available on a specific JDK, then that program was skipped on that JDK. For each program execution, we recorded the outcome as either passing, failing due to deviations from the original program semantics, or failing due to incorrect taint tag propagation. If an unexpected crash, exception, timeout, or assertion failure occurred during the execution of the program, then that execution was marked as failing due to deviations from the original program semantics. Otherwise, if the set of taint tags reported by the taint tracking system did not match our manually determined set of expected taint tags, then the execution was marked as failing due to incorrect taint tag propagation. All other executions were marked as passing.

For RQ3 and R4, we collected measurements on each DaCapo benchmark for four treatments: without tainting, with Galette, with Phosphor, and with MirrorTaint. For each treatment on each DaCapo benchmark, we performed five warm-up iterations, which are discarded, followed by five measurement iterations on each of twenty JVM processes for a total of one hundred measurements. Warm-up iterations mitigate major performance fluctuations that occur when the JVM is first started due to dynamic optimizations like Just-in-Time (JIT) compilation (Georges et al., 2007; Barrett et al., 2017; Traini et al., 2023). For each measurement iteration, we record the execution time and the peak memory usage. The execution time is the elapsed real (wall-clock) time taken to complete the workload. Peak memory usage is computed by sampling the resident set size (RSS) of the Java process with a sample interval of one millisecond. The maximum RSS sampled during the iteration is recorded as the peak memory usage. We observed that on certain benchmarks the instrumentation used by some taint tracking systems resulted in infinite loops or deadlock. Therefore, we imposed an overall timeout of twenty-four hours for each DaCapo process. Each DaCapo benchmark performs a checksum-based “validity check” that confirms that the benchmark execution is correct (Blackburn et al., 2006). If any validity check failed, an unhandled exception was thrown, the DaCapo process crashed, or the twenty-four-hour timeout elapsed, we discarded all measurements for the process.

Experimental Setup

All experiments were conducted on a virtual machine with four 2.6 GHz AMD EPYC 7H12 vCPUs, with sixteen GB of RAM, running Ubuntu 20.04.3. For RQ1 and RQ2, we used OpenJDK Temurin versions 1.8.0_402-b06, 11.0.22+7, 17.0.10+7, and 21.0.2+13. For RQ3 and RQ4, we used OpenJDK Temurin version 11.0.22+7. For all experiments, we used the Java options `-Xmx6G` and `-ea` in addition to any options needed to use a particular taint tracking system.

5.6.2 Results

Functional

Table 11 reports the number of programs failing due to semantic and propagation failures in each benchmark group for each taint tracking system using each JDK. A total of 11,364 tests were attempted for each taint tracking system across twenty-six groups and four JDK versions.

For both Galette and MirrorTaint, we observed no semantic failures. For Phosphor, there were 7,214 semantic failures including all 6,902 tests for JDK versions 17 and 21. Even when using the custom jlink plugins described in Section 5.6.1, the Phosphor-instrumented JVMs for Java 17 and 21 crashed during

Table 11: Semantics Preservation and Propagation Accuracy. For each taint tracking system, we report the number of tests in each group (Group) that failed due to deviations from the original program semantics (Semantic) and incorrect taint tag propagation (Tag) on JDK versions 8, 11, 17, and 21. Next to each group, we list the total number of test cases in that group (#). Non-zero entries are colored red. Values are omitted (—) for a group on a JDK version if that group uses features that are unavailable on that JDK version.

		Galette								MirrorTaint								Phosphor							
		Semantic				Tag				Semantic				Tag				Semantic				Tag			
Group	#	8	11	17	21	8	11	17	21	8	11	17	21	8	11	17	21	8	11	17	21	8	11	17	21
Array Access	74	0	0	0	0	0	0	0	0	0	0	0	0	55	55	55	55	0	0	74	74	0	0	0	0
Array Length	5	0	0	0	0	0	0	0	0	0	0	0	0	3	3	3	3	0	0	5	5	0	0	0	0
Array Reflection	75	0	0	0	0	0	0	0	0	0	0	0	0	65	65	65	65	0	0	75	75	7	7	0	0
Assignment	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12	12	0	0	0	0
Boxed Type	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	8	0	0	0	0
Class Reflection	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7	7	0	0	0	0
Collection	6	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	0	0	6	6	0	0	0	0
Conditional	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	0	0	0	0
Constructor Reflection	88	0	0	0	0	0	0	0	0	0	0	0	0	12	12	12	12	18	18	88	88	0	0	0	0
Field	6	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	0	0	6	6	0	0	0	0
Field Reflection	176	0	0	0	0	0	0	0	0	0	0	0	0	31	31	31	31	0	0	176	176	0	0	0	0
Jdk Unsafe	322	—	0	0	0	—	0	0	0	—	0	0	0	—	213	213	213	—	0	322	322	—	27	0	0
Lambda	7	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	7	7	1	1	0	0
Loop	4	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	4	4	0	0	0	0
Method Call	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	14	0	0	0	0
Method Handle	40	0	0	0	0	0	0	0	0	0	0	0	0	34	34	34	34	3	3	40	40	36	36	0	0
Method Handle 9+	13	—	0	0	0	—	0	0	0	—	0	0	0	—	11	11	11	—	4	13	13	—	7	0	0
Method Reflection	213	0	0	0	0	0	0	0	0	0	0	0	0	25	25	25	25	38	38	213	213	1	1	0	0
Record Type	68	—	—	0	0	—	—	0	0	—	—	0	0	—	—	16	16	—	—	68	68	—	—	0	0
Static Initializer	2	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	2	2	0	0	0	0
String	26	0	0	0	0	0	0	0	0	0	0	0	0	13	13	13	13	0	0	26	26	2	1	0	0
String Builder Concat	37	0	0	0	0	0	0	0	0	0	0	0	0	19	19	19	19	0	0	37	37	6	6	0	0
String Indy Concat	37	—	0	0	0	—	0	0	0	—	0	0	0	—	19	19	19	—	0	37	37	—	19	0	0
Sun Unsafe	270	0	0	0	0	0	0	0	0	0	0	0	0	177	177	177	177	0	0	270	270	0	0	0	0
Throwable	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	5	2	2	0	0
Var Handle	1,932	—	0	0	0	—	0	0	0	—	0	0	0	—	1,353	1,353	1,353	—	190	1,932	1,932	—	726	0	0

initialization. Without these plugins, the Phosphor-instrumented JVM for Java 11 also crashed during initialization. The remaining semantic failures for Phosphor were in a total of five groups each of tested either reflection or signature polymorphic methods. In several of these cases, Phosphor's use of array wrappers to store taint tags caused the JVM to crash.

There were no observed propagation failures for Galette. There were 6,584 propagation failures for MirrorTaint from twenty unique groups of the twenty-six total functional groups. For Phosphor, we observed 888 propagation failures from eleven unique groups. Both systems performed particularly poorly on groups that tested signature polymorphic methods: "Lambda", "Method Handle", "Method Handle 9+", "Record Type", "String Indy Concat" and "Var Handle". There were 4,321 and 826 propagation failures for MirrorTaint and Phosphor in these groups, respectively. Most of the propagation failures for MirrorTaint were due to incorrect or missing modeling of JVM semantics which as discussed in Section 5.2.2 is a limitation of mirroring. The propagation failures for Phosphor were generally due to an inability to track the flow of information through signature polymorphic method calls and missing special handling for parts of the Reflection and Unsafe APIs.

Overall, we found that Phosphor was prone to introducing deviations into original programs' semantics particularly in the presence of modern Java features like signature polymorphism. On JDK versions 17 and 21, these deviations prevented the JVM from successfully initializing. By contrast, there were no semantic failures for MirrorTaint. However, MirrorTaint failed to accurately propagate taint tags in tests from twenty of the twenty-six total functional groups. GALETTE was able to accurately propagate taint tags without introducing deviations into programs' semantics.

Performance

Table 12 reports the execution time and peak memory usage for each taint tracking system and the baseline (without taint tracking) on each performance benchmark. For the baseline, we report the median value. For each taint tracking system, we report the overhead compared to the baseline. Overhead was computed as $\frac{\tilde{x_t}-\tilde{x}}{\tilde{x}}$, where $\tilde{x_t}$ was the median value for the taint tracking system and $\tilde{x}$ was the median value for the baseline.

Table 12: Execution Time and Peak Memory Usage. For the baseline (Base), we report the median ($\tilde{x}$) execution time in milliseconds (left) and peak memory usage in megabytes (right) on each benchmark. For each taint tracking system, we report the overhead (OV) for execution time and peak memory usage relative to the baseline. For each reported overhead, we also report the lower confidence limit (LCL) and upper confidence limit (UCL) of a two-tailed, bias-corrected 95% bootstrap confidence interval. We used one thousand resamples to compute each confidence interval. For MIRRORTAINT and PHOSPHOR, values that are statistically significantly greater than or less than GALETTE's are colored green and red, respectively.

	Execution Time										Peak Memory Usage									
	BASE	GALETTE			MIRRORTAINT			PHOSPHOR			BASE	GALETTE			MIRRORTAINT			PHOSPHOR		
	$\tilde{x}$	OV	LCL	UCL	OV	LCL	UCL	OV	LCL	UCL	$\tilde{x}$	OV	LCL	UCL	OV	LCL	UCL	OV	LCL	UCL
avrora	2,360	0.90	0.84	0.98	662.53	628.58	689.92	1.59	1.47	1.76	126.89	0.71	0.44	0.83	16.72	15.83	17.98	1.56	1.45	1.74
batik	272	11.30	10.33	12.21	2,908.87	2,715.91	3,101.19	—	—	—	226.22	1.48	1.40	1.56	9.25	8.96	9.54	—	—	—
biojava	134	29.28	27.82	30.31	2,642.57	2,421.95	2,781.04	15.32	14.69	16.22	171.58	2.34	2.27	2.42	4.01	3.94	4.06	1.30	1.26	1.33
eclipse	628	1.68	-0.49	3.33	—	—	—	—	—	—	295.46	0.99	0.83	1.25	—	—	—	—	—	—
fop	110	3.01	2.57	3.50	—	—	—	5.64	4.84	6.22	144.34	1.10	1.08	1.13	—	—	—	2.25	2.21	2.28
graphchi	504	5.40	5.12	5.77	—	—	—	13.81	13.17	14.51	404.80	1.27	1.12	1.36	—	—	—	0.36	0.34	0.38
h2	134	4.40	4.06	4.71	866.75	834.94	898.80	7.69	7.22	8.31	330.46	1.15	1.11	1.19	1.26	1.23	1.30	1.49	1.46	1.52
h2o	600	16.17	15.59	17.38	—	—	—	—	—	—	393.08	0.75	0.72	0.80	—	—	—	—	—	—
jme	1,022	3.16	2.86	3.47	3,916.11	3,730.04	4,183.92	3.65	3.41	3.96	265.96	1.64	1.52	1.75	8.34	7.89	8.74	1.27	1.16	1.38
jython	381	10.37	9.81	10.93	—	—	—	—	—	—	420.20	6.50	6.28	6.76	—	—	—	—	—	—
luindex	1,100	5.32	2.54	7.18	—	—	—	10.05	5.19	13.37	226.56	0.86	0.81	0.90	—	—	—	1.54	1.49	1.59
lusearch	182	9.59	8.61	10.25	3,630.84	3,320.09	3,825.41	8.50	7.62	9.12	216.05	2.29	2.10	2.45	2.72	2.61	2.85	0.93	0.89	1.01
pmd	98	2.88	2.56	3.29	—	—	—	—	—	—	137.40	0.88	0.85	0.92	—	—	—	—	—	—
spring	98	1.68	1.52	1.86	—	—	—	4.62	4.23	5.04	408.66	1.01	0.99	1.03	—	—	—	1.67	1.65	1.70
sunflow	285	14.18	13.52	15.05	27,399.37	26,446.86	27,774.67	16.94	16.33	17.42	166.03	1.18	1.12	1.24	7.21	7.03	7.44	1.15	1.12	1.20
tomcat	352	4.03	3.58	4.38	63.63	60.30	68.10	—	—	—	280.74	1.49	1.45	1.54	3.88	3.54	4.13	—	—	—
tradebeans	162	6.90	6.48	7.49	—	—	—	—	—	—	607.82	1.40	1.36	1.46	—	—	—	—	—	—
tradesoap	453	8.74	7.98	9.42	—	—	—	—	—	—	681.13	1.30	1.25	1.32	—	—	—	—	—	—
xalan	106	5.29	4.46	5.90	—	—	—	4.81	3.90	5.55	132.14	0.82	0.78	0.90	—	—	—	1.64	1.59	1.68
zxing	162	3.57	3.36	3.85	8,252.57	7,972.77	8,712.91	—	—	—	108.98	1.56	1.52	1.63	36.35	35.40	37.87	—	—	—

The missing entries in Table 12 indicate cases where we were unable to collect valid measurements for a tool on a benchmark due to failures. In all such cases, we were able to consistently reproduce these failures. There were nine benchmarks for which we could not collect results for PHOSPHOR. On seven of these benchmarks, PHOSPHOR's instrumentation induced a deviation from the benchmark's original semantics that produced an exception or error. On the remaining two benchmarks, "tradebeans" and "tradesoap", PHOSPHOR's instrumentation induced a deviation from the benchmark's original semantics that produced an infinite loop. There were eleven benchmarks for which we could not collect results for MIRRORTAINT. On ten of these benchmarks, MIRRORTAINT's instrumentation induced a deviation from the benchmark's original semantics that produced an exception or error. On the remaining benchmark, "graphchi", the JVM ran out of heap space when run with MIRRORTAINT. As mentioned in Section 5.6.1, we used the Java option `-Xmx6G` to specify a maximum heap size of six gigabytes. GALETTE was able to run all twenty of the benchmarks successfully.

Overall, the mean execution time overheads for GALETTE, PHOSPHOR, and MIRRORTAINT across all benchmarks were 7.393, 8.420, and 5593.692, respectively. The mean memory overheads for GALETTE, PHOSPHOR, and MIRRORTAINT across all benchmarks were 1.536, 1.379, and 9.971, respectively.

We performed Mann-Whitney U tests to compare the overhead of GALETTE with respect to execution time and memory usage against the other taint tracking systems on each benchmark. A continuity correction of 0.5 was used when calculating two-tailed, asymptotic p-values. Following current best practices as described by Arcuri and Briand (2014), a base significance level of 0.05 was adjusted for three comparisons resulting in a Bonferroni-corrected significance level of 0.0167 per test. We used the Vargha-Delaney $\hat{A}_{12}$ statistic (Vargha and Delaney, 2000) to quantify effect sizes for these comparisons. The effect size for all comparisons marked as statistically significant in Table 12 was large ($\hat{A}_{12} \geq 0.71$) except for the comparison between GALETTE and PHOSPHOR on "jme" with respect to execution time which had a medium effect size ($\hat{A}_{12} \geq 0.64$). The full results of these statistical tests are shown in Table 13.

Table 13: Execution Time and Peak Memory Usage Statistical Tests. We report the two-tailed, asymptotic p-value (p) and the Vargha-Delaney $\hat{A}_{12}$ statistic ($\hat{A}_{12}$) for Mann-Whitney U tests comparing between the execution time (left) and peak memory usage (right) of GALETTE against MIRRORTAINT and PHOSPHOR. Values that are statistically significantly ($p < 0.0167$) greater than or less than GALETTE's are colored green and red, respectively.

	Execution Time				Peak Memory Usage			
	MirrorTaint		Phosphor		MirrorTaint		Phosphor	
	p	$\hat{A}_{12}$	p	$\hat{A}_{12}$	p	$\hat{A}_{12}$	p	$\hat{A}_{12}$
avrora	$2.562 \cdot 10^{-34}$	1.000	$1.125 \cdot 10^{-33}$	0.995	$2.562 \cdot 10^{-34}$	1.000	$5.928 \cdot 10^{-34}$	0.997
batik	$2.562 \cdot 10^{-34}$	1.000	—	—	$2.562 \cdot 10^{-34}$	1.000	—	—
biojava	$2.561 \cdot 10^{-34}$	1.000	$2.560 \cdot 10^{-34}$	1.000	$2.562 \cdot 10^{-34}$	1.000	$2.562 \cdot 10^{-34}$	1.000
eclipse	—	—	—	—	—	—	—	—
fop	—	—	$5.930 \cdot 10^{-22}$	0.894	—	—	$2.562 \cdot 10^{-34}$	1.000
graphchi	—	—	$3.672 \cdot 10^{-34}$	0.999	—	—	$2.562 \cdot 10^{-34}$	1.000
h2	$2.560 \cdot 10^{-34}$	1.000	$9.947 \cdot 10^{-31}$	0.972	$6.592 \cdot 10^{-16}$	0.831	$1.046 \cdot 10^{-32}$	0.988
h2o	—	—	—	—	—	—	—	—
jme	$2.562 \cdot 10^{-34}$	1.000	$1.021 \cdot 10^{-5}$	0.681	$2.562 \cdot 10^{-34}$	1.000	$5.236 \cdot 10^{-20}$	0.875
jython	—	—	—	—	—	—	—	—
luindex	—	—	$1.968 \cdot 10^{-13}$	0.801	—	—	$4.530 \cdot 10^{-34}$	0.998
lusearch	$2.561 \cdot 10^{-34}$	1.000	$1.617 \cdot 10^{-7}$	0.714	$5.535 \cdot 10^{-21}$	0.885	$2.562 \cdot 10^{-34}$	1.000
pmd	—	—	—	—	—	—	—	—
spring	—	—	$1.753 \cdot 10^{-33}$	0.994	—	—	$2.562 \cdot 10^{-34}$	1.000
sunflow	$2.561 \cdot 10^{-34}$	1.000	$2.064 \cdot 10^{-25}$	0.926	$2.562 \cdot 10^{-34}$	1.000	$6.770 \cdot 10^{-1}$	0.517
tomcat	$2.561 \cdot 10^{-34}$	1.000	—	—	$2.562 \cdot 10^{-34}$	1.000	—	—
tradebeans	—	—	—	—	—	—	—	—
tradesoap	—	—	—	—	—	—	—	—
xalan	—	—	$7.212 \cdot 10^{-2}$	0.574	—	—	$2.562 \cdot 10^{-34}$	1.000
zxing	$2.560 \cdot 10^{-34}$	1.000	—	—	$2.562 \cdot 10^{-34}$	1.000	—	—

We found that the overhead of GALETTE was statistically significantly less than MIRRORTAINT with respect to both execution time and memory usage on all benchmarks. This supports the claim made in prior work that using a mapping structure, like the MIRRORTAINT's mirrored heap, to store taint tags is slower than using shadow variables (Bell and Kaiser, 2014). GALETTE's overhead was statistically significantly less than PHOSPHOR's on eight benchmarks and greater than PHOSPHOR's on two benchmarks with respect to execution time. GALETTE's overhead was statistically significantly less than PHOSPHOR's on six benchmarks and greater than PHOSPHOR's on four benchmarks with respect to memory usage. By using a mirrored mapping structure to store only the taint tags associated with arrays, GALETTE was able to achieve a performance profile that was competitive with that of PHOSPHOR, which uses pure shadowing.

5.6.3 Threats to Validity

Our functional benchmark suite included only synthetic programs and may not be representative of all JVM programs. As noted in Section 5.6.1, we needed to use synthetic benchmarks in order to determine the ground truth expected taint tag propagation for each benchmark. Furthermore, our suite does not cover all the functionality offered by the JVM. However, we tried to ensure that common functionality like array operations and method calls were well tested. We also tried to cover new Java features like record types and variable handles particularly thoroughly.

The ground truth expected taint tag propagation for each functional benchmark was manually determined and, therefore, could be incorrect. However, we constructed each functional benchmark so that the expected taint tag propagation was easy to follow and double-checked benchmarks on which at least one taint tracking system reported incorrect labels.

Our performance benchmark suite included only twenty workloads and may not be representative of all workloads. However, it is a well-established benchmark suite constructed for performance benchmarking in the JVM.

Our choice for the number of warm-up iterations in performance trials may not have been large enough to fully mitigate performance fluctuations that occur when the JVM is first started. To further mitigate the effect of these fluctuations, we took repeated measurements using multiple JVM invocations and report confidence intervals across these measurements in accordance with the recommendations of Georges et al. (2007).

Our choice of a twenty-four-hour timeout for the performance trials may have prevented measurements from being recorded for some taint tracking systems. We manually analyzed benchmark-tool combinations that timed out and determined that these processes were stuck in infinite loops and would not complete if given more time. However, our choice of timeout duration may affect experiment results if trials are performed in a different environment.

Our choice of maximum heap size, six gigabytes, prevented measurements from being recorded on "graphchi" for MIRRORTAINT. Additionally, the maximum heap size can affect the frequency and duration of garbage collection in the JVM impacting execution times.

5.7 CONCLUSION

Existing approaches to dynamic taint tracking in the JVM perform poorly on modern JVMs due to signature polymorphism and the JPMS. GALETTE, our approach for dynamic taint tracking in the JVM, provides precise, robust taint tag propagation in the presence of modern Java features. GALETTE is able to trace the flow of information through signature polymorphic methods by using a thread-local frame store. By modifying the primordial module, GALETTE is able to circumvent constraints imposed by the JPMS. In our empirical evaluation, we found that GALETTE was able to propagate taint tags with perfect accuracy while preserving the original semantics of programs. We also found that GALETTE's runtime and memory overheads were significantly less than MIRRORTAINT's overheads and competitive with PHOSPHOR's overheads.

5.8 STATUS

This work was submitted to the *ACM International Conference on the Foundations of Software Engineering* (FSE 2025). Supplemental materials containing GALETTE's source code, experimental scripts, raw experimental data, and the full results of the statistical tests that we conducted in our evaluation of this work are available at `https://figshare.com/s/b73c3c08400e2fda8977`.

6 RELATED WORK

6.1 TESTING FOR INJECTION VULNERABILITIES

A variety of testing-based approaches have been used to identify injection vulnerabilities. Kieżun et al. (2009) detect SQLi and XSS vulnerabilities in web applications using symbolic execution to explore application states and dynamic taint tracking to identify user-controlled inputs that flow to sensitive program locations. Halfond et al. (2006) provide an automated technique for detecting and preventing SQLi attacks using "positive-tainting". Positive-tainting tracks the flow of trusted values opposed to the more common approach of "negative" tainting which tracks untrusted values. Alhuzali et al. (2018) use concolic execution to generate sequences of HTTP requests that reach vulnerable sinks. Mohammadi et al. (2017) use a grammar-based approach to generate XSS attack strings for test inputs. Simos et al. (2019) combine a grammar-based approach for generating SQLi attack strings with combinatorial testing to detect SQLi vulnerabilities. Thomé et al. (2014) generate inputs to trigger SQLi vulnerabilities using evolutionary fuzzing.

6.2 TAINT TRACKING IN FUZZING

Taint tracking has been used by fuzzing approaches to generate "interesting" inputs capable of finding bugs deep in a program's execution. Ganesh et al. (2009) use taint tracking to target input bytes that flow to a sink and replace those bytes with large, small, and zero-valued integers. Rawat et al. (2017) record values that fuzzer-derived values are compared against and use them as inputs. Similarly, Wang et al. (2010) trace flows from fuzz-derived values to checksum-like routines. Knowledge of these checksum bytes is then used to generate new inputs that satisfy input constraints. Kukucka et al. (2022) propose "hinting", a form of intelligent mutation for fuzzing. Hints are identified based on comparisons against the input using concolic execution and taint tracking.

6.3 PARAMETRIC FUZZING

Padhye et al. (2019b) introduce the concept of parametric fuzzing, leveraging input-validity and coverage feedback to guide input generation. Reddy et al. (2020) use reinforcement learning to guide parametric generators to produce a diverse set of valid inputs. Nguyen and Grunske (2022) perform structural-aware mutation which distinguishes between structure-changing and structure-preserving mutations to increase the behavioral diversity of inputs generated by a parametric fuzzer. Li et al. (2024) investigate the "havoc effect" in parametric fuzzing where small changes to a parameter sequence produce large changes to the structure generated from it. Lampropoulos et al. (2019) propose an alternative approach for guided and generator-based fuzzing that uses type-aware mutation operators instead of mutating a parameter sequence. These mutation operators are automatically synthesized based on input types, allowing the fuzzer to mutate inputs at the algebraic datatype.

6.4 SPECIFICATION-BASED FUZZING

Specification-based fuzzers use a knowledge of an input's expected structure to generate valid inputs. Holler et al. (2012) learn code fragments from a corpus of seed inputs using a context-free grammar for the input structure. Then, they modify inputs by randomly replacing fragments in the input with learned fragments of the same type. Wang et al. (2019) and Aschermann et al. (2019) incorporate coverage feedback into grammar-based fuzzing by employing tree-based mutation. Srivastava and Payer (2021) improve upon Aschermann et al. (2019)'s coverage-guided, grammar-aware fuzzing approach by introducing larger, more "aggressive" mutations and restructuring input grammars' production rules to eliminate sampling bias. Pham et al. (2021) use file format specifications to parse inputs into a virtual structure, a tree of file chunks. They then define structural mutation operators that operate on a file's virtual structure.

6.5 INFERRING INPUT STRUCTURE

You et al. (2019b) use a dynamic probing strategy to identify fields, regions of linked bytes, by observing the effect of applied mutations while fuzzing. Identified fields are then mutated using type-specific mutation strategies. You et al. (2019a) identify input validity checks on portions of an input and employ targeted mutation strategies to satisfy these checks. Blazytko et al. (2019) infer structural properties of input formats over the course of a fuzzing campaign using code coverage feedback. Mathis et al. (2020) use dynamic taint tracking to infer lexical tokens and generate seed inputs for an input language.

6.6 DEBUGGING ASSISTANCE

Zeller and Hildebrandt (2002) present "delta debugging", a general algorithm for minimizing failing inputs. Herfert et al. (2017) propose an extension to delta debugging that is more effective at reducing the size of tree-structured inputs. Clause and Orso (2009) use taint tracking to identify the subset of failure-inducing inputs that are failure-relevant to assist with program debugging. Attariyan and Flinn (2010) localize the root cause of configuration errors using dynamic taint tracking.

Weiser (1984) propose "dynamic slicing" which computes the subset of program statements that affect values at a particular program point for a particular program execution or executions. This subset is referred to as a "slice". Dynamic slicing has been used to support automated debugging and program understanding tasks (Weiser, 1984; Tip, 1995). Zhang et al. (2006) improve upon traditional dynamic slicing using a heuristic that considers the correctness of outputs computed using a statement to remove statements that are unlikely to be related to a fault from computed slices. A related technique, thin slicing, was proposed by Sridharan et al. (2007). Thin slicing only includes statements which are part of some sequence of assignments that compute and copy a value to a target location in the slice for the target location (Sridharan et al., 2007).

6.7 TRACKING CONTROL-FLOW RELATIONSHIPS

Some dynamic taint tracking systems support the standard semantics for tracking control flow discussed in Section 3.2 (Chandra and Franz, 2007; Clause et al., 2007; Bell and Kaiser, 2015). Several of these systems attempt to address control-flow-related over-tainting. Bao et al. (2010) propose propagating control flows only along branches that correspond to equivalence checks. Kang et al. (2011) put forward a similar approach, but instead target branches related to information-preserving transformations. These branches are determined by analyzing execution traces to find control flow paths that can only be reached by a single input value. Attariyan and Flinn (2010) mitigate over-tainting from control flows by reducing the weight of a taint tag when it is propagated via a control flow. Cox et al. (2014) address control-flow-related over-tainting by quantifying the amount of information revealed by control flows. This quantity is then used to decide whether to propagate taint tags along a control flow.

6.8 DYNAMIC TAINT TRACKING FOR THE JAVA VIRTUAL MACHINE

Franz et al. (2005) trace relationships between strings in the JVM by instrumenting string-related classes. Halfond et al. (2006) also track relationships between strings by replacing instances of string-related classes with instances of corresponding meta-classes. These meta-classes are used to store and propagate taint tags. Chin and Wagner (2009) improve upon these approaches by allowing individual characters in a string to be tainted and traced independently. These approaches are unable to track non-string values.

Nair et al. (2008) perform dynamic taint tracking by modifying the Kaffe JVM interpreter. They extended the JVM stack, objects, and arrays to add space for storing associated taint tags. Enck et al. (2010a) instrument the Dalvik Virtual Machine interpreter to store and propagate taint tags. Taint tags are stored adjacent to original values in Dalvik's internal data structures. These approaches are able to perform fine-grained taint tracking on all forms of data. However, they rely on interpreter-specific properties and, therefore, are difficult to port to commodity JVMs.

Bell and Kaiser (2014) and Perkins et al. (2020) propose shadowing which uses Java bytecode instrumentation to store and propagate taint tags on all forms of data by directly shadowing original values and instructions. Wang et al. (2022) extend Bell and Kaiser (2014)'s approach to support inter-node taint tracking in JVM-based distributed systems. Ouyang et al. (2023) propose mirroring which uses a separate mirror-space to store and propagate taint tags. Mirroring and shadowing are able to port to different JVMs, because they do not rely on interpreter-specific properties. However, neither mirroring nor shadowing address signature polymorphism or the JPMS.

6.9 DYNAMIC TAINT TRACKING FOR OTHER MANAGED RUNTIMES

Karim et al. (2020) present a platform-independent approach for dynamic taint tracking in JavaScript that instruments applications to omit instructions for an abstract machine that encode the flow of information. When the application terminates, the omitted instructions are executed to trace the flow of information. Aldrich et al. (2023) improve upon Karim et al. (2020)'s approach to add support for `async` and `await`. Eghbali and Pradel (2022) propose an instrumentation-based, general-purpose dynamic analysis framework for Python and use this framework to build a dynamic taint analysis for Python. Son et al. (2013) track the flow of string values in PHP applications using a PHP interpreter extension.

6.10 EVALUATING TAINT TRACKING SYSTEMS

Jee (2015) examines system outputs when given different input values to determine if taint tags should propagate from the inputs to the outputs. Differences in the outputs for different input values are interpreted as meaning that taint tags from the input should have propagated to the output. By contrast, Pauck et al. (2018)'s framework for comparing Android taint analysis tools, requires the ground truth for test cases to be manually classified. Pauck et al. (2018) note that this manual determination of the ground truth is necessary because, "tools that could potentially be used to derive the ground truth are at the same time the tools we want to evaluate." Staicu et al. (2019) investigate the prevalence of implicit flows and the criticality of detecting implicit flows when using dynamic taint tracking to enforce security and privacy policies in JavaScript applications. They conclude that it is sufficient to consider only data flows in order to detect security-related source-to-sink flows, but in order to discover privacy-related source-to-sink flows, implicit flows also needed to be considered. Clause et al. (2007) explore the relationship between different taint propagation approaches and the amount of memory tainted, finding that propagating control flows resulted in significantly more tainted memory.

7 CONCLUSION

In this dissertation, we present four techniques for improving software testing by leveraging a knowledge of dynamic data relationships. In this final chapter, we first summarize the main findings of those four works. Next, we reflect on the research presented in this dissertation and discuss directions for future research into the use of dynamic relationships in software testing. Finally, we conclude by revisiting the thesis of this work: knowledge of dynamic data relationships can be leveraged to amplify existing software tests to reduce the amount of developer effort and expertise needed to detect and correct defects.

7.1 REVEALING INJECTION VULNERABILITIES BY LEVERAGING EXISTING TESTS

In Chapter 2, we present RIVULET, a novel approach for detecting injection vulnerabilities in web applications that leverages dynamic taint tracking. RIVULET monitors the execution of existing functional tests in order to detect information flows that may be vulnerable to attack, i.e., flows from untrusted system inputs to sensitive program locations. To determine whether a detected flow is genuinely vulnerable to attack or adequately neutralized by the application under analysis, RIVULET reruns the functional test used to detect the flow. During the test rerun, RIVULET performs precise, targeted mutation of test inputs, enabling the existing functional test to be re-used to detect injection vulnerabilities. By leveraging the context of how user-controlled values are used in security-sensitive parts of the application, RIVULET is able to limit the number reruns needed to verify detected flows.

In our empirical evaluation, we found that RIVULET's contextual rerun generation resulted in between 3.12 and 21.71 times fewer reruns being needed to analyze an application. We also compared RIVULET to a state-of-the-art static vulnerability detector, JULIA (Spoto et al., 2019), on several established benchmark suites and found that RIVULET was more precise and specific than JULIA. When applied to the version of Apache Struts exploited in the 2017 Equifax attack, RIVULET quickly identified the vulnerability, leveraging only the tests that existed in Struts at that time. RIVULET also discovered six previously unknown injection vulnerabilities in two open-source projects.

7.2 A PRACTICAL APPROACH FOR DYNAMIC TAINT TRACKING WITH CONTROL-FLOW RELATIONSHIPS

In Chapter 3, we describe control flows—a type of information flow in which information is passed indirectly between values via the control structure of the program. In our empirical evaluation, we confirmed the results of prior work, finding that it is essential to track control-flow relationships for certain downstream analyses that rely on dynamic taint tracking. We also found that tracking control flows using the standard semantics, as described in 3.2, resulted in significant over-tainting, dramatically reducing the precision of the taint tracking system. We show that the standard, post-dominator-based definition for the scope of a control flow is overly conservative. We also show that looping structures can produce over-tainting by introducing multiple program paths along which an effect can occur. To address these sources of imprecision, we introduce Conflux, alternative semantics for propagating taint tags along control flows. Conflux reduces over-tainting by decreasing the scope of control flows and providing a dynamic heuristic for reducing loop-related over-tainting.

We performed a case study exploring the effect of Conflux on a concrete application of taint tracking, automated debugging. By using its alternative semantics to trace relationships between failure-inducing inputs and thrown exceptions, Conflux was better able to identify failure-relevant inputs than the standard semantics. In addition to this case study, we evaluated Conflux's accuracy using a novel benchmark consisting of popular, real-world programs. We compared Conflux against existing taint propagation policies, including a state-of-the-art approach for reducing control-flow-related over-tainting, finding that Conflux had the highest F1 score on forty-three out of the forty-eight total tests.

7.3 CROSSOVER IN PARAMETRIC FUZZING

Chapter 4 introduces linked crossover, an approach that reasons about relationships between portions of input sequences to enable parametric fuzzers to more thoroughly exercise system behaviors. By dynamically tracking the execution behavior of parametric generators, linked crossover is able to intelligently select crossover points that preserve logical boundaries in inputs. In our empirical evaluation, we found that child parameter sequences produced using linked crossover inherited more desirable traits from their parents compared to those produced using one- and two-point crossover.

We compared linked crossover's performance on seven real-world Java projects against three state-of-the-art parametric fuzzers and two other forms of crossover on both long and short fuzzing campaigns. We found that linked crossover was demonstrably effective at discovering coverage-revealing inputs in both long and short campaigns. We also found that linked crossover had the highest detection rate for nine out of twelve defects on short fuzzing campaigns and eight out of twelve defects on long fuzzing campaigns. However, linked crossover was generally more effective on subjects using the JavaScript or Java class generator, possibly indicating that its efficacy may be impacted by either the nature of the generator or the input type itself.

7.4 DYNAMIC TAINT TRACKING FOR MODERN JAVA VIRTUAL MACHINES

In Chapter 5, we discuss the challenges of building a dynamic taint tracking system for modern JVMs and present Galette—a precise, robust approach for dynamic taint tracking in modern JVMs. Galette addresses limitations of existing approaches for dynamic taint tracking in the JVM, enabling tools that rely on dynamic taint tracking, like Rivulet and Conflux, to be built for the latest Java versions. On a benchmark suite of 3,451 synthetic Java programs, we found that Galette was able to propagate taint tags with perfect accuracy while preserving program semantics on all four active LTS versions of Java. We also found that Galette's runtime and memory overheads were competitive with that of two state-of-the-art dynamic taint tracking systems on a benchmark suite of twenty real-world Java programs.

7.5 DISCUSSION

In this section, we reflect on the research presented in this dissertation and discuss possible directions for future research on the use dynamic data relationships to improve software testing techniques. This discussion is focused on three major subjects. First, in Section 7.5.1, we talk about the application of the approaches presented in this manuscript for improving generated tests opposed to developer-written tests. Next, Section 7.5.2 describes common challenges that we faced while designing evaluations for dynamic analyses. Then, in Section 7.5.3, we discuss how the alternative control-flow semantics presented in Chapter 3 may be useful for addressing a limitation of evolutionary fuzzing that we encountered while working on the technique presented in Chapter 4. Finally, in Section 7.5.4, we explore the potential impact of dynamic dispatch on taint tracking and coverage-guided fuzzing.

7.5.1 Developer-Written Versus Generated Tests

The work presented in this manuscript primarily focuses on improving existing developer-written tests. However, a related line of research is improving automatically generated tests. Several automated techniques for generating tests have been proposed including Fraser and Arcuri (2011)'s search-based approach and Pacheco et al. (2007)'s feedback-driven approach. Studies have shown that these techniques are effective at finding defects in large-scale, real-world systems (Pacheco et al., 2008; Fraser and Arcuri, 2014). Recently, Large Language Models have also successfully been applied for test generation (Lemieux et al., 2023; Schäfer et al., 2023).

An interesting question for future research is whether the techniques presented in this dissertation are still effective when applied to generated tests instead of developer-written tests. In particular, combining automated test generation with RIVULET seems promising. RIVULET's vulnerability detection ability is independent of the quality of the original tests' oracles, since it replaces original test oracles with attack detectors. Test oracle generation is an open problem in automated test generation; therefore, RIVULET security-focused oracles may be particularly useful in generated tests. However, our original evaluation of RIVULET (Section 2.6) included only manually written tests. It is therefore unclear whether automated test generation approaches are capable of producing tests that perform the complex workflows needed to exercise vulnerable flows.

Similarly, our evaluation of ZEUGMA used only handwritten generators and fuzzing drivers. Recent work has shown that automatically generated fuzzing drivers can be effective at finding defects (Zhang et al., 2024; Babić et al., 2019). It is not clear whether linked crossover would be as effective on generated drivers. Another possible application of the work presented in Chapter 4 is the use of the heritability metrics presented in Section 4.6 to evaluate and improve generated fuzzing drivers. These heritability metrics could be used to identify and bias generation towards higher quality drivers. It may also be possible to create automated approaches for improving the heritability of a driver.

7.5.2 Selecting Evaluation Subjects and Baseline Tools

Since dynamic analyses monitor the behavior of a system at runtime, they require the system under analysis to be executed. As a result, when selecting subjects for use in the evaluations presented in this manuscript, we needed to ensure that selected systems could be reliably executed. We also needed to select systems for which we could either find or construct a representative workload. Additionally, the languages, frameworks, and libraries used by selected subjects need to be supported by the tools being evaluated. In practice, these constraints often restricted the pool of evaluation subjects that could be included in our experiments. Furthermore, it is possible that these constraints bias experimental results, producing a sample that is not representative of software in general.

We often needed to modify evaluation subjects, workloads, and even sometimes the tools being evaluated in order to perform our experiments. For instance, in our evaluation of Rivulet (Section 2.6), we needed to modify some of the benchmarks to fix incorrect SQL connection strings. In our evaluation of Galette (Section 5.6), we made minor fixes to MirrorTaint and major changes to Phosphor to ensure that they could run our evaluation subjects. For our evaluation of Conflux (Section 3.6), we had to fully implement Bao et al. (2010)'s approach for the JVM in order to have a suitable baseline for comparison. When evaluating Zeugma, we were unable to include both Confetti (Kukucka et al., 2022) and the latest release of Zest (Padhye et al., 2019b); Zest required Java version 9 or greater and Confetti only supports Java version 8. This lack of support for newer Java versions in taint-tracking-based tools like Confetti is what motivated our work on Galette.

One additional challenge, we encountered when selecting evaluation subjects was in obtaining ground truth data. Three of the four evaluations described in this manuscript—Sections 2.6, 3.6 and 5.6—feature experiments in which ground truth data was needed to analyze experimental results. In all three cases, it was infeasible to automatically derive and difficult to manually determine the ground truth for arbitrary, real-world programs. As a result, all three of the evaluations use two distinct sets of evaluation subjects. The first set was constrained to programs for which a ground truth could be constructed. The second set was not constrained in this way, allowing a variety of large, real-world programs to be included. However, a complete ground truth could not be constructed for the second set of subjects.

The challenges that we encountered in designing our evaluations are not limited to the work presented in this dissertation but affect evaluations of dynamic analyses in general. We feel that this underscores the importance of "meta-evaluations" which investigate the effectiveness, accuracy, and generalizability of evaluations of software engineering tools. For instance, Bundt et al. (2021) compared fuzzer performance on synthetic and organic bugs, finding that organic bugs were significantly more difficult for fuzzers to detect than the synthetic bugs often used to evaluate fuzzer performance.

Recent work has proposed tools which can assist researchers with the collection evaluation subjects and ground truth data. Pauck et al. (2018) present a framework that supports researchers conducting evaluations of taint tracking systems. Arteca and Turcotte (2022) present a tool that automatically downloads and runs JavaScript projects, allowing researchers to more easily identify projects that satisfy desired inclusion criteria.

7.5.3 Control-Flow Relationships in Evolutionary Fuzzing

As discussed in Chapter 4, evolutionary fuzzers collect feedback about system executions and use that information to bias the sampling of the input space towards known, interesting inputs. This biasing is what enables an evolutionary fuzzer to gradually evolve a corpus of behaviorally diverse inputs. As a result, the ability to make fine-grained, incremental progress towards new behaviors is critical for a fuzzer. Less sensitive forms of feedback, such as edge coverage, may be unable to identify small, incremental progress towards new execution behaviors. This insensitivity can cause the fuzzer to become stuck because it is unlikely to randomly jump to an input that demonstrates a new execution behavior, and it cannot gradually evolve its way towards new behaviors. Conversely, overly-sensitive feedback metrics, like path coverage, are impacted by the path explosion problem—it is infeasible to explore all possible execution paths. Thus, these metrics struggle to scale to real-world systems.

An interesting potential application of alternative control-flow semantics, such as those presented in Chapter 3, is towards solving this problem of enabling fuzzers to make progress when feedback insensitivity causes them to get stuck. Traditionally, dynamic taint tracking has been used in fuzzing to create smart mutation operators (Kukucka et al., 2022; Rawat et al., 2017). These approaches track data flows from fuzzer-controlled inputs into program branches and comparison operations. Observed flows are then used to perform targeted mutations on inputs. However, these approaches do not generally consider control-flow relationships, resulting in an incomplete view of the influence of fuzzer-controlled inputs on conditional branches. It is possible that a conditional branch is not data dependent on the input, but is still control dependent on the input; therefore, changes to the input may still affect the outcome of the branch. Using specialized semantics for reasoning about the effect of control-flow relationships, it may be possible to either provide a fine-grain measurement of how close an input came to exercising a branch outcome or design targeted mutation operators that are aware of control-flow relationships.

7.5.4 Dynamic Dispatch in Dynamic Analysis

Dynamic dispatch selects the implementation of an operation at runtime. This selection affects the flow of control—the next instruction executed depends upon which implementation is selected. Since dynamic dispatch can be used to select between alternative instructions for execution, it can also produce implicit flows like the ones discussed in Section 3.1. However, these dispatch-based flows are not considered by dynamic taint tracking systems despite behaving similarly to control flows. Furthermore, coverage-guided fuzzers, like Zeugma (presented in Chapter 4), typically use a coverage definition that does not consider dispatching to a different implementation of an operation from a call site as an interesting behavior. The impact of dynamic dispatch on the performance of these analyses is an interesting question for future work.

7.6 CLOSING THOUGHTS

In this manuscript, we present approaches for leveraging dynamic data relationships to improve software testing techniques. In Chapter 2, we demonstrate that a knowledge of data-flow relationships can be used to re-purpose functional tests to precisely detect injection vulnerabilities in web applications. We show in Chapter 3 that failure-relevant inputs can be identified by using alternative semantics to track control-flow relationships. In Chapter 4, we describe how a knowledge of relationships between portions of input sequences can be leveraged to create an effective crossover operator for parameter fuzzing. Finally, in Chapter 5, we present a general-purpose approach for automatically tracking information flows in modern JVMs. Based on our empirical evaluations of these approaches, we conclude that a *knowledge of dynamic data relationships can be leveraged to amplify existing software tests to reduce the amount of developer effort and expertise needed to detect and correct defects.*

BIBLIOGRAPHY

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Mark W. Aldrich, Alexi Turcotte, Matthew Blanco, and Frank Tip. 2023. Augur: Dynamic Taint Analysis for Asynchronous JavaScript. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22).* Association for Computing Machinery, New York, NY, USA, Article 153, 4 pages. https://doi.org/10.1145/3551349.3559522

[3] 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* (Baltimore, MD, USA) *(SEC'18).* USENIX Association, Berkeley, CA, USA, 377–392. http://dl.acm.org/citation.cfm?id=3277203.3277232

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

[5] Apache Software Foundation. 2019a. Apache Commons Codec (version 1.14). http://commons.apache.org/proper/commons-codec/.

[6] Apache Software Foundation. 2019b. Apache Commons Text (version 1.8). https://commons.apache.org/proper/commons-text/.

[7] Apache Software Foundation. 2019c. OGNL - Apache Commons OGNL - Developer Guide. https://commons.apache.org/proper/commons-ognl/developer-guide.html.

[8] Apache Software Foundation. 2019d. Security. https://struts.apache.org/security/.

[9] Apache Software Foundation. 2023a. Apache Ant (version 1.10.13). https://ant.apache.org/.

[10] Apache Software Foundation. 2023b. Apache Commons BCEL Issue #367. https://issues.apache.org/jira/browse/BCEL-367.

[11] Apache Software Foundation. 2023c. Apache Commons BCEL Issue #368. https://issues.apache.org/jira/browse/BCEL-368.

[12] Apache Software Foundation. 2023d. Apache Commons BCEL (version 6.7.0). https://commons.apache.org/proper/commons-bcel/.

[13] Apache Software Foundation. 2023e. Apache Log4j. https://logging.apache.org/log4j/2.x/.

[14] Apache Software Foundation. 2023f. Apache Maven (version 3.9.2). https://maven.apache.org/.

[15] Apache Software Foundation. 2023g. Apache Tomcat (version 10.1.9). https://tomcat.apache.org.

[16] Apache Software Foundation. 2024. Apache Tomcat. https://tomcat.apache.org.

[17] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. https://doi.org/10.1002/stvr.1486 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1486

[18] Ellen Arteca and Alexi Turcotte. 2022. npm-filter: automating the mining of dynamic information from npm packages. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 304–308. https://doi.org/10.1145/3524842.3528501

[19] 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 *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[20] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, USA. https://doi.org/10.14722/ndss.2019.23412

[21] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI '10)*. USENIX Association, USA, 237–250.

[22] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. FUDGE: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 975–985. https://doi.org/10.1145/3338906.3340456

[23] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Strict Control Dependence and Its Effect on Dynamic Information Flow Analyses. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/1831708.1831711

[24] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[25] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (oct 2017), 27 pages. https://doi.org/10.1145/3133876

[26] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 83–101. https://doi.org/10.1145/2660193.2660212

[27] Jonathan Bell and Gail Kaiser. 2015. Dynamic Taint Tracking for Java with Phosphor (Demo). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 409–413. https://doi.org/10.1145/2771783.2784768

[28] Jonathan Bell and Phosphor Contributors. 2024. Phosphor. https://central.sonatype.com/artifact/edu.gmu.swe.phosphor/Phosphor.

[29] 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.

[30] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[31] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure While Fuzzing. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1985–2002.

[32] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. 2021. Evaluating Synthetic Bugs. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (Virtual Event, Hong Kong) *(ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 716–730. https://doi.org/10.1145/3433210.3453096

[33] Pierre Carbonnelle. 2024. PYPL PopularitY of Programming Language Index. https://pypl.github.io/PYPL.html.

[34] D. Chandra and M. Franz. 2007. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE Computer Society, 463–475. https://doi.org/10.1109/ACSAC.2007.37

[35] Check Point Research Team. 2021. The Numbers Behind Log4j Vulnerability CVE-2021-44228. https://blog.checkpoint.com/security/the-numbers-behind-a-cyber-pandemic-detailed-dive/.

[36] Checkstyle Contributors. 2020a. Checkstyle Commit 70c7ae0. https://github.com/checkstyle/checkstyle/commit/70c7ae0e1866074530a49c983d015936a0c2c10f.

[37] Checkstyle Contributors. 2020b. Checkstyle Issue #8934. https://github.com/checkstyle/checkstyle/issues/8934.

[38] Checkstyle Contributors. 2020c. Checkstyle (version 8.37). https://github.com/checkstyle/checkstyle.

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

[40] 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 pages. https://doi.org/10.1109/ISCC.2006.158

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

[42] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[43] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA '07)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/1273463.1273490

[44] James Clause and Alessandro Orso. 2009. Penumbra: Automatically Identifying Failure-relevant Inputs Using Dynamic Tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. Association for Computing Machinery, New York, NY, USA, 249–260. https://doi.org/10.1145/1572272.1572301

[45] Closure Compiler Authors. 2014a. Google Closure Compiler Commit aac5d11. https://github.com/google/closure-compiler/commit/aac5d11480a0ed3f37919c23a5d3cc210e534bd5.

[46] Closure Compiler Authors. 2014b. Google Closure Compiler Issue #652. https://github.com/google/closure-compiler/issues/652.

[47] Closure Compiler Authors. 2014c. Google Closure Compiler (version v20140814). https://github.com/google/closure-compiler.

[48] Closure Compiler Authors. 2020. Google Closure Issue #3593. https://github.com/google/closure-compiler/issues/3593.

[49] Closure Compiler Authors. 2023a. Google Closure Issue #4096. https://github.com/google/closure-compiler/issues/4096.

[50] Closure Compiler Authors. 2023b. Google Closure (version v20230502). https://github.com/google/closure-compiler.

[51] Keith Cooper, Timothy Harvey, and Ken Kennedy. 2006. *A Simple, Fast Dominance Algorithm.* Rice University, CS Technical Report 06-33870. Rice University.

[52] Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. 2014. SpanDex: Secure Password Tracking for Android. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 481–494. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/cox

[53] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (October 1991), 451–490. https://doi.org/10.1145/115372.115320

[54] DaCapo Project Contributors. 2023a. dacapo-23.11-chopin RELEASE NOTES 2023-11. https://github.com/dacapobench/dacapobench/blob/fd292e92f8c40495a6ca05ff3b8a77c6c4265606/benchmarks/RELEASE_NOTES.md.

[55] DaCapo Project Contributors. 2023b. The DaCapo Benchmark Suite (version dacapo-23.11-chopin). https://github.com/dacapobench/dacapobench/releases/tag/v23.11-chopin.

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

[57] Kenneth Alan De Jong. 1975. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* Ph. D. Dissertation. University of Michigan, USA. AAI7609381.

[58] Kenneth A. De Jong and William M. Spears. 1991. An analysis of the interacting roles of population size and crossover in genetic algorithms. In *Parallel Problem Solving from Nature*, Hans-Paul Schwefel and Reinhard Männer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 38–47.

[59] Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (July 1977), 504–513. https://doi.org/10.1145/359636.359712

[60] Eclipse Foundation. 2019. Jetty - Servlet Engine and Http Server. https://www.eclipse.org/jetty/.

[61] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: a dynamic analysis framework for Python *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 760–771. https://doi.org/10.1145/3540250.3549126

[62] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010a. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/taintdroid-information-flow-tracking-system-realtime-privacy-monitoring

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

[64] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (December 2007), 35–45.

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

[66] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, USA.

[67] M. Franz, D. Chandra, and V. Haldar. 2005. Dynamic Taint Propagation for Java. In *Computer Security Applications Conference, Annual*. IEEE Computer Society, Los Alamitos, CA, USA, 303–311. https://doi.org/10.1109/CSAC.2005.21

[68] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary Generation of Whole Test Suites. In *Proceedings of the 2011 11th International Conference on Quality Software (QSIC '11)*. IEEE Computer Society, USA, 31–40. https://doi.org/10.1109/QSIC.2011.19

[69] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2, Article 8 (Dec. 2014), 42 pages. https://doi.org/10.1145/2685612

[70] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*. 474–484. https://doi.org/10.1109/ICSE.2009.5070546

[71] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

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

[73] Google LLC. 2020. Guava (version 28.2-jre). https://github.com/google/guava.

[74] H2 Contributors. 2020a. H2 Commit 6c564e6. https://github.com/h2database/h2database/commit/6c564e63eb6a3c819eaab19f4aece3298db2ab5f.

[75] H2 Contributors. 2020b. H2 Issue #2550. https://github.com/h2database/h2database/issues/2550.

[76] H2 Contributors. 2020c. H2 (version 1.4.200). https://github.com/h2database/h2database/.

[77] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. 2006. Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Portland, Oregon, USA) *(SIGSOFT '06/FSE-14)*. Association for Computing Machinery, New York, NY, USA, 175–185. https://doi.org/10.1145/1181775.1181797

[78] 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* (Gothenburg, Sweden) *(ICSE-SEET '18)*. Association for Computing Machinery, New York, NY, USA, 1–4. https://doi.org/10.1145/3183377.3183393

[79] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically Reducing Tree-Structured Test Inputs. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 861–871.

[80] John H. Holland. 2000. Building Blocks, Cohort Genetic Algorithms, and Hyperplane-Defined Functions. *Evol. Comput.* 8, 4 (dec 2000), 373–391. https://doi.org/10.1162/106365600568220

[81] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 38.

[82] Paul Holser. 2023. junit-quickcheck. https://github.com/pholser/junit-quickcheck.

[83] 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

[84] Katherine Hough and Jonathan Bell. 2021a. Conflux: A Practical Approach for Dynamic Taint Tracking with Control-Flow Relationships (Artifact). (9 2021). https://doi.org/10.6084/m9.figshare.16611424.v1

[85] Katherine Hough and Jonathan Bell. 2021b. A Practical Approach for Dynamic Taint Tracking with Control-Flow Relationships. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 26 (Dec. 2021), 43 pages. https://doi.org/10.1145/3485464

[86] Katherine Hough and Jonathan Bell. 2024a. Crossover in Parametric Fuzzing. In *Proceedings of the ACM/IEEE 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639160

[87] Katherine Hough and Jonathan Bell. 2024b. Crossover in Parametric Fuzzing (Artifact). (1 2024). https://doi.org/10.6084/m9.figshare.23688879.v1

[88] Katherine Hough, Gebrehiwet Welearegai, Christian Hammer, and Jonathan Bell. 2020a. Revealing Injection Vulnerabilities by Leveraging Existing Tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 284–296. https://doi.org/10.1145/3377811.3380326

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

[90] Chen Huo and James Clause. 2014. Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 621–631. https://doi.org/10.1145/2635868.2635917

[91] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 435–445. https://doi.org/10.1145/2568225.2568271

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

[93] Kangkook Jee. 2015. *On Efficiency and Accuracy of Data Flow Tracking Systems.* Ph. D. Dissertation. Columbia University. https://doi.org/10.7916/D8MG7P9D

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

[95] Jianyu Jiang, Shixiong Zhao, Danish Alsayed, Yuexuan Wang, Heming Cui, Feng Liang, and Zhaoquan Gu. 2017. Kakute: A Precise, Unified Information Flow Analysis System for Big-Data Security. In *Proceedings of the 33rd Annual Computer Security Applications Conference* (Orlando, FL, USA) *(ACSAC 2017)*. Association for Computing Machinery, New York, NY, USA, 79–90. https://doi.org/10.1145/3134600.3134607

[96] Jonathan Hedley. 2024. jsoup: Java HTML Parser. https://jsoup.org/.

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

[98] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[99] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Xiaodong Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *NDSS*.

[100] Rezwana Karim, Frank Tip, Alena Sochůrková, and Koushik Sen. 2020. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering* 46, 12 (2020), 1364–1379. https://doi.org/10.1109/TSE.2018.2878020

[101] 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* (London, England, UK) *(VEE '12)*. Association for Computing Machinery, New York, NY, USA, 121–132. https://doi.org/10.1145/2151024.2151042

[102] 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.

[103] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[104] Herb Krasner. 2022. *The Cost of Poor Software Quality in the US: A 2022 Report.* Technical Report. Consortium for Information and Software Quality (CISQ). https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf

[105] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. CONFETTI: Amplifying Concolic Guidance for Fuzzers. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 438–450. https://doi.org/10.1145/3510003.3510628

[106] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage Guided, Property Based Testing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 181 (oct 2019), 29 pages. https://doi.org/10.1145/3360607

[107] Legion of the Bouncy Castle Inc. 2011. Bouncy Castle Provider (version 1.46). http://bouncycastle.org/.

[108] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[109] Ao Li, Madonna Huang, Caroline Lemieux, and Rohan Padhye. 2024. The Havoc Paradox in Generator-Based Fuzzing (Registered Report). In *Proceedings of the 3rd ACM International Fuzzing Workshop* (Vienna, Austria) *(FUZZING 2024)*. Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/3678722.3685529

[110] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[111] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. 2013. The Java Virtual Machine Specification: Java SE 21 Edition. https://docs.oracle.com/javase/specs/jvms/se21/jvms21.pdf.

[112] 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.*

[113] 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* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 385–398. https://doi.org/10.1145/2429069.2429115

[114] LLVM Project. 2024. libFuzzer. https://llvm.org/docs/LibFuzzer.html.

[115] 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 — Building Trustworthy Applications* (St. Louis, Missouri) *(SESS '05)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/1082983.1083216

[116] Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning Input Tokens for Effective Fuzzing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 27–37. https://doi.org/10.1145/3395363.3397348

[117] Stephen McCamant and Michael D. Ernst. 2008. Quantitative Information Flow As Network Flow Capacity. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 193–205. https://doi.org/10.1145/1375581.1375606

[118] MDN Contributors. 2018a. Mozilla Rhino Issue #397. https://github.com/mozilla/rhino/issues/397.

[119] MDN Contributors. 2018b. Mozilla Rhino Issue #405. https://github.com/mozilla/rhino/issues/405.

[120] MDN Contributors. 2018c. Mozilla Rhino Issue #406. https://github.com/mozilla/rhino/issues/406.

[121] MDN Contributors. 2018d. Mozilla Rhino Issue #409. https://github.com/mozilla/rhino/issues/409.

[122] MDN Contributors. 2019a. Mozilla Rhino Commit 0c0bb39. https://github.com/mozilla/rhino/commit/0c0bb391647600ec706b1ec66f71831893a6f564.

[123] MDN Contributors. 2019b. Mozilla Rhino Issue #539. https://github.com/mozilla/rhino/issues/539.

[124] MDN Contributors. 2019c. Mozilla Rhino (version 1.7.11). https://github.com/mozilla/rhino.

[125] MDN Contributors. 2023a. Mozilla Rhino Issue #1337. https://github.com/mozilla/rhino/issues/1337.

[126] MDN Contributors. 2023b. Mozilla Rhino (version 1.7.14). https://github.com/mozilla/rhino.

[127] Michaël Mera. 2019. Mining Constraints for Grammar Fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 415–418. https://doi.org/10.1145/3293882.3338983

[128] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1393–1403. https://doi.org/10.1145/3468264.3473932

[129] Michał Zalewski. 2024. American Fuzzing Lop (AFL). https://lcamtuf.coredump.cx/afl/.

[130] MirrorTaint Contributors. 2023. MirrorTaint (commit 54fcbfc). https://github.com/MirrorTaint/MirrorTaint.

[131] Melanie Mitchell, John H. Holland, and S. Forrest. 1992. The royal road for genetic algorithms: Fitness landscapes and GA performance. In *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*. MIT Press, Cambridge, MA, USA, 245–254.

[132] MITRE Corporation. 2019. CWE-601: URL Redirection to Untrusted Site ('Open Redirect'). https://cwe.mitre.org/data/definitions/601.html.

[133] 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

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

[135] Mountainminds GmbH & Co. KG and Contributors. 2021. JaCoCo Java Code Coverage Library. https://github.com/jacoco/jacoco.

[136] Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. 2008. A Virtual Machine Based Information Flow Control System for Policy Enforcement. *Electron. Notes Theor. Comput. Sci.* 197, 1 (feb 2008), 3–16. https://doi.org/10.1016/j.entcs.2007.10.010

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

[138] National Institute of Standards and Technology. 2021. CVE-2021-44228. https://nvd.nist.gov/vuln/detail/CVE-2021-44228.

[139] 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.

[140] New Relic, Inc. 2023. *2023 State of the Java Ecosystem.* Technical Report. New Relic, Inc. https://newrelic.com/sites/default/files/2023-04/new-relic-2023-state-of-the-java-ecosystem-2023-04-20.pdf

[141] Hoang Lam Nguyen and Lars Grunske. 2022. BeDivFuzz: Integrating Behavioral Diversity into Generator-Based Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 249–261. https://doi.org/10.1145/3510003.3510182

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

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

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

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

[146] Open Web Application Security Project. 2019d. XSS Filter Evasion Cheat Sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.

[147] OpenRefine Contributors. 2020a. OpenRefine Commit 825e687. https://github.com/OpenRefine/OpenRefine/commit/825e687b0b676fd1be1fa0a9d00be22de0e57060.

[148] OpenRefine Contributors. 2020b. OpenRefine Issue #2584. https://github.com/OpenRefine/OpenRefine/issues/2584.

[149] OpenRefine contributors. 2020. OpenRefine (version 3.4-SNAPSHOT). https://github.com/OpenRefine/OpenRefine.

[150] Oracle Corporation. 2023a. Java Platform, Standard Edition & Java Development Kit: Version 11 API Specification. https://docs.oracle.com/en/java/javase/11/docs/api/index.html.

[151] Oracle Corporation. 2023b. Java Platform, Standard Edition & Java Development Kit: Version 21 API Specification. https://docs.oracle.com/en/java/javase/21/docs/api/index.html.

[152] Oracle Corporation. 2023c. JDK Releases. https://www.java.com/releases/.

[153] Oracle Corporation. 2023d. Oracle Java Bug Database JDK-8309911. https://bugs.java.com/bugdatabase/view_bug?bug_id=JDK-8309911.

[154] Oracle Corporation. 2023e. Oracle Java Bug Database JDK-8309914. https://bugs.java.com/bugdatabase/view_bug?bug_id=JDK-8309914.

[155] Oracle Corporation. 2023f. Oracle Java Bug Database JDK-8309915. https://bugs.java.com/bugdatabase/view_bug?bug_id=JDK-8309915.

[156] Oracle Corporation. 2024a. Java Native Interface: Introduction. https://docs.oracle.com/en/java/javase/21/docs/specs/jni/intro.html.

[157] Oracle Corporation. 2024b. JEP 193: Variable Handles. https://openjdk.org/jeps/193.

[158] Oracle Corporation. 2024c. JEP 280: Indify String Concatenation. https://openjdk.org/jeps/280.

[159] Oracle Corporation. 2024d. JEP 395: Records. https://openjdk.org/jeps/395.

[160] Oracle Corporation. 2024e. jlink Tool Reference. https://docs.oracle.com/en/java/javase/11/tools/jlink.html.

[161] Oracle Corporation. 2024f. JSR 292: Supporting Dynamically Typed Languages on the Java Platform. https://jcp.org/en/jsr/detail?id=292.

[162] Oracle Corporation. 2024g. JSR 335: Lambda Expressions for the Java Programming Language. https://jcp.org/en/jsr/detail?id=335.

[163] Oracle Corporation. 2024h. JSR 376: Java Platform Module System. https://www.jcp.org/en/jsr/detail?id=376.

[164] Oracle Corporation. 2024i. OpenJDK Java Class Library. https://openjdk.java.net/.

[165] OSS-Fuzz Contributors. 2024. OSS-Fuzz. https://github.com/google/oss-fuzz.

[166] Yicheng Ouyang, Kailai Shao, Kunqiu Chen, Ruobing Shen, Chao Chen, Mingze Xu, Yuqun Zhang, and Lingming Zhang. 2023. MirrorTaint: Practical Non-intrusive Dynamic Taint Tracking for JVM-based Microservice Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2514–2526. https://doi.org/10.1109/ICSE48619.2023.00210

[167] OW2 Consortium. 2024. ASM. https://asm.ow2.io/.

[168] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. 2008. Finding errors in .net with feedback-directed random testing. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. Association for Computing Machinery, New York, NY, USA, 87–96. https://doi.org/10.1145/1390630.1390643

[169] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, USA, 75–84. https://doi.org/10.1109/ICSE.2007.37

[170] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019a. JQF: Coverage-Guided Property-Based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002

[171] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019b. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[172] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 331–341. https://doi.org/10.1145/3236024.3236029

[173] Jeff Perkins, Jordan Eikenberry, Allessandro Coglio, and Martin Rinard. 2020. Comprehensive Java Metadata Tracking for Attack Detection and Repair. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 39–51. https://doi.org/10.1109/DSN48063.2020.00024

[174] V. Pham, M. Bohme, A. E. Santosa, A. Caciulescu, and A. Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* 47, 09 (sep 2021), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681

[175] Pivotal Software. 2020. Spring Framework (version 5.2.5). https://spring.io/projects/spring-framework.

[176] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, USA, 135–148. https://doi.org/10.1109/MICRO.2006.29

[177] Günther R. Raidl and Jens Gottlieb. 2005. Empirical Analysis of Locality, Heritability and Heuristic Bias in Evolutionary Algorithms: A Case Study for the Multidimensional Knapsack Problem. *Evolutionary Computation* 13, 4 (12 2005), 441–475. https://doi.org/10.1162/106365605774666886

[178] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, USA. https://doi.org/10.14722/ndss.2017.23404

[179] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1410–1421. https://doi.org/10.1145/3377811.3380399

[180] Rohan Padhye and JQF Contributors. 2023. jqf-examples (version 2.0). https://central.sonatype.com/artifact/edu.berkeley.cs.jqf/jqf-examples/2.0.

[181] John Rose. 2013. Method handle invocation. https://wiki.openjdk.org/display/HotSpot/Method+handle+invocation.

[182] Franz Rothlauf. 2006. *Representations for Genetic and Evolutionary Algorithms.* Springer-Verlag, Berlin, Heidelberg.

[183] A. Sabelfeld and A. C. Myers. 2006. Language-based Information-flow Security. *IEEE J.Sel. A. Commun.* 21, 1 (September 2006), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[184] 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* (Dallas, Texas, USA) *(PLAS '17)*. Association for Computing Machinery, New York, NY, USA, 15–24. https://doi.org/10.1145/3139337.3139341

[185] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Trans. Softw. Eng.* 50, 1 (Nov. 2023), 85–105. https://doi.org/10.1109/TSE.2023.3334955

[186] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, USA, 317–331. https://doi.org/10.1109/SP.2010.26

[187] 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.

[188] Haichen Shen, Aruna Balasubramanian, Anthony LaMarca, and David Wetherall. 2015. Enhancing Mobile Apps to Use Sensor Hubs without Programmer Effort. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (Osaka, Japan) *(UbiComp '15)*. Association for Computing Machinery, New York, NY, USA, 227–238. https://doi.org/10.1145/2750858.2804260

[189] 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* (Montreal, Quebec, Canada) *(AST '19)*. IEEE Press, Piscataway, NJ, USA, 55–61. https://doi.org/10.1109/AST.2019.00014

[190] John Singleton. 2018. *Advancing Practical Specification Techniques for Modern Software Systems.* Ph. D. Dissertation. University of Central Florida. http://purl.fcla.edu/fcla/etd/CFE0007099

[191] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. 2013. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 1181–1192. https://doi.org/10.1145/2508859.2516696

[192] 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

[193] 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* (Portland, Oregon, USA). Association for Computing Machinery, 16 pages. https://doi.org/10.1145/2048066.2048145

[194] Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. 2007. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/1250734.1250748

[195] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective Grammar-Aware Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 244–256. https://doi.org/10.1145/3460319.3464814

[196] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. 2019. An Empirical Study of Information Flows in Real-World JavaScript. In *Proceedings of the 14th Workshop on Programming Languages and Analysis for Security* (London, United Kingdom) *(PLAS '19)*. Association for Computing Machinery, New York, NY, USA, 15 pages. https://doi.org/10.1145/3338504.3357339

[197] 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* (Charleston, South Carolina, USA) *(POPL '06)*. Association for Computing Machinery, New York, NY, USA, 372–382. https://doi.org/10.1145/1111037.1111070

[198] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS XI* (Boston, MA, USA). Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/1024393.1024404

[199] Terence Parr. 2019. ANTLR. https://www.antlr.org/.

[200] 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* (Hyderabad, India) *(SBST 2014)*. Association for Computing Machinery, New York, NY, USA, 5–14. https://doi.org/10.1145/2593833.2593835

[201] TIOBE Software BV. 2024. TIOBE Index for January 2024. https://www.tiobe.com/tiobe-index/.

[202] Frank Tip. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3 (1995), 121–189.

[203] John Toman and Dan Grossman. 2016. Staccato: A Bug Finder for Dynamic Configuration Updates. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 24:1–24:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.24

[204] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. 2023. Towards effective assessment of steady state performance in Java software: are we there yet? *Empirical Softw. Engg.* 28, 1 (jan 2023), 57 pages. https://doi.org/10.1007/s10664-022-10247-x

[205] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *PLDI '09* (Dublin, Ireland). Association for Computing Machinery, New York, NY, USA, 87–97. https://doi.org/10.1145/1542476.1542486

[206] Chia-che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. 2020. Civet: An Efficient Java Partitioning Framework for Hardware Enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 505–522. https://www.usenix.org/conference/usenixsecurity20/presentation/tsai

[207] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. http://www.jstor.org/stable/1165329

[208] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1072–1084. https://doi.org/10.1109/ICSE43902.2021.00100

[209] Dong Wang, Yu Gao, Wensheng Dou, and Jun Wei. 2022. DisTA: Generic Dynamic Taint Tracking for Java-Based Distributed Systems. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 547–558. https://doi.org/10.1109/DSN53405.2022.00060

[210] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[211] T. Wang, T. Wei, G. Gu, and W. Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *2010 IEEE Symposium on Security and Privacy*. 497–512.

[212] Richard A. Watson and Thomas Jansen. 2007. A Building-Block Royal Road Where Crossover is Provably Essential. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation* (London, England) *(GECCO '07)*. Association for Computing Machinery, New York, NY, USA, 1452–1459. https://doi.org/10.1145/1276958.1277224

[213] M. Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

[214] Laurie Williams, Gunnar Kudrjavets, and Nachiappan Nagappan. 2009. On the Effectiveness of Unit Test Automation at Microsoft. In *2009 20th International Symposium on Software Reliability Engineering*. 81–89. https://doi.org/10.1109/ISSRE.2009.32

[215] World Wide Web Consortium. 2017. HTML 5.2. https://www.w3.org/TR/html52/.

[216] World Wide Web Consortium. 2019. Parsing HTML Documents. https://html.spec.whatwg.org/multipage/parsing.html.

[217] Hyrum Wright, Titus Delafayette Winters, and Tom Manshreck. 2020. *Software Engineering at Google.*

[218] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019a. SLF: Fuzzing without Valid Seed Inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 712–723. https://doi.org/10.1109/ICSE.2019.00080

[219] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019b. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *2019 IEEE Symposium on Security and Privacy (SP)*. 769–786. https://doi.org/10.1109/SP.2019.00057

[220] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. 28, 2 (February 2002), 183–200. https://doi.org/10.1109/32.988498

[221] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2024. How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1223–1235. https://doi.org/10.1145/3650212.3680355

[222] Xiangyu Zhang, Neelam Gupta, and Rajeev Gupta. 2006. Pruning dynamic slices with confidence. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 169–180. https://doi.org/10.1145/1133981.1134002