Towards Automatic Detection and Exploitation of Java Web Application Vulnerabilities via Concolic Execution guided by Cross-thread Object Manipulation

Xinyou Huang[1*], Lei Zhang[2*], Yongheng Liu[1], Peng Deng[1], Yinzhi Cao[3], Yuan Zhang[2], and Min Yang[2]
1: Fudan University, {xinyouhuang22, yhliu24, pdeng21}@m.fudan.edu.cn
2: Fudan University, {zxl, yuanxzhang, m_yang}@fudan.edu.cn
3: Johns Hopkins University, yinzhi.cao@jhu.cn
**: co-first authors*

Abstract

Java Web applications are of great importance for information systems deployed across critical sections of our society as demonstrated in the severe impacts caused by notorious log4j vulnerability. One major challenge in detecting Java Web Application vulnerabilities is cross-thread dataflows, which are caused by shared Java objects and triggered by multiple web requests in the same session. To the best of our knowledge, none of the prior works can handle such cross-thread dataflows in Java Web applications.

In this paper, we design and implement the first framework, called JAEX, to automatically detect and exploit Java Web Application vulnerabilities via concolic execution guided by so-called Cross-thread Object Manipulation. Our key insight is that cross-thread dataflows can be triggered by manipulation of shared Java objects using different requests, thus guiding concolic execution to reach the sink and generate exploits. We also evaluate JAEX on popular Java applications, which discovers 35 zero-day vulnerabilities. We responsibly disclosed all the vulnerabilities to their vendors and received acknowledgments for all of them.

1 Introduction

Java Web applications (apps) are extensively deployed in information systems across critical sections of society, such as enterprises and government departments. Such wide deployment also draws the attention from adversaries: For example, the notorious log4j vulnerability [1] brought a nightmare to the entire Internet, which affected millions of web apps. Therefore, automatic detection and exploitation of Java Web apps are of critical importance to minimize security risks.

One major barrier of automatic vulnerability detection and exploitation is due to the nature of Java Web apps, which are multi-threads by default. That is, each request of the same session may generate a new thread and the Java objects are shared across different threads within the session. Such a *cross-thread dataflow* inevitably renders vulnerability detection and exploitation challenging. First, a specific sequence of requests is needed to reach the final sink via different cross-thread dataflows. Second, payloads with complex structures and inter-dependencies are needed to trigger each cross-thread dataflow.

To the best of our knowledge, none of the prior works can handle cross-thread dataflows for automatic vulnerability detection and exploitation of Java Web apps. Let us describe prior works from two aspects. On one hand, previous works have studied non-Java Web apps [2–11], sometimes involving dataflows across requests or threads, in their program analysis. For example, NAVEX [3] proposed a Navigation Graph-guided symbolic execution approach for PHP Web apps with connecting the dataflows between different requests. Many works for Android apps [8–11], e.g., Horndroid [11], also modeled cross-thread dataflows, for information flow analysis, e.g, privacy leaks. However, it is challenging to apply them in detecting and exploiting complex vulnerabilities of Java Web apps. Specifically, NAVEX, focusing on PHP applications, cannot model shared cross-thread Java objects in their navigation graph. Then, Horndroid, which is mainly designed for information flow analysis, cannot provide or generate inputs to exploit Java vulnerabilities.

On the other hand, previous Java vulnerability detection focuses more on specific types of vulnerabilities, e.g., Deserialization Vulnerabilities [12–15] and Denial-of-Service Vulnerability [16–18], which do not have cross-thread dataflows heavily involved. Witcher [19], a multi-language fuzzer of web apps, does support Java Web apps, but their guidance is based on code coverage. That is, Witcher is unaware of cross-thread dataflows, let alone able to specifically generate request sequences and related payloads. Joern [20], a well-known static analysis framework supporting Java white-box code analysis, also lacks support for cross-thread dataflow analysis. Furthermore, Joern often suffers from a large number of false positives in its detection result, because of over-approximations in the static analysis [21] and lack of constraint solving.

In this paper, we design and implement *the first* automatic framework, called JAEX, to detect and exploit Java Web Ap-

plication vulnerabilities via concolic execution guided by so-called Cross-thread Object Manipulation. Our key insight is that cross-thread dataflows are connected and triggered via shared Java Objects, and therefore the manipulation of such objects via different requests to the web application can guide concolic execution to reach the sink.

Specifically, JAEX constructs the request sequence and then the payloads of each request via concolic execution. The first step is the request sequence construction. JAEX first analyzes the whole program to extract all potential execution paths to sinks and the objects that can not be influenced by the input entry of these paths with a pre-generated cross-thread dataflow graph. After that, JAEX marks these objects as guidance symbols and utilizes them to guide concolic execution with an on-demand, iterative path exploration strategy, thereby figuring out the correct web entry (i.e., request) sequence to trigger the sinks. The second step is the payload construction. JAEX records and solves the path constraints and the data-flow dependencies among execution paths of different threads during concolic execution, represents such dependencies using a novel Cross-thread Object Manipulation Graph, and eventually constructs the concrete payloads.

We evaluate JAEX's capabilities on 25 of the latest versions of popular Java apps and discover over 35 zero-day vulnerabilities, of which exploits were generated for all of them. All discovered vulnerabilities have been reported to the respective vendors, and acknowledgments have been received.

We also evaluate JAEX on a benchmark consisting of 92 historical vulnerabilities from 16 popular Java open-source apps. The results demonstrate that JAEX detected 90 vulnerabilities in this benchmark and generated exploits for 87 of them. In contrast, our baselines, Witcher [19] and Joern [20] detected only two and 44 of these vulnerabilities, respectively, and were unable to generate any exploits.

To sum up, this paper makes the following contributions:

- We introduce for the first time an Automated Exploit Generation (AEG) approach for Java Web apps. We have designed and implemented our prototype tool, JAEX, to address the unique challenges brought by Java Web apps.
- We are the first to highlight cross-thread vulnerabilities in Java and propose a sink-guided, on-demand, and step-forward path exploration strategy for concolic execution approach to generate exploits.
- Our approach has successfully identified 35 zero-day vulnerabilities and generated exploits for each of them, demonstrating its efficacy in detecting and exploiting complex vulnerabilities in Java Web apps.

2 Overview

In this section, we describe a real-world vulnerability discovered by JAEX as a motivating example in §2.1, then illustrate the challenges and the overview of our solutions in §2.2.

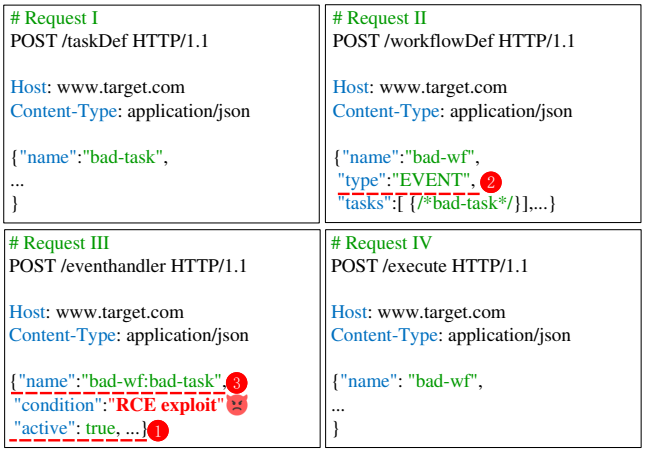

Figure 1: The exploit of our motivating example. Fields with a red dashed underline indicate the control flow conditions that must be met for vulnerability exploitation.

2.1 A Motivating Example

This subsection illustrates a motivating example using a zero-day, remote code execution (RCE) vulnerability found by JAEX in a widely-used open-source component. Figure 1 shows the exploit code with four HTTP requests and then Figure 2 illustrates the vulnerable code, which contains four web entry methods (Lines 4, 10, 16, and 30) accessible via POST requests and a sink function (i.e., where the vulnerability is located) at Line 55. We have reported this vulnerability to the developers and assisted them in fixing it in the latest version.

We now describe how this vulnerability and its exploit work. The exploitation starts with the first two HTTP requests in Figure 1, which add `TaskDef` and `WorkflowDef` into the database. After that, the third request needs to be sent to trigger the `addEventHandler` method (Line 16 in Figure 2). This action will add the `eh` (a `EventHandler` object) into the database, and then the background thread method `refreshEventQueues` method (Line 21) loads the newly added `EventHandler` from the database and sets up an `ObQueue` as a listener (i.e., a new background thread) with the name of new `EventHandler` to handle message generated at program runtime. Finally, the attacker needs to send the fourth request to `startWorkflow` method at Line 30 to ultimately finish the exploitation. In this method, the `WorkflowDef`, i.e., `wfd` added by Request II is retrieved from the database with the status set to 'RUNNING', and the `handleWorkflow` method is invoked with `wfd` as a parameter. The `handleWorkflow` method processes the task according to the `task.type`, gets an `ObQueue` with the name of the `WorkflowDef` and `taskDef` that is created by the Request III, and sends a message to the queue(Lines 40-42). Finally, the message will be received by `Daemon Thread II` (Lines 46-49), handled by `EventProcessor` (Line 50), and triggered the vulnerability (Line 55).

Note that there are many cross-thread dataflows triggered

by different requests here. First, the field `tasks` of the payload of Request II should contain the `taskDef` sent by Request I; Second, the field `active` at Request III should be 'true' (❶) to ensure the `Daemon Thread II` is set up; Third, the field `type` at Request II should be 'EVENT' (❷); Fourth, the field `eventName` (❸) should be a specifically formatted string composed of `taskName` and `workflowName`. This ensures that the application can correctly retrieve the `ObQueue` on Line 42 in Figure 2 after receiving the fourth request.

2.2 Challenges and Solution Overview

In this subsection, we describe how cross-thread dataflows in Java Web apps bring challenges for vulnerability detection and exploitation via our motivating example. Then, we also describe how JAEX solves these challenges.

Challenge I: Identifying attack request sequence with a specific order. Because of cross-thread dataflow dependencies, certain requests have to be prepared in a specific order so that other requests can proceed. For example, Figure 1 shows that these four HTTP requests have to come in the stated order so that all the data dependencies in Code Snippet (c) of Figure 2 can be fulfilled. Existing works [3, 19, 20] are blind to or insufficiently address cross-thread dataflow, which can either be unable to generate such a sequence or generate a wrong sequence.

To address this Challenge I, JAEX proposes an on-demand, iterative vulnerability path searching algorithm (details are presented later in Algorithm 1), which considers cross-thread dataflows. The key observation here is that to reach the sink, a series of shared objects need to be manipulated via different requests. As a result, by analyzing when and by which request these objects need to be manipulated, JAEX can infer the correct request sequence. For example, along the potential vulnerability path triggered by Request IV method `startWorkflow`, the `eh` needed to exploit the vulnerability (Line 53) is generated from Request III method `addEventHandler` (Line 17), and therefore JAEX can find that the `addEventHandler` should be invoked before the `startWorkflow`.

Challenge II: Generating attack payloads for each request to exploit the vulnerability. Vulnerability exploitation requires valid, highly structured payloads for each request identified in Challenge I, with these payloads being interdependent across requests. Figure 3 shows the dependencies between the payloads of different requests in the motivating example in the form of a Cross-thread Object Manipulation Graph (COMG, see § 3.5). For instance, the List field `wfDef.tasks` in Request II needs to include the `tDef` as an item from Request I. This is challenging for previous fuzzing-based approaches, e.g., Witcher [19], because they are unaware of cross-thread dataflows, and the payloads are generated randomly, e.g., adopting AFL's random strategy.

Note that these dependent fields between these payloads

```
1  class TaskDefController {
2   /** Entry for Request I **/
3   @Post("/taskDef")
4   void addTaskDef(TaskDef tDef) {
5    DB.addTaskDef(tDef);
6  }}
7  class WorkflowDefController {
8   /** Entry for Request II **/
9   @Post("/workflowDef")
10  void addWorkflowDef(WorkflowDef wfDef) {
11   DB.addWorkflowDef(wfDef);
12 }}
```

(a) Codes behind the Request I and II

```
13 class EventController {
14  /** Entry for Request III **/
15  @Post("/eventhandler")
16  void addEventHandler(EventHandler eh) {
17   DB.addEventHandler(eh);
18 }}
19 class QueueManager {
20  static QueueMap qMap;
21  void refreshEventQueues(){//Daemon Thread I
22   EventHandler e = DB.getNewEventHandler());   Req III to Daemon Thread I (eh)
23  ❶if(e.active)
24    qMap.put(e.name, new ObQueue(e.name));
25  static ObQueue getQueue(String name) {...}
26 }
```

(b) Codes behind the Request III

```
27 class WorkflowController {
28  /** Entry for Request IV **/
29  @Post("/execute")
30  void startWorkflow(StartWorkflowReq req) {
31   WorkflowDef wfd = DB.getWorkflowDef(req.name);   Req II to IV (wfDef)
32   wfd.status = RUNNING;
33   handleWorkflow(wfd)
34  }
35  void handleWorkflow(WorkflowDef wfd){
36   if (wfd.status == RUNNING) {
37    for (TaskDef task : wfd.tasks) {
38  ❷if (task.type.equals("EVENT")) {
39     Message msg = doTask(task);
40  ❸ String qn = wfd.name + ":" + tDef.name;
41     ObQueue q = QueueManager.qMap.get(qn);   Daemon Thread I to Req IV (ObQueue)
42     if (queue != null) { q.mq.push(msg); }
43 }}}}
44 class ObQueue {
45  EventProcessor ep; MsgQueue mq;
46  void enable() {// Daemon Thread II
47   List messages = mq.pop();   Req IV to Daemon Thread II (msg)
48   messages.forEach(ep::handle(this, msg));
49 }}
50 class EventProcessor {
51  void handle(ObQueue queue, Message msg) {
52   String name = queue.getEventName();
53   EventHandler eh = DB.getEventHandler(name);   Req III to Daemon Thread II (eh)
54   ScriptEvaluator.eval(eh.condition, msg);
55 }}        RCE exploit here!
```

(c) Codes behind the Request IV

Figure 2: The vulnerable code of our motivating example. The code is simplified to facilitate presentation, e.g., uniformly representing all database operations using methods from `DB`. Dashed arrows in different colors represent cross-thread data flows of different objects.

are connected by the cross-thread dataflows triggered by different requests. As shown in Figure 2, the `eh` (Line 17) from Request III flows to the `qMap.key` with a newly generated `ObQueue` (Line 24). Then, the new `ObQueue` can be obtained with `qn` at Line 41, and the `qn` is constructed by the `wfd.name` and `tDef.name` that flow from Request I and II. Therefore, with these dataflow relations, JAEX can infer that the de-

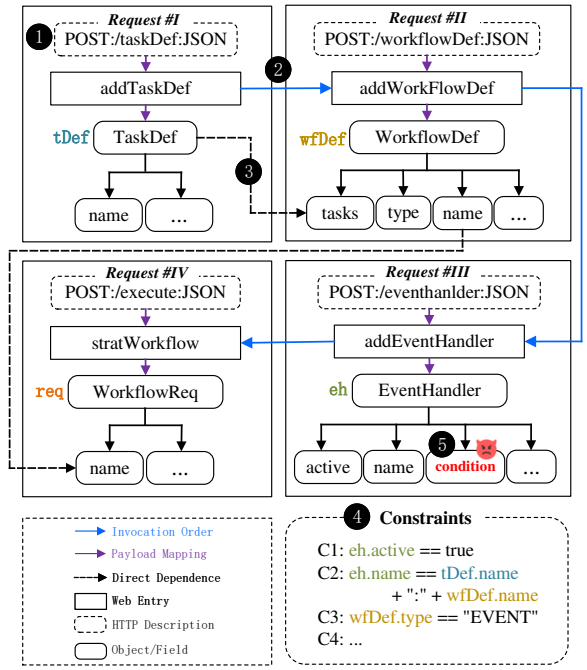

Figure 3: The Simplified COMG of the Motivating Example

pendency between `eh.name`, `tDef.name`, and `wfDef.name` in each payloads.

JAEX uses concolic execution to identify cross-thread dataflow dependencies and help construct these payloads via two steps. First, by tracking user input during concolic execution, JAEX precisely identifies which fields in the input have cross-thread dataflow dependencies. Second, by incorporating a constraint solver, JAEX determines the constraints that are met to pass through specific branches so that the injected payload can reach the final sink.

2.3 Threat Model

In this work, we focus on all Java apps that have web access interfaces (i.e., accepting HTTP requests) and can be deployed on a server, collectively referred to as Java Web apps. That is, our victim is a server-side Java Web application, and our adversary is a malicious client that directly sends one or more HTTP requests with malicious payloads through these web interfaces to exploit vulnerabilities. Our *in-scope* vulnerability primarily includes injection vulnerabilities in Java Web apps, such as SQL injection, command injection, and expression injection [22]. Other vulnerability types, e.g., Denial of Service (DoS) and deserialization, are currently out of the scope of JAEX, but may be included in the future with sufficient modeling because cross-thread dataflows can also be involved for these vulnerabilities.

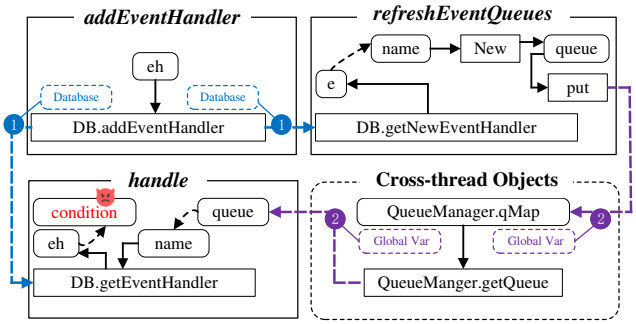

Figure 4: The Simplified Cross-thread DFG of the partial of codes in Motivating Example

3 Approach

In this section, we first present the system architecture of JAEX at §3.1 and how it works on the motivating example. Then, we describe the details of each module.

3.1 System Architecture

Architecture Overview. Figure 5 presents the overall architecture of JAEX, which is a hybrid framework with four phases in two main stages.

Stage 1: Static analysis. In this stage, JAEX detects the potential vulnerability and extracts the information that is needed for conducting concolic execution.

In Phase 1, JAEX models and identifies the source and sink APIs that indicate the vulnerabilities. One obstacle here is that, Java Web apps commonly contain a lot of abstract-type objects (e.g., using dependency injection to declare Java objects), JAEX hardly can obtain a complete control-flow graph (CFG) of the app, resulting a high false negative of source/sink identifications. Thus, we propose a pointer-based ICFG by utilizing pointer-to analysis to enhance the construction of the control-flow graph.

In Phase 2, JAEX identifies shared objects and builds a cross-thread data-flow graph (DFG) to connect the sources and sinks across different threads. Specifically, JAEX uses a sink-guided, on-demand iterative algorithm (will be illustrated with Algorithm 1 at §3.3) to search the DFG and detect if the parameters of sinks could be tainted by user input. Once a vulnerable data flow is identified, JAEX further extracts the execution path from CFG. Then, it could infer the web entry invocation sequence by identifying the API manipulation sequence of shared objects (e.g., a write API should be executed before a read API). Last, for guiding concolic execution, JAEX will further extract the needed code context and initialize the necessary information with the identification of entries, sinks, and shared objects among different threads.

Stage 2: Concolic Execution. In this stage, JAEX verifies the possible vulnerabilities detected by static analysis and

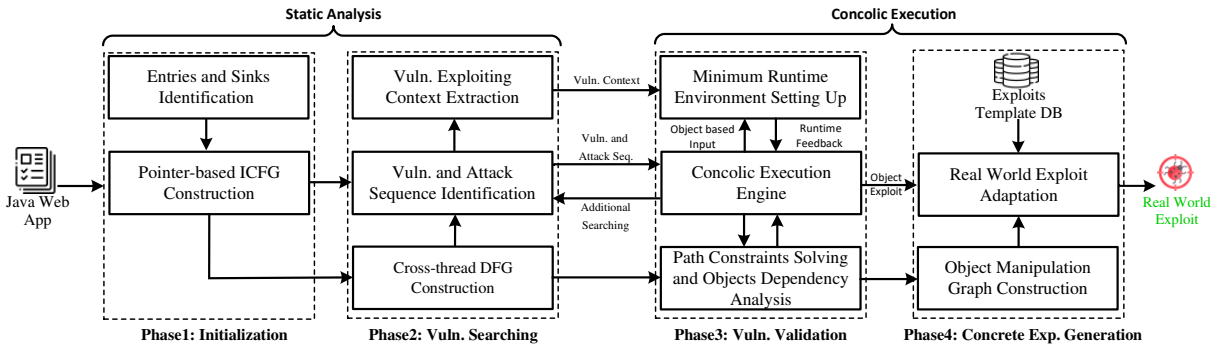

Figure 5: The Overall Architecture of JAEX with Four Phases.

generates exploits for them.

In Phase 3, JAEX first set up the minimum runtime environment needed for concolic testing of each potential vulnerability. After that, JAEX will generate initial input based on the information extracted from static analysis and pass them to the corresponding entry method for driving concolic execution. During the execution process, JAEX collects constraints along the execution path through instrumentation and solves them to help the execution engine generate the next input. Additionally, if any conditional statements are found to be unaffected by the current inputs during this process, the concolic engine will feed this information back to the static analysis stage to refine path searching.

In Phase 4, JAEX generates exploits for verified vulnerabilities. Specifically, it utilizes a Cross-thread Object Manipulation Graph (COMG) to depict the relationships between the input, the shared objects and the sink's parameters, as well as the constraints that should be satisfied along the execution path. Then, it constructs the sequence of HTTP requests required for exploitation based on expert-derived exploit templates and the obtained COMG.

Analyzing the Motivating Example. We now describe how JAEX works on the motivating example.

In the static analysis stage, after phase 1 and constructing the Cross-thread DFG like Figure 4, JAEX first identifies a potential vulnerability path in the `Daemon Thread II`, i.e., `enable` at Line 46 in Figure 2 that reaches the sink at Line 54. Then, because the `ObQueue` object associated with the background thread method `enable` is currently uncontrollable. JAEX conducts cross-thread analysis with the goal of understanding how to influence it through user inputs. With the ICFG and the Cross-thread DFG, JAEX finds that the `enable` method is invoked by another background thread method, `refreshEventQueues` (Line 21). Within this method, the `ObQueue` object needed by the aforementioned background thread is created through `e`, which is also an uncontrollable object but can be found have cross-dataflow from web entry `addEventHandler` (Line 16) in Figure 2. At this point, the `ObQueue` object associated with the background thread and `e` object, previously marked as uncontrollable objects have converged. JAEX iterates through multiple rounds of analysis until all uncontrollable objects along the path converge, which we refer to as a sink-guided, on-demand iterative algorithm, and detect the potential vulnerabilities. Besides, based on the order of convergence, a sequence of web entry methods invocations that can represent the request sequence is derived.

In the concolic execution stage, JAEX performs concolic execution based on the web entry invocation sequence and expected execution paths. Specifically, the concolic execution engine applies Step-forward Strategy, that is, it executes these entries one by one, sets the aforementioned uncontrollable objects in the execution path of the next web entry as target points, and continue to execute subsequent web entry only if these uncontrollable objects can be influenced. During each web entry's execution, JAEX extracts and solves path constraints For example, when JAEX reaches Line 23 in Figure 2, it detects that the `active` field in the input `EventHandler` should be true and feeds this information back to the execution engine to construct the next input. Additionally, JAEX records the data dependencies among objects along the path. For instance, the `qn` at Line 40 originates from the concatenation of the `name` field of the `WorkflowDef` and `TaskDef` objects (i.e., `wfd` and `tDef`) inputted through `Request I` and `Request II`. Furthermore, `ObQueue` at Line 41 is associated with `qn`, and `ObQueue` is linked to the name of `EventHandler` at Line 24 through the key of `queueMap` at Line 24. JAEX will further represent these data relationships in the COMG (see Figure 3). Finally, once it reaches the sink, JAEX uses the COMG to generate the concrete exploits as seem in Figure 1.

3.2 Phase 1: Initialization for Vulnerability Detection

In this phase, JAEX identifies sources and sinks with pointer-based ICFG construction to facilitate subsequence vulnerability detection.

Pointer-based ICFG Construction. Since Java Web apps commonly involve a lot of dynamic features, e.g., reflection, dependency injection, etc., their CFGs contain many abstract objects and method invocations. To enhance the precision

Table 1: Entries/Sources Used in Static Analysis

Name	Category	Entry Type	Entry Pattern Numbers	Examples
Spring	Web Framework	Web	2	@Controller
Structs2	Web Framework	Web	2	XWorks2 APIs
Servlet	J2EE APIs	Web	2	Methods (e.g., doGet)
JAX RS/WS	J2EE APIs	Web	2	@POST
JDK Native	Libs	Thread	3	Methods (e.g., Tread.run)
Third Party	Libs	Thread	6	Asyn. APIs

and completeness of our analysis, we use pointer analysis to construct the necessary Inter-procedural Control Flow Graph (ICFG). To ensure our analysis is as sound as possible, we extended Andersen's pointer analysis algorithm to support common framework features in Java Web apps (e.g., support for Spring's Dependency Injection and AOP mechanisms). Additionally, for abstract type variables that cannot be resolved to specific types during static analysis, JAEX switches the graph construction algorithm to Class Hierarchy Analysis to ensure the analysis is as comprehensive as possible.

Entries/Sources Identification. For entries, we categorize them into two types, shown in Table 1:

- Regular web entry points, including the methods that handle HTTP request-mapped objects extracted from mainstream web application frameworks, e.g., `doGet` method of Servlet.
- Thread entry points, including Java's built-in threading mechanisms (e.g., `Thread.run`) and the asynchronous communication APIs provided by third-party frameworks and libraries.

As shown in Table 1, JAEX utilizes 17 patterns to identify these entries, summarized by analyzing five major web frameworks, including Spring, Struts2, Servlet, JAX RS and JAX WS. Given a Java Web app, JAEX would traverse all its classes and extract the corresponding web entries by matching these patterns. Noting that, to identify thread entry points, JAEX also performs control flow analysis on the Inter-procedural Control Flow Graph (ICFG) to determine whether the thread entry method has been invoked or if it runs as a background thread (such as methods in infinite loops).

Sinks Identification. For sinks, to make JAEX as generalizable as possible, we collected over 500 dangerous methods (e.g., command execution method, `Runtime.exec()`) as well as several framework-level dangerous method patterns (e.g., user-defined SQL API in ORM frameworks like Mybatis). JAEX identifies the call sites of these methods throughout the app as sinks for subsequent analysis.

3.3 Phase 2: Vulnerability Detection and Attack Request Sequence Identification

In this phase, JAEX detects potential vulnerabilities in the app as well as the entry invocation sequence, i.e., the attack request sequence to trigger them. Specifically, JAEX first constructs a Cross-thread Dataflow Graph (DFG) that represents the potential cross-thread dataflow to facilitate vulnerability detection. Then, it uses a sink-guided, on-demand, and iterative algorithm to search vulnerabilities and infer the sequence of requests required to trigger the vulnerabilities.

Constructing Cross-thread DFG. We first describe how to construct the cross-thread DFG, which represents the potential dataflow relation between different threads. Our observation is that cross-thread dataflows are connected by shared objects between threads. Based on this, our key insight is that these shared objects are usually persistent throughout the program's runtime lifecycle, and those that have cross-thread data interaction are always accessed through specific APIs for read and write operations. Therefore, by identifying those persistent objects as potential shared objects and whether these objects are read and written across different threads, the potential inter-thread data flow relationships can be uncovered.

For example, in Figure 2, the `EventHandler` object passed by the user through `addEventHandler` at line 16 can flow into different background thread methods at lines 22 and 53 through persistent database layer via `DB` operations. Similarly, the `queue` object created at line 24 can flow into the background thread method at line 52 through the persistent global static variable `qMap` at Line 20. Based on these relationships, JAEX identifies those shared objects and constructs the cross-thread DFG as shown in Figure 4.

Specifically, JAEX identifies four kinds of cross-thread dataflow mechanisms and uses them as patterns to identify shared objects and related access operations, and connect cross-thread dataflows:

(i) Global Variable Access. JAEX identifies three kinds of global variables and their access operations:

- Static variables. JAEX directly identifies Java methods that can access and modify them, extracting those methods for further analysis.
- Fields of background thread. JAEX identifies all thread classes within the app, then checks if these thread classes are instantiated and used as persistent background threads (e.g., by calling the thread's entry method in an infinite loop). Last, JAEX identifies and extracts the methods that operate on the fields of these persistent background thread objects.
- Singleton variables of Java Web framework. JAEX models the common-used Java Web development frameworks to help identify related singleton variables (such as Java Beans in a Spring IoC container), and then identifies the methods that directly operate on them.

(ii) Database Operation. JAEX models the APIs of mainstream database access (e.g., JDBC) and ORM frameworks (e.g., MyBatis, Hibernate, and etc.), In our practice, we found that there are generally two ways to access the database:

- Fixed API, accessing the database through fixed APIs

```
1 <!— ItemMapper.xml —>
2 <mapper namespace = "com.dao.ItemDao">
3   <select id = "selectItem" paramType = "int">
4     select * from item where id=#{id}
5   </select>
6 </mapper namespace = "com.dao.ItemDao">
```

Figure 6: An Example of Mybatis's Mapper

with SQL commands configured in codes. Take `exexute(sql)` as an example. It uses JDBC's `execute` API and passes a SQL string directly into it.

- Mapping API, accessing the database with SQL statements in configuration files. It will invoke specific user APIs based on the configuration. A typical example is MyBatis, which achieves this by configuring the `*Mapper.xml` file like Figure 6.

JAEX models the semantics of these APIs to understand their behaviors. Specifically, for fixed APIs, JAEX performs data flow analysis to extract the SQL statements from the API context and uses SQLParser [23, 24] to parse them. For mapping APIs, JAEX parses the external configuration filesto extract the APIs and their SQL statements. Moreover, in Java Web apps, some SQL statements are generated dynamically during the runtime. For these statements, JAEX will hook the modeled APIs (e.g., Mybatis's `getBoundSql`) to extract them and analyze their semantics during the concolic execution phase.

(iii) File Operation. JAEX performs detailed modeling of JDK's native file operation APIs and also models common APIs from third-party libraries (i.e., 60 APIs in total). For these APIs, JAEX uses fine-grained data flow analysis to determine the filenames passed into them, thereby assessing whether the files being operated on have the potential to transmit data across threads. For example, for a file read operation like `new File(fileName);`, JAEX conducts backward data flow analysis from this call site to identify the value of `fileName`. If it later encounters a `new File` operation with the same file name as the parameter, JAEX considers there to be a potential data flow relationship between these two sites.

(iv) Asynchronous Communication: JAEX also models the common asynchronous communication mechanisms in Java Web apps, which can be categorized into two types:

- JDK-native and third-party asynchronous frameworks, for example, JDK's `CompletableFuture` APIs. For another example, RxJava framework.
- Asynchronous mechanisms provided by web frameworks, e.g., Spring's `ApplicationEvent`

JAEX analyzes the invocation of these Asyn. APIs within the app to infer the dataflows and treat those objects transfer through these APIs as shared objects.

Vulnerability and Attack Sequence Identification. In this part, we explain how JAEX detects potential vulnerabilities and infers related attack request sequences in detail using the pseudocode in Algorithm 1.

Algorithm 1: Vulnerability Path Searching

```
   Input: Entries Set S_e, Sinks Set S_s
   Input: Pointer-based ICFG G_a, Cross-thread DFG G_b
   Output: Vuln. Report Set S_vrp
   Output: Vuln. Execution Context Set S_vctx
1  S_vrp ← ∅, S_vpInit ← ∅;
2  for sink in S_s do
3      S_vpInit ← getPathTo(sink);
4  for vPath in S_vpInit do
5      vFullPath ← fullPathSearch(vPath);
6      if vFullPath ≠ ∅ then
7          S_vrp ← S_vrp ∪ collectVPathReport(vFullPath);
8          S_vctx ← S_vctx ∪ collectVPathCtx(vFullPath);
9  return S_vp, S_vctx;
10 function fullPathSearch(path):
11     fullPath ← ∅;
12     unControlSymbs ← getUncontrolSymb(path);
13     if unControlSymbs == ∅ then
14         S_vp ← S_vp ∪ path;
15         S_vctx ← S_vctx ∪ collectVPathCtx(path);
16     while unControlSymbs ≠ ∅ do
           // we first handle thread symbols
17         unControlSymbs ← prioritizeThreadSymb(unControlSymbs);
18         symb ← getOne(unControlSymbs);
19         newPath ← getPathForSymb(symb);
20         assertEmpty(newPath);
21         fullPath ← fullPath ∪ newPath;
22         unControlSymbs ← unControlSymbs − symb;
23     return fullPath;
24 function getPathForSymb(symb):
25     symbSink ← getSinkFor(symb, G_a, G_b);
26     path2Sink ← getPathTo(S_e, SymbSink);
27     fullPath2Sink ← fullPathSearch(path2Sink);
28     return fullPath2Sink;
```

JAEX first performs a simple source-to-sink taint analysis to obtain the initial vulnerability paths, S_{vpInit} (Lines 1-3). Then, JAEX iterates through each *vPath* and uses the `fullPathSearch` function for more detailed analysis. Specifically, `fullPathSearch` first checks if there are any variables along the path from source to sink that cannot be directly controlled by external inputs from the source (Line 12). If no such variables exist, the path is considered a potential complete vulnerability exploitation path (Lines 13-15). If such variables do exist, JAEX records them as guidance symbols (which will also guide the subsequent concolic execution process) and searches the app for paths that can taint these guidance symbols through external inputs (Lines 17-22). Note that JAEX prioritizes handling thread-related guidance symbols (Line 17), as they are often the most crucial element in vulnerabilities that require cross-thread triggering. The function `getPathForSymb` (Line 24) is responsible for finding paths that affect these guidance symbols by searching through the global ICFG and Cross-thread DFG for statements that can taint these symbols. It first finds statements that manipulate these symbols, and these statements are then treated as sinks for further analysis. If such paths are found, they are passed back to `fullPathSearch` for more comprehensive path analysis (Lines 26-28).

During the path-searching process, JAEX records the following key information:

- *Web entry invocation order*. We first identify the web entry W_d directly triggers the path. Then, we identify if there is another web entry W_O which could taint the guidance symbols of W_d. Then, we label W_O should be invoked before W_d. Note that, we will iteratively find a web entry for all guidance symbols.
- *Web entry parameters*. This means that which fields of the parameters are used in the execution path, which can be used for the lazy generation strategy of concolic execution.

For example, in Figure 2, after identifying this vulnerability with static analysis, JAEX can further infer its web entry invocation order, i.e., `(addTaskDef, addWorkflowDef, addEventHanlder, startWorkflow)`. Then, it infers the needed parameters for them. Take `startWorkflow` as an example. Since the field `name` of the parameter `req` is been accessed in Line 31, JAEX would give it a concrete value when generating initial variables for concolic execution.

3.4 Phase 3: Vulnerability Path Validation via Concolic Execution

In this phase, JAEX conducts vulnerability verification via concolic execution, which is guided by the vulnerability paths and web entry invocation sequence identified in the previous phase. Specifically, JAEX employs a step-forward concolic execution strategy, which is specifically designed to trigger vulnerabilities involving cross-thread dataflows. During the execution, JAEX accurately tracks the execution positions of variables to extract the constraints required for satisfying specific path branches and to detect dependencies between objects, preparing for subsequent exploit generation. We now describe our design in four key aspects.

(i) Concolic Variable. To accurately track user input during execution (e.g., where a certain variable is from and what operations it has met), JAEX defines a concolic variable as a quadruple: `(Type, Value, Source, Symbolic Expression)`. `Type` represents the variable's data type. `Value` is the variable's concrete value during runtime. `Source` indicates the origin of the variable, such as the method and the parameter it comes from (see Appendix B for the formal definition). `Symbolic Expression` includes the path constraints of the variable encountered along the runtime execution.

JAEX supports eight primitive types of Java (e.g., int, etc.), String, Arrays, Collections, Maps, and Object types. Specifically, We categorize object types into two groups. The first is Plain Old Java Objects (POJOs), which are commonly used to hold and pass user inputs, such as the `EventHandler` in Line 4 of Figure 2. POJOs typically *only* contain fields that are *simple primitive types* or other POJOs, making them relatively simple for JAEX to handle. JAEX recursively generates concolic variables for their fields. The second group includes more complex objects that have many non-primitive fields and are hard to model generally, such as `HttpServletRequest`, which maps to HTTP requests in J2EE Servlet. For these objects, JAEX models their fields and APIs specifically to support concolic execution. In practice, modeling the objects in the mainstream web frameworks listed in Table 1 has been sufficient to handle all the scenarios we encountered.

Take the parameter `eh` of the `addEventHandler` method at Line 16 in Figure 2 as an example. JAEX first generates an empty `EventHandler` object with its concolic variable: `(Type: EventHandler, Value: eh, Source: addEventHandler/0, Expression: eh)`. Based on static analysis, JAEX detects that the `name` field of this object is involved in the expected path (Figure 2, Line 24). Consequently, JAEX generates a concolic variable for the `name` field as well: `(Type: String, Value: "dummy", Source: addEventHandler/0.name, Expression: "dummy")`. This step ensures that all relevant fields are properly tracked during the concolic execution process.

(ii) Concolic Operation. JAEX defines a set of rules to handle various situations encountered by concolic variables during execution, especially related to the external resources like databases that are commonly ignored by prior works for implementing more comprehensive concolic execution.

Firstly, during execution, a concolic variable may interact with other concolic variables or concrete values. In this situation, JAEX first converts all variables into concolic variables. Take the statement `String qName = wfd.name + ":" + tDef.name;` at Line 40 in Figure 2 as example, JAEX first converts the `":"` into a concolic variable `(Type: String, Value: ":", Source: EventTaskExec.execute#Line46, Expression: ":")`. Secondly, JAEX calculates the result based on the operations involved in the specific statement. We categorize these operations into the following four types:

- *Regular Operators*: This refers to common operators in programming languages like `+, -, !`. JAEX hooks all of these JVM opcodes and handles the concolic variables flow through them based on their semantics.
- *Common Operation APIs*: This refers to methods like `StringBuilder.append` in JDK and APIs of popular third-party libraries, such as `StringUtil.concat` in Hutool.
- *External Resource-related APIs*: These APIs manipulate external resources like databases or files. JAEX models APIs encountered in practice and maintains a global state for each type of external resource. JAEX then returns the correct concolic variables based on the runtime state, API call context, and semantics. For example, in database operations, JAEX simulates a database (e.g., MySQL) composed of Java objects, reflecting the state changes during CRUD operations. JAEX instruments the key statements in the related APIs to capture the expected SQL expression and the anticipated return values,

allowing it to understand the API's complete semantics. By combining the passed parameters and the semantics, it returns specific concolic variables from the simulated database.

- *Unmodeled Operations*: For operations that have not been modeled, such as methods from third-party libraries introduced by developers, JAEX directly executes these methods, encapsulates the return values and transforms them to concolic variables.

(iii) Step-forward Execution Strategy. JAEX employs a step-forward execution strategy for concolic execution. It follows the web entry invocation sequence inferred from our static analysis, constructs the necessary method parameters, and invokes the corresponding methods. Each web entry's execution is guided by guidance symbols, which are concolic variables identified during static analysis that must be influenced by the current web entry to satisfy certain path constraints in the execution path triggered by another web entry. During each entry's execution, JAEX adopts the depth-first exploration strategy before reaching the sinks or points that can influence the guidance symbols. For conditional branches encountered during exploration, JAEX categorizes them into two types: mandatory and non-mandatory. The former is identified during the static analysis phase through control flow graph analysis, if these branches cannot be satisfied, JAEX immediately terminates the execution of the current entry. For the latter, JAEX skips unsolvable branches and prioritizes the execution of solvable ones. Once all required guidance symbols for the current path are influenced, JAEX marks the task for that entry as completed and moves on to the next entry for testing.

However, this step-forward process may encounter two types of obstacles: (i) guidance concolic variables influenced by earlier steps do not satisfy certain constraints on the current path, and (ii) new uncontrollable symbols are discovered during execution. Specifically, for (i), JAEX identifies the origin `entry` of unsatisfied guidance symbols based on their identifier. It then resolves the issue by using the constraint solver and re-executing the corresponding `entry`. For symbols in (ii), JAEX's concolic execution engine feeds them back to the static analysis module to assist in path searching. If no suitable path is found, JAEX considers the current vulnerability to be a false positive.

(iv) Constraints and Object Dependency Analysis. During execution, JAEX may find that some conditional statements should involve concolic variables. In these cases, JAEX uses a constraint solver to determine the required values for them. For instance, in the motivating example (Figure 2, Line 22), the variable `e` is obtained through a database API, making it a concolic variable. Based on the static analysis results, we know that `e` is the same as `eh` from Line 17. Therefore, JAEX can solve for the value of `eh` and determine that its `active` field should be set to `true` during input construction.

In addition, JAEX collects data dependency relationships between objects along intra- and inter-thread execution paths. For instance, the `ObQueue` object at Line 41 is associated with the `ObQueue` object at Line 20 through the global `qMap` variable via the key of the map. The key for the `ObQueue` object created at Line 24 is derived from the `name` field of the external input `e`, while the key read at Line 41 is generated from the concatenation of the `name` fields of `wfd` and `tDef`. Thus, JAEX identifies a dependency between the external input `eh` and the external inputs `wfd` and `tDef`.

JAEX records these constraints and dependencies related to external inputs for use in the next phase of Cross-thread Object Manipulation Graph construction.

3.5 Phase 4: COMG-aided Concrete Exploit Generation

In this phase, JAEX first builds a Cross-thread Manipulation Graph (COMG) to guide exploit generation. Next, JAEX retrieves appropriate exploit templates from our Exploits Templates Database based on the current vulnerability type. Guided by the COMG, it generates payloads that satisfy the necessary constraints. Finally, JAEX produces the correct HTTP exploits. We now describe the three core components.

Exploits Templates Database. The Exploits Template is defined as follows:

```
[EscapeString1]exploit[EscapeString2]
```

Here, EscapeString1 and EscapeString2 are used to escape the exploit into the required format, typically consisting of syntax symbols such as `}`, `//`, depending on the context.

We collected more than 150 real-world exploits and compiled 37 exploit templates for 7 types of vulnerabilities, which we have made available in an anonymous git repository* for review. The database maintains one exploit template for different exploitation methods of the same vulnerability type, allowing variations by replacing predefined parts to produce different exploit results.

COMG Construction. The COMG is built based on the constraints and dependency information retrieved from the concolic execution phase. JAEX combines this information with a pre-modeled mapping between HTTP requests and Java object exploits for the specific web framework, and finally obtains this data structure that represents the core elements needed to generate concrete exploits. As illustrated in Figure 3, the COMG provides the following five types of information. ❶*HTTP Description*. Maps HTTP requests to web entry parameters or runtime objects. JAEX obtains this mapping based on the web framework modeled in Table 4. ❷*Invocation Order*. Specifies the sequence in which HTTP requests are sent. ❸*Field Dependencies*. Defines dependencies between specific fields across different requests.

*https://anonymous.4open.science/r/SEC25_EXPDB-25C0/

❹*Constraints of Payloads*. Outlines the constraints that request fields must satisfy. ❺*Sink Position*. Identifies the locations where attack payloads can be injected.

Real World Exploit Adaptation. Using the COMG, JAEX converts the exploit objects obtained from concolic execution into concrete HTTP requests. We introduce this process using Request III in Figure 3 as an example. First, based on the HTTP description, JAEX determines that a POST request to `/eventhandle` is needed, containing JSON-formatted data. The JSON structure must match that of the `EventHandler` object `eh`. According to the field dependencies and constraint information, `eh.active` is set to `true`, and `eh.name` is a concatenation of `tDef.name` from Request I, `":"`, and `wfDef.name` from Request II. Next, the `condition` field is set to the attack payload. In this case, it is a sink for an expression injection vulnerability, JAEX selects exploit candidates from the Exploit Database, e.g., `Runtime.getRuntime().exec("xxx");`. Finally, once all payloads for the requests is constructed, JAEX organizes them in the correct sequence, generating an exploit like the one shown in Figure 1

4 Evaluation

We evaluated JAEX with the following research questions:

- RQ1: How does JAEX perform in vulnerability detection and exploitation on benchmarks compared to state-of-the-art tools?
- RQ2: How effective is JAEX in detecting and exploiting 0-day vulnerabilities in real-world apps?
- RQ3: What is the performance overhead of JAEX during the analysis process?

Implementation. The static analysis of JAEX is built on top of tai-e [25], which is an object-, field-, array- and context-sensitive static analysis framework. Excluding the code of tai-e itself and third-party packages, we implemented over 20,000 lines of new code. JAEX's concolic execution module drew inspiration from JPF [26] and Phosphor [27]. It contains more than 8,000 lines of code, using Java Agent, the ASM bytecode editing framework, Java dynamic proxy technology, and Java reflection. Before concolic execution, JAEX sets up the minimum runtime environment or context for each vulnerability. Specifically, JAEX maintains two kinds of context: (i) Internal Context. This refers to the call context needed to execute the web entry methods. (ii) External Context. This refers to the resources that must be obtained from external sources via specific APIs during the execution of the vulnerability paths. For (i), we referred to the approaches mentioned in previous works [18]. For (ii), JAEX hooks these APIs and concolized their return values.

Environment Setup. All experiments were conducted on a Linux ESXi virtual machine instance equipped with a 64-core Intel(R) Xeon(R) Gold 5218 CPU at 2.30 GHz and 128 GB of RAM. The instance is running Ubuntu 18.04.5 LTS with jdk-17.0.10 installed. To verify the effectiveness of the exploits generated by JAEX, we manually verify them by setting up all the apps under test using Docker 24.0.2 and Docker Compose 1.29.2.

Datasets. To assess the capabilities of JAEX, we constructed the following two datasets:

(i) Benchmark. Our benchmark primarily comprises historical vulnerabilities from real-world scenarios. We collected these vulnerabilities through the following steps.

- Step 1: We collected Java Web apps used in prior studies [19, 28] and popular Java Web apps from renowned open-source organizations on GitHub, such as Apache, using keywords like 'Java CMS' and 'Java Web'.
- Step 2: We searched for historical vulnerabilities in these apps within prominent vulnerability databases, such as the National Vulnerability Database (NVD). As a result, we obtained 92 vulnerabilities with publicly available proofs of concept (PoCs), which affect 16 apps.

(ii) Real-world Apps. We collected 25 popular Java Web apps, whose stars exceed 500, for evaluating JAEX's capabilities on 0-day vulnerability mining. These apps are actively maintained by their developers, as they are updated frequently, and all apps are tested in their latest versions.

Baselines. We used the following baselines for comparison:

- Witcher [19]. A grey-box fuzzing tool that supports Java Web apps. However, based on our practice, Witcher's built-in crawler was unable to effectively crawl the web entries of those apps in the benchmark. Therefore, to make the comparison fairer, we extracted all Web entries through static analysis for Witcher.
- Joern [20]. An open-source white-box code analysis platform that supports Java. We configured the Web entries and sinks using the Scala environment and interfaces provided by Joern, and leveraged its built-in taint analysis module to perform vulnerability detection.
- JAEX-NOCT. Besides, we also implemented a JAEX without the cross-thread analysis module called JAEX-NOCT as another baseline tool for ablation study.

4.1 RQ1: Comparation with Baselines

Overall Results. In this section, we compare JAEX with the baselines. Table 2 shows the overall results. In this table, we present the number of detected vulnerabilities (Detected) and successfully exploited vulnerabilities (Exploited) for Joern, Witcher, JAEX-NOCT, and JAEX. Note that we mark the columns of Exploited of Witcher and Joern as 'N/A' because neither of them has the capability to generate concrete exploits.

Overall, both JAEX-NOCT and JAEX significantly out-

Table 2: Comparison of the number of vulnerability detection and exploits generated by JAEX and the state-of-the-art tools across the 92 vulnerabilities among 16 popular Java Web apps and the well-known Java vulnerability testbed WebGoat. Note that the numbers in parentheses indicate true positives (TP) on ground truth.

Application	Version¶	Known Vuln.♣ (Gound Truth)		Joern				Witcher				JAEX-NOCT				JAEX			
				Detected		Exploited ♠		Detected		Exploited ♠		Detected		Exploited		Detected		Exploited	
		Total	CT♢	Total	CT	Total	CT	Total	CT	Total	CT	Total	CT	Total	CT	Total	CT	Total	CT
Real-world apps																			
ActiveMQ	5.1x.x	3	3	4 (1)	1 (1)	N/A	N/A	0	0	N/A	N/A	6 (1)	1 (1)	0	0	15 (2)	10 (2)	2	2
Archiva	2.2.x	2	1	22 (2)	1 (1)	N/A	N/A	0	0	N/A	N/A	12 (2)	1 (1)	1	0	12 (2)	1 (1)	2	1
CrushFTP	10.x	2	0	8 (0)	0	N/A	N/A	0	0	N/A	N/A	20 (2)	0	2	0	35 (2)	15 (0)	2	0
DolphinSecheduler	2.x, 3.x	8	5	29 (3)	0	N/A	N/A	0	0	N/A	N/A	35 (3)	0	3	0	58 (8)	23 (5)	8	5
Elastic Search	1.x	4	3	11 (0)	0	N/A	N/A	0	0	N/A	N/A	44 (4)	3 (3)	1	0	51 (4)	10 (3)	4	3
Halo	1.x	6	2	21 (4)	0	N/A	N/A	0	0	N/A	N/A	11 (4)	0	4	0	17 (6)	6 (2)	6	2
Hadoop	3.0.0	1	1	12 (0)	0	N/A	N/A	0	0	N/A	N/A	4 (0)	0	0	0	7 (1)	3 (1)	1	1
JeecgBoot	3.2.x	5	0	89 (3)	0	N/A	N/A	0	0	N/A	N/A	47 (5)	0	5	0	63 (5)	16 (0)	5	0
Apache Kylin	3.x	8	5	27 (2)	0	N/A	N/A	0	0	N/A	N/A	13 (3)	0	3	0	23 (8)	10 (5)	8	5
MCMS	5.1.x, 5.2.x	27	2	52 (12)	0	N/A	N/A	2 (2)	0	N/A	N/A	86 (26)	1 (1)	25	0	94 (27)	9 (2)	27	2
MeterSphere	1.x, 2.x, 3.x	9	1	63 (8)	0	N/A	N/A	0	0	N/A	N/A	52 (9)	1 (1)	8	0	69 (9)	16 (1)	8	0
NRM3‡	3.x.x	5	1	16 (2)	0	N/A	N/A	0	0	N/A	N/A	68 (4)	1 (1)	3	0	98 (4)	31 (1)	3	0
RocketMQ	4.5.x, 5.1.x	2	1	7 (0)	0	N/A	N/A	0	0	N/A	N/A	15 (2)	2 (1)	1	0	17 (2)	4 (1)	2	1
Ruoyi	4.x	6	1	84 (4)	0	N/A	N/A	0	0	N/A	N/A	13 (5)	0	5	0	18 (6)	5 (1)	6	1
Jeesite	1.x	3	2	29 (3)	2 (2)	N/A	N/A	0	0	N/A	N/A	14 (3)	2 (2)	1	0	19 (3)	7 (2)	3	2
Vivo Moonbox	1.0.0	1	1	4 (0)	0	N/A	N/A	0	0	N/A	N/A	6 (0)	0	0	0	7 (1)	1 (1)	0	0
Total		92	30	478 (44)	4 (4)	N/A	N/A	2 (2)	0	N/A	N/A	427 (73)	12 (11)	62	0	603 (90)	160 (28)	87	25
Synthetic Java Web Application Vulnerability Testbed																			
WebGoat	v2023.8	12†	0	16 (12)	0	N/A	N/A	11	0	N/A	N/A	17 (12)	0	12	0	17 (12)	0	12	0
Total		12	0	16 (12)	0	N/A	N/A	11	0	N/A	N/A	17 (12)	0	12	0	17 (12)	0	12	0

¶ Since the vulnerabilities in the dataset span multiple different historical versions, we have consolidated multiple versions of the same application into a single row in the table, listing only the corresponding major version.
♣ We have compiled the information regarding these vulnerabilities into an anonymous repository, available at https://anonymous.4open.science/r/SEC25_EXPDB-25C0/.
♠ Since neither Joern nor Witcher can generate exploits, we performed manual analysis on their detected vulnerabilities to determine their exploitability and marked their columns of Exploited as 'N/A'.
† WebGoat contains over 40 vulnerabilities, including issues like access control and weak passwords. In this paper, similar to previous work, we focus only on the 12 Injection-style vulnerabilities included in the testbed.
‡ NRM3 stands for Nexus Repository Manager 3, and here we are using its open-source version.
♢ CT stands for Cross-thread (Vulnerabilities).

performed Witcher and Joern. In the real-world vulnerability benchmark, JAEX-NOCT detected 427 potential vulnerabilities with 73 true positives and generated exploits for 62 of them. In comparison, JAEX detected 603 potential vulnerabilities with 90 true positives and generated 87 exploits. Witcher and Joern only detected two and 44 out of the 92 vulnerabilities, respectively, and were unable to generate any exploits for them. In WebGoat, the performance of Witcher is much better and it detected 11 out of 12 vulnerabilities. This is because most of the vulnerabilities in WebGoat are quite simple and can be triggered directly without the need for inputs to meet complex constraints.

False Positive Analysis. Joern has large false positive rates (90.8%) based on our manual verification, while JAEX does not. Both JAEX and JAEX-NOCT detected all the vulnerabilities that Joern found. Although they have relatively high false positive rates with static analysis (85.07% and 82.90%, respectively), their use of concolic execution for validation ensures that the final results have 0 false positives. Witcher, as a fuzzing tool, also produces 0 false positives. However, as shown above, JAEX detected (and exploited) 43X more vulnerabilities than Witcher.

False Negative Analysis. As illustrated in Table 2, both Witcher and Joern have significantly high false negative rates, being 97.8% (90/92) and 51.2%(48/92), respectively. During Witcher's execution, we captured the test cases generated via traffic sniffing and conducted a manual analysis. This analysis revealed the following reasons for Witcher's poor performance on the current dataset: Firstly, Witcher treats all HTTP parameters as a whole and applies AFL's random byte-level mutation strategy, which is unable to produce valid test cases. Secondly, real-world Web apps often contain requests with numerous parameters that have varying types and semantics. This complexity further reduces Witcher's chances of mutating test cases that can trigger vulnerabilities. For Joern, the majority of the false negatives (26/48) come from the vulnerabilities that need cross-thread dataflows involved. However, it still detected 4 cross-thread vulnerabilities since their sinks can be directly affected by the input of the current entry.

JAEX-NOCT detected 73 vulnerabilities but only generated exploits for 62 of them. Upon manual inspection, we confirmed that all the missed cases are cross-thread vulnerabilities. Specifically, the 19 vulnerabilities that JAEX-NOCT failed to detect are because the parameters at the sinks required to be tainted by cross-thread dataflows.

JAEX achieved a vulnerability detection rate of 97.8% (90/92) during static analysis and generated exploits for 87 of the detected vulnerabilities. Although it performed the best among the four tools, it still missed 2 vulnerabilities in the static analysis stage and 3 vulnerabilities in the concolic execution stage. We summarize the reasons as follows:

- *Inadequate language feature support*. Some language features could not be fully supported by static analysis. For example, the JMX mechanism in ActiveMQ causes

Table 3: JAEX's Performance on Real-world apps. We only present those exploited apps.

Application¶	App. Overview		Vulnerability Discovery				Runtime Overhead		
	Stars	LoCs	Detected (True Positive)	Exploited	Cross-thread	Status	Graph† Construction	Vuln. Path Searching	Concolic Exectuion
[A1]C****t	19.2k	10.2k	13 (2)	2	2	Confirmed	8min20s	25min56s	23min06s
[A2]C****r	14.3k	85.3k	17 (3)	3	3	Confirmed	10min21s	36min20s	41min02s
[A3]S****r	9.8k	9.9k	18 (3)	3	1	Confirmed	3min57s	12min17s	32min31s
[A4]D***y	3.1k	92.7k	36 (13)	13	1	Confirmed	12min24s	29min20s	1h02min43s
[A5]J***m	1.6k	310.4k	4 (1)	1	0	Reported	30min54s	24min11s	7min47s
[A6]J***m	1.4k	55.3k	56 (5)	5	3	Confirmed	10mins07s	41min22s	2h22min12s
[A7]A***o	0.7k	191.8k	12 (1)	1	1	Confirmed	21min23s	51min22s	27min38s
[A8]A***t	4.7k	40.3k	21 (4)	4	0	Reported	9min15s	19min25s	34min48s
[A9]D***o	3.5k	57.2k	10 (2)	2	1	Reported	9min47s	20min46s	19min20s
[A10]Q***s	0.6k	117.6k	7 (1)	1	1	Reported	18min34s	14min22s	17min02s
Total	-	-	-	35	12	-	-	-	

¶ We have reported all the vulnerabilities and received confirmations from the vendors. For ethical consideration, we choose to withhold the names of these real-world apps until the developers agree to public them.
† The Graph means Pointer-based ICFG and Cross-thread CFG

the flow interruption, which made JAEX miss it.

- *Lack of domain knowledge*. One vulnerability in NRM3 can not be exploited since it needs the knowledge of how to exploit OrientDb. This could be fixed by arming JAEX with specific knowledge about OrientDb.
- *Unsupported constraints*. Some constraints are overly complex. For example, in the case of Vivo Moonbox, there are numerous complex string operations such as concatenation and slicing, which lead to constraint-solving failures.

Contribution of Cross-thread Analysis. We now describe how such cross-thread analysis benefits vulnerability detection and exploitation in Java Web apps two-fold.

Firstly, based on the comparison results between JAEX and baselines, JAEX benefits from the presence of the Cross-thread DFG, enabling more comprehensive static analysis and detecting more potential vulnerabilities.

Secondly, through cross-thread analysis, JAEX can derive guidance symbols to direct concolic execution, thereby generating more comprehensive exploits. Let's take Elastic Search in the benchmark as an example: although JAEX-NOCT can detect all cross-thread vulnerabilities, it fails to generate concrete exploits for them since there are certain path constraints that need to be addressed by triggering cross-thread dataflows. JAEX can identify these constraints, find proper paths, and guide concolic execution to execute them.

4.2 RQ2: Exploiting Real-world apps

In this section, we evaluated JAEX on 25 popular Java Web apps, the results are summarized in Table 3. Overall, JAEX ultimately discovered 35 zero-day vulnerabilities across 10 of them. Among these, 12 require the exploitation of cross-thread dataflow to be triggered. To further demonstrate how

```
1  /** Web Entry for Request I **/
2  void setBusinessConfig(Request req) {
3  ❶if (req.action == 'ADD') {
4    ItemConfig conf = req.getItemConfig();
5    DB.setBusinessConfig(req.domain, config);
6   }
7  }
8  /** Web Entry for Request II **/
9  void viewBusinessItem(Request req) {
10 ❷if (req.action == 'VIEW') {
11   ItemConfig config = DB.getBusinessConfig(req.name);   Req_I_to_II (conf)
12   List baseLines = prepareCustomBaseLines(req.name, config);
13   String pattern = config.getPattern();
14   String repacePattern = "${" + req.name + "," + config.title + ",AVG}"❸
15   calculate(pattern, repacePattern, baseLines);
16  }
17 }
18 void calculate(String pattern, String repPattern, List infos) {
19  pattern = pattern.replace(repPattern, infos.toString);❸
20  JexlExpression e = jexl.createExpression(pattern);
21  Number result = (Number) e.evaluate(null);
22 }
```

(a) Vulnerability Code in Case Study 1

```
# Request I
POST /config?action=ADD&domain=dog

itemConfig.title=poc&itemConfig.pattern=${dog,poc,AVG}.toString().class.forName('java.lang.Runtime').getRuntime().exec('touch /tmp/poc.txt')

# Request II
GET /show?name=dog
```

(b) Exploit Requests in Case Study 1

Figure 7: The vulnerable code and exploit of Case #1.

JAEX detects and exploits real-world vulnerabilities, we now conduct case studies using one typical real-world example.

Case #1: Cross-thread CI Exploiting Figure 7 illustrates a Code Injection (CI) vulnerability that needs to trigger cross-thread dataflows to exploit(blue dash between Line 5 and 11). The `pattern` field of `itemConfig` set by Request I, will flow into the sink at Lines 20-21 of the `calculate` method invoked by Request II and exploit the vulnerability, leading to code injection and ultimately remote command execution (RCE). Note that, Traditional taint-based static analysis tools

cannot detect this vulnerability because they only mark the parameter `req` of the entry method as tainted, and the fields within `req` do not directly flow to the sink at Lines 20-21. Similarly, Witcher fails to trigger the vulnerability because it cannot construct this two-step request sequence. Furthermore, even if the correct request sequence is constructed, exploiting this vulnerability remains challenging due to the multiple constraints and dependencies between fields that need to be satisfied. First, specific fields in Request I and Request II need to satisfy the constraints at Lines 3 (❶) and 10 (❷), respectively. Second, the `domain` field in Request I must match the `name` field in Request II to correctly trigger the cross-thread dataflows. Finally, the `pattern` field of the `ItermConfig` object set by Request I must contain a specific string format composed of the `name` field from Request II and the `title` field of the `itemConfig` object (❸) at the beginning (Line 14) of the attack payload, ensuring that the final exploit is valid. JAEX is able to detect this vulnerability through cross-thread analysis and successfully exploit it by using concolic execution to solve the necessary field constraints and dependencies.

4.3 RQ3: Runtime Overhead

In this section, we evaluated the runtime overhead of JAEX on the 10 vulnerable apps. The results are shown in Table 3. First, the time required for graph construction generally correlates with the number of web entries that should be analyzed. An example is A1 and A3. Since A1 has more web entries than A3, it consumes more time on graph construction. Second, the time spent on vulnerability searching is related to the number of potential vulnerabilities. For instance, A5, despite being the largest in code size, has relatively few sink points, leading to a shorter time in searching vulnerabilities. Last, the time spent on vulnerability validation and exploit generation is related to the number of cross-thread vulnerabilities. An example is A2, which consumes more time than A1 in concolic execution.

5 Discussions and Limitations

Extension to Micro-service Scenarios. A typical scenario prone to cross-thread vulnerabilities is Java Web apps under a microservices architecture, where an app is decomposed into multiple independent web services that communicate with each other through mechanisms like RPC or HTTP requests. JAEX can be easily extended to support vulnerability discovery in this kind of apps. An intuitive approach is to enhance static analysis by modeling common micro-service communication mechanisms in Java Web apps and deploying agents on different microservices as well as a centralized execution engine to orchestrate concolic execution across these microservice modules.

Enhancement of Vulnerability Detection and Validation. To support the detection of vulnerabilities that require triggering cross-thread dataflows for exploitation, which typically involve complex data flow relationships, constraints, and field dependencies, we enhanced both the static analysis and concolic execution capabilities to enable more accurate and comprehensive analysis. First, we enhanced Z3 constraint solving to support more comparison types (such as Java Collections, Maps, etc.) and comparison operations. Second, we support more precise type inference. Taking the commonly used generics mechanism in Java as an example, we perform data flow analysis to track whether generic variables are involved in `CAST` statements, thereby completing type inference. Third, we comprehensively model those data structures within the Java `Collectoin` framework(e.g., `ArrayList` and `HashMap`), which can help JAEX to conduct fine-grained dataflow analysis with Collection-item and Map-key Sensitive.

Modeling of Mainstream Frameworks and Libraries. In this paper, JAEX performs extensive modeling to support the analysis of Java Web apps, shown in Table 4 (in Appendix A). This is primarily because Java Web app development is typically based on existing frameworks and third-party libraries, which are often quite complex. Since we only focus on the developer's code within the apps, modeling these APIs helps us save significant analysis time. Additionally, these knowledge could be reused and support large-scale analysis of Java Web apps, as these mainstream frameworks and libraries are widely used in them. For example, Spring occupies over 80% [29] of the market share in Java Web application development.

Dynamic Java Features. Dynamic Java features, e.g., reflection and dynamic proxy, are traditionally challenging for vulnerability detection works [20] including JAEX. At the same time, these features are not commonly encountered in typical injection vulnerability detection scenarios. Our analysis shows that only two out of 92 real-world benchmark vulnerabilities involved these features. JAEX does miss the one related to the JMX mechanism, a complex reflection-based feature. We will leave more comprehensive support for these dynamic features as our future work.

Constraint Solving. JAEX adopts Z3 solver, thus also inheriting Z3's limitations, e.g., handling complex constraints like complex string operations, regular expressions, and hash value comparisons. Based on our analysis, the majority of path conditions in Java Web applications are relatively simple, such as variable non-null checks and string equality comparisons. Specifically, we selected MCMS, the application with the highest number of vulnerabilities among the benchmark apps, and conducted a manual analysis of its analyzed path conditions. Our results show that 60.24% (97/161) of the analyzed path conditions are simple ones without the aforementioned string operations. Note that most of those complex constraints in MCMS do not reside in the dataflow between vulnerability sources and sinks, thus leading to minimal impacts on JAEX. JAEX's small false negatives also manifest this.

Modeling of Vulnerability Pattern with Specific Domain Knowledge. The vulnerability supported by JAEX depends on the collected domain knowledge, e.g., the modeling of sink functions. As an analogy, prior works [20] suffer from the same problem since they also need to model vulnerabilities using manually-curated queries. We will leave the further generalization of JAEX to support more features of these vulnerability patterns as future work.

6 Related Work

Java Vulnerability Detection and Exploiting. There are many works in the field of Java Vulnerability Detection. However, most of these efforts focus on Java Object Injection (JOI) and Algorithmic Complexity (AC) DoS vulnerability. For JOI detection [12, 13, 15, 30, 31], the most representative work is JDD [15], which proposed a bottom-up approach to address the path explosion issues and a dataflow-aided approach to obtain the correct structure of injection objects. However, JDD cannot perform end-to-end vulnerability detection in web app scenarios, as it assumes that the attacker has direct access to dangerous deserialization methods, without considering how to trigger the vulnerability through web entry requests. In the context of AC DoS detection, two recent representative works are Acquirer [18] and HotFuzz [32]. The Acquirer supports analysis on web apps, but its approach is highly coupled with AC DoS detection, making it difficult to generalize to broader web app scenarios. Additionally, it cannot handle vulnerabilities that require cross-thread data flow interactions. HotFuzz, on the other hand, targets Java libraries, making it unsuitable for detecting vulnerabilities in Java Web apps. As for general Java Web vulnerability detection, the most relevant work is Witcher [19]. However, due to its lack of knowledge about constructing valid inputs for web apps, it often has a low code coverage in real-world scenarios. Additionally, there are some works [8, 28, 33] aimed at optimizing static analysis for Java Web apps. Compared to these works, JAEX is fully adapted to the characteristics of Java Web apps, making it more generalizable in this context. Besides, JAEX provides support for detecting cross-thread vulnerabilities, addressing a significant gap in traditional Java vulnerability detection in this area.

Vulnerability Detection and Exploiting in Other Web Languages. PHP and Node.js are the most studied web-related development languages in vulnerability detection. For PHP vulnerability detection [34–40], TChecker [39] is the state-of-art static analysis tool, which enhances PHP Joern [41] by fully supporting PHP's dynamic features ignored by prior works, thereby generating a more complete Code Property Graph to enable more comprehensive analysis. JAEX, on the other hand, models common cross-thread dataflows in Java Web apps, enabling the detection of more complex vulnerabilities. SYMPHP [40] extends the symbolic execution engine S2E and applies it to the PHP interpreter, achieving more comprehensive concolic execution for PHP Web apps. Atropos [2] improves code coverage during fuzzing by instrumenting super-global variables to capture runtime input constraints and generate high-quality test cases. However, unlike PHP apps, which have unified user input handling interfaces, Java Web apps involve more diverse input methods and more complex input formats, making it difficult to dynamically detect input requirements simply by instrumenting certain interfaces. Instead, we choose to model mainstream web development frameworks to obtain the correct format of input payloads. As for Node.js vulnerability detection [5–7, 34, 42–51], one of the most popular topics is the prototype pollution vulnerabilities [7, 42, 43]. Among these, the most related work is UoPF [6], it uses those undefined properties along the execution path to guide concolic execution to chain multiple prototype pollution vulnerability fragments, so-called gadgets, to reach sinks. Its high-level idea is similar to JAEX, which uses shared objects between threads to guide concolic execution. However, UoPF focuses on offline detection within Node.js template engines, while JAEX targets onilne Java Web apps. Additionally, JAEX's concolic execution engine supports a wider variety of types, making it more generalizable in Java Web app scenarios.

Dataflow Modeling in Android. There are many works [8–11, 52–54] have modeled dataflows similar to cross-thread dataflows in Android to enable more comprehensive analysis. For instance, Blueseal [9] models specific Inter-Process Communication (IPC) mechanisms in the Android framework, which connect the dataflows between user-called methods and corresponding framework-invoked methods, thereby supporting cross-app permission analysis. Similarly, many works [10, 52–54] conduct Inter-Component Communication (ICC) mechanisms modeling, such as Intent, in Android to ensure the consistency of dataflows during static analysis. Among these works, Amandroid [54] is one of the most representative. It is the first to consider the statefulness of ICC dataflows and connects these dataflows by analyzing the sequential order of method invocations. Similarly, JAEX also analyzes the invocation order of different web entries (methods) to trigger cross-thread dataflows correctly. Additionally, JAEX further analyzes data dependencies between these dataflows to enhance vulnerability detection and exploitation.

7 Conclusion

In this paper, we presented JAEX, the first automatic framework tailored for detecting vulnerabilities and automatically generating exploits in Java Web apps with the help of cross-thread dataflow analysis and concolic execution. Our evaluation shows that JAEX successfully detects and exploits 35 zero-day vulnerabilities in 10 popular open-source Java Web apps, which demonstrates its effectiveness.

Acknowledgement

We would like to thank anonymous shepherd and reviewers for their helpful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (62102093, U2436207, 62172105, 62202106, 62302101, 62172104, 62102091, 62472096, 62402114, 62402116). Min Yang is the corresponding author and a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). Yinzhi Cao was supported in part by National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361 and a Defense Advanced Research Projects Agency (DARPA) Young Faculty Award (YFA) under Grant Agreement D22AP00137-00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF or DARPA.

Ethics Consideration

Our research was conducted in local environments and avoided interacting with real-world systems. For all discovered vulnerabilities, we engaged in responsible disclosure, reporting all detected vulnerabilities to vendors and aiding in their remediation.

Open Science

To foster transparency and facilitate further research, we have made our software tool, datasets, and evaluation baselines publicly available at https://zenodo.org/records/14723855.

References

[1] Belgian defense ministry confirms cyberattack through log4j exploitation. https://www.zdnet.com/article/belgian-defense-ministry-confirms-cyberattack-through-log4j-exploitation/.

[2] E. Güler, S. Schumilo, M. Schloegel, N. Bars, P. Görz, X. Xu, C. Kaygusuz, and T. Holz, "Atropos: Effective fuzzing of web applications for server-side vulnerabilities," in *USENIX Security Symposium*, 2024.

[3] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "{NAVEX}: Precise and scalable exploit generation for dynamic web applications," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 377–392.

[4] S. Neef, L. Kleissner, and J.-P. Seifert, "What all the phuzz is about: A coverage-guided fuzzer for finding vulnerabilities in php web applications," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1523–1538. [Online]. Available: https://doi.org/10.1145/3634737.3661137

[5] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining node. js vulnerabilities via object dependence graph and query," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 143–160.

[6] Z. Liu, K. An, and Y. Cao, "Undefined-oriented programming: Detecting and chaining prototype pollution gadgets in node. js template engines for malicious consequences," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 121–121.

[7] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. N. Venkatakrishnan, and Y. Cao, "Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1059–1076.

[8] A. Antoniadis, N. Filippakis, P. Krishnan, R. Ramesh, N. Allen, and Y. Smaragdakis, "Static analysis of java enterprise applications: frameworks and caches, the elephants in the room," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 794–807.

[9] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek, "Information flows as a permission mechanism," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 515–526.

[10] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*, vol. 15, no. 201, 2015, p. 110.

[11] S. Calzavara, I. Grishchenko, and M. Maffei, "Horndroid: Practical and sound static analysis of android applications by smt solving," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 47–62.

[12] S. Rasheed and J. Dietrich, "A hybrid analysis to detect java serialisation vulnerabilities," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1209–1213.

[13] S. Cao, B. He, X. Sun, Y. Ouyang, C. Zhang, X. Wu, T. Su, L. Bo, B. Li, C. Ma *et al.*, "Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2726–2743.

[14] P. Srivastava, F. Toffalini, K. Vorobyov, F. Gauthier, A. Bianchi, and M. Payer, "Crystallizer: A hybrid path analysis framework to aid in uncovering deserialization vulnerabilities," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1586–1597.

[15] B. Chen, L. Zhang, X. Huang, Y. Cao, K. Lian, Y. Zhang, and M. Yang, "Efficient detection of java deserialization gadget chains via bottom-up gadget search and dataflow-aided payload construction," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 150–150.

[16] P. Awadhutkar, G. R. Santhanam, B. Holland, and S. Kothari, "Discover: detecting algorithmic complexity vulnerabilities," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1129–1133.

[17] Y. Liu, M. Zhang, and W. Meng, "Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1468–1484.

[18] Y. Liu and W. Meng, "Acquirer: A hybrid approach to detecting algorithmic complexity vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2071–2084.

[19] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, "Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities," in *2023 IEEE symposium on security and privacy (SP)*. IEEE, 2023, pp. 2658–2675.

[20] Joern. https://github.com/joernio/joern.

[21] Z. Guo, T. Tan, S. Liu, X. Liu, W. Lai, Y. Yang, Y. Li, L. Chen, W. Dong, and Y. Zhou, "Mitigating false positive static analysis warnings: Progress, challenges, and opportunities," *IEEE Transactions on Software Engineering*, 2023.

[22] Expression language injection. https://owasp.org/www-community/vulnerabilities/Expression_Language_Injection.

[23] Jsqlparser. https://github.com/JSQLParser/JSqlParser.

[24] Alibaba-druid. https://github.com/alibaba/druid.

[25] T. Tan and Y. Li, "Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. Association for Computing Machinery, 2023, p. 1093–1105.

[26] Java pathfinder. https://github.com/javapathfinder/jpf-core.

[27] J. Bell and G. Kaiser, "Phosphor: Illuminating dynamic data flow in commodity jvms," *ACM Sigplan Notices*, vol. 49, no. 10, pp. 83–101, 2014.

[28] M. Chen, T. Tu, H. Zhang, Q. Wen, and W. Wang, "Jasmine: A static analysis framework for spring core technologies," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[29] (2020, Feb.) Spring dominates the Java ecosystem with 60% using it for their main applications. [Online]. Available: https://snyk.io/blog/spring-dominates-the-java-ecosystem-with-60-using-it-for-their-main-applications/

[30] S. Cao, X. Sun, X. Wu, L. Bo, B. Li, R. Wu, W. Liu, B. He, Y. Ouyang, and J. Li, "Improving java deserialization gadget chain mining via overriding-guided object generation," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 397–409.

[31] I. Haken, "Automated discovery of deserialization gadget chains," *Proceedings of the Black Hat USA*, vol. 48, 2018.

[32] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda, and M. Egele, "Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing," *arXiv preprint arXiv:2002.03416*, 2020.

[33] J. Wang, Y. Wu, G. Zhou, Y. Yu, Z. Guo, and Y. Xiong, "Scaling static taint analysis to industrial soa applications: A case study at alibaba," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1477–1486.

[34] J. Dahse and T. Holz, "Simulation of built-in php features for precise static code analysis." in *NDSS*, vol. 14, 2014, pp. 23–26.

[35] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," in *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 334–349.

[36] J. Huang, Y. Li, J. Zhang, and R. Dai, "Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 581–592.

[37] P. Li and W. Meng, "Lchecker: Detecting loose comparison bugs in php," in *Proceedings of the Web Conference 2021*, 2021, pp. 2721–2732.

[38] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1125–1142.

[39] C. Luo, P. Li, and W. Meng, "Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2175–2188.

[40] P. Li, W. Meng, M. Zhang, C. Wang, and C. Luo, "Holistic concolic execution for dynamic web applications via symbolic interpreter analysis," in *Proceedings of the 45th IEEE Symposium on Security and Privacy (Oakland). San Francisco, CA, USA*, 2024.

[41] phpjoern. https://github.com/malteskoruppa/phpjoern.

[42] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting node. js prototype pollution vulnerabilities via object lookup analysis," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 268–279.

[43] M. Shcherbakov, M. Balliu, and C.-A. Staicu, "Silent spring: Prototype pollution leads to remote code execution in node. js," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5521–5538.

[44] J. F. Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner, "Symbolic execution for javascript," in *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, 2018, pp. 1–14.

[45] B. Loring, D. Mitchell, and J. Kinder, "Sound regular expression semantics for dynamic symbolic execution of javascript," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 425–438.

[46] S. Park, D. Kim, S. Jana, and S. Son, "{FUGIO}: Automatic exploit generation for {PHP} object injection vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 197–214.

[47] Y. Zhao, Y. Zhang, and M. Yang, "Remote code execution from {SSTI} in the sandbox: Automatically detecting and exploiting template escape bugs," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3691–3708.

[48] Z. Li and F. Xie, "In-situ concolic testing of javascript," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 236–247.

[49] M. H. M. Bhuiyan, A. S. Parthasarathy, N. Vasilakis, M. Pradel, and C.-A. Staicu, "Secbench. js: An executable security benchmark suite for server-side javascript," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1059–1070.

[50] B. Loring, D. Mitchell, and J. Kinder, "Expose: practical symbolic execution of standalone javascript," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, 2017, pp. 196–199.

[51] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner, "Javert 2.0: Compositional symbolic execution for javascript," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, 2019.

[52] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective {Inter-Component} communication mapping in android: An essential step towards holistic security analysis," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 543–558.

[53] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 280–291.

[54] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, pp. 1–32, 2018.

Appendies

A JAEX's modeling specifically for Java Web applications

Table 4 presents the typical examples of JAEX's modeling for mainstream frameworks and third-party libraries used in Java Web development. It mainly consists of two parts: static analysis entry points and APIs that can trigger cross-thread dataflows.

B Source Element in Concolic Variable

To accurately track concolic variables during concolic execution, JAEX uses the `Source` field to identify the specific origin of the variable. Specifically, we represent this field using a tuple:

```
(Source Type, Allocation)
```

Here, `Source Type` refers to the type of origin, which we currently classify into the following categories:

- `WEB_ENTRY`: This refers to the parameters of a Web Entry Method, such as the `addTaskDef` method at line 4 in Figure 2, which is a Web Entry Method annotated with `@POST` in Spring. The parameter `tDef` falls into this category.
- `INPUT_GLOBAL`: This refers to framework-level global variables related to user input. For example, the `HttpServletRequest` object used to store client HTTP request information in Servlet applications belongs to this category.
- `EXTERNAL`: This refers to those variables from external resources (e.g., database). As mentioned in §3.4, JAEX models the APIs of external resources and returns appropriate concolic variables based on semantics. These variables are marked with this type. Additionally, this is only a broad category label. In practice, JAEX further distinguishes different external resource types by marking them accordingly.
- `HYBRID`: This refers to variables generated during execution through concolic operations on any of the above three types of variables.

As for `Allocation`, different `Source Type`s have distinct methods for marking `Allocation`. We will now describe them in detail:

- `WEB_ENTRY's Allocation`: This identification is done using the `methodSignature/paramIndex.[field]`, where the field is optional.
- `INPUT_GLOBAL's Allocation`: This identification is done using `methodSignature/KeyName`, where the method signature refers to the method used to retrieve the parameter value, and `KeyName` refers to the parameter's key. For example, `getParameter('a')` would be recorded as `getParameter/a`.
- `EXTERNAL's Allocation`: This identification is done using `className.methodName#LineNumber`, which essentially records the invocation context of APIs.
- `HYBRID's Allocation`: This is directly represented by the set of sources from which the variable originates, `[SOURCE1,...]`

Table 4: Overall Summary of JAEX's Modeling of different frameworks and libraries

Framework/Lib	Category	Entry Points	Cross-thread Dataflow APIs		
			Asyn. Communications	Database Operation	File Operation
Spring	Web Framework	@Controller, etc. XML Configuration	@Async, etc. Spring Event	Spring Data JPA	FileCopyUtils APIs FileSystemUtils APIs
Struts2	Web Framework	XWorks2 APIs XML Configuraion	-	-	-
Servlet	J2EE APIs	@WebServlet, etc. Methods(e.g., doGet) HttpServletRequest APIs	@WebServlet, etc.	-	-
JAX RS/WS	J2EE APIs	@POST, etc.	AsyncHandler, etc.	-	-
MyBatis	ORM Framework	-	-	SqlSession APIs BaseMapper APIs XML Configuration	-
Hibernate	ORM Framework	-	-	Session APIs	-
JDBC	J2SE APIs	-	-	Statement APIs	-
JDK Native	Libs	Methods(e.g., Thread.run)	-	-	FileInputStream APIs, etc.
Third Party	Libs	-	RxJava Message Queue Libs	Hutool-db Apache commons-dbutils	Hutool-core Google Guava I/O Apache commons-fileupload