计 算 机 研 究 与 发 展
Journal of Computer Research and Development

https://crad.ict.ac.cn
63(1): 214−226, 2026

基于前后端联合分析的 Java Web 漏洞挖掘方法

邹福泰[1] 姜开达[2] 占天越[1] 施 纬[1] 张 亮[3]

[1](上海交通大学计算机学院 上海 200240)

[2](上海交通大学网络信息中心 上海 200240)

[3](国家电网公司华东分部 上海 200120)

(zoufutai@sjtu.edu.cn)

Java Web Vulnerability Discovery Method Based on Co-Analysis of Front-End and Back-End

Zou Futai[1], Jiang Kaida[2], Zhan Tianyue[1], Shi Wei[1], and Zhang Liang[3]

[1](*School of Computer Science*, *Shanghai Jiao Tong University*, *Shanghai* 200240)

[2](*Network and Information Center*, *Shanghai Jiao Tong University*, *Shanghai* 200240)

[3](*East China Branch of State Grid Company*, *Shanghai* 200120)

Abstract Accurately and efficiently identifying security vulnerabilities in Web applications holds significant research value, especially as Web systems grow in complexity and scale. Most existing studies in the field of Web vulnerability discovery have focused on PHP-based applications, rendering them less effective or even inapplicable when transferred to the domain of Java Web applications. Furthermore, traditional vulnerability discovery methods often struggle to meet the demands of large-scale and high-efficiency scenarios. Specifically, these methods face difficulties in achieving the precision of dynamic analysis while maintaining the performance benefits of static analysis. To address these challenges, we propose a novel Web vulnerability discovery approach based on joint frontend-backend analysis. By parsing the frontend code to extract taint source information, the method guides backend analysis for pruning irrelevant paths, thereby enhancing both the vulnerability coverage and discovery efficiency. Additionally, the approach integrates both static and dynamic features of the program to construct a comprehensive code model. It combines techniques such as data flow analysis, taint analysis, symbolic execution, and lightweight dynamic solving to detect and verify potential vulnerabilities. This integrated strategy leads to a significant performance boost with minimal computational overhead. The proposed method is evaluated on 105 Java Web vulnerabilities sourced from CVE (common vulnerabilities and exposure) entries, open-source CMS (content management system) platforms, and community-developed applications. The experimental results demonstrate that each component of the system performs effectively, and the overall framework exhibits strong capability in discovering real-world Java Web vulnerabilities.

Key words Web security; Java; vulnerability discovery; tainted analysis; symbolic execution

摘 要 精准高效地挖掘 Web 应用当中存在的安全漏洞具有极高的研究价值。Web 漏洞挖掘相关研究大多是针对 PHP 应用的,无法直接应用于 Java Web 漏洞挖掘。且现有的 Web 漏洞挖掘方法难以适应批量高效的需求,即难以在保持静态代码分析的性能下取得动态分析的精确度。为解决上述问题,提出了一种前后端联合分析的 Web 漏洞挖掘方法,利用前端解析提取污点源信息来帮助后端分析进行剪枝,提高漏洞覆盖率和检测性能;同时在漏洞挖掘时利用程序的动静态信息进行代码建模,结合数据流分析、污点

收稿日期: 2024-11-10; **修回日期**: 2025-07-15

基金项目: 国家重点研发计划项目(2020YFB1807500)

This work was supported by the National Key Research and Development Program of China(2020YFB1807500).

分析、符号执行以及轻量动态求解技术完成漏洞的挖掘和验证，在引入较少开销前提下带来较大的效果提升。选取了 CVE（common vulnerabilities and exposure）漏洞、开源 CMS（content management system）以及开源社区应用中共 105 个 Java Web 漏洞对所提出的方法进行了实验，证明了各模块具有较好的分析效果，整体具有较强的漏洞挖掘能力。

关键词 Web 安全；Java；漏洞挖掘；污点分析；符号执行

中图法分类号 TP311.13；TP393

DOI: 10.7544/issn1000-1239.202440871 **CSTR:** 32373.14.issn1000-1239.202440871

面对不断增长的 Web 服务体量和日益复杂严峻的安全形势，传统的 Web 漏洞挖掘技术的性能、精度以及覆盖范围愈来愈难以适应新的安全需求[1]。Web 相关技术飞速发展，出现了各种适应于互联网体系的开发架构和系统，Java 因为其面向对象、包管理体系完善、低耦合、拓展性强等特点成为了主流的后端语言。Java 体系下具有很多公共开源的组件依赖，这一方面降低了开发的门槛和难度，但同时也放大了安全风险，例如 Log4j 组件漏洞就造成了相当大的影响，美国网络安全审查委员会的报告[2]指出，该漏洞是史上最严重的软件安全漏洞之一。

然而，目前学术界针对 Java Web 漏洞的研究比较少，大部分 Web 漏洞挖掘研究都是聚焦于 PHP 应用[3-5]。在过去十几年当中，安全研究者们针对 PHP 体系下的 Web 漏洞提出了各种各样的检测思路，Dahse 等人[6]提出 PHP 静态分析方法及其开发的著名检测工具 RIPS[7]。然而，Java Web 漏洞的原理、表现和检测方式和传统的 PHP 漏洞具有很大的不同，这些针对 PHP 的研究成果很难直接用到 Java Web 漏洞挖掘上。

Java 是强类型、编译型语言，Java Web 系统是前后端分离的，一般前端使用 NodeJS 框架[8]，后端基于 Spring 框架[9]实现 Restful 接口供前端交互，在这种架构下，后端的具体实现对攻击者而言是完全不可见的（本文讨论的漏洞都是这种 Java 应用架构下，不考虑 JSP（Java server pages）这种类似 PHP 的方式）。所以，Java Web 漏洞的表现形式也与 PHP 不同，例如 XSS 漏洞；文件上传、反序列化、SQL 注入等其他常见漏洞的成因和表现也和 PHP 不同，检测方式也大不相同。

以 Web 应用中最为常见且危害极大的 SQL 注入攻击为例，要实现对 SQL 注入漏洞进行代码级别的检测，必须明确代码中是如何通过 SQL 语句访问数据库的。在 Java 应用中，一种广泛使用的实现方式是借助 ORM（object-relational mapping）框架对数据库中各个数据表进行建模，映射到 Java 对象，通过对象的访问来实现数据库操作，SQL 语句拼接的过程都交由框架完成。使用 ORM 框架时，开发者通常会将查询的逻辑按照特定的规则写在 Java 代码或 XML 文件中，不同的框架可能会使用不同的方法。因此对于 ORM 实现的数据库访问，我们很难从代码中直接看到用户输入和最终的 SQL 语句之间的关系，也就很难直接分析拼接过程带来的 SQL 注入漏洞。

我们总结了 Java Web 漏洞检测目前存在的主要难点：

1）Java Web 应用的实现具有复杂性和多样性[10]。前端可能会使用不同的 NodeJS 框架和 UI（user interface）组件，后端 Java 代码实现会使用不同的依赖管理，同一 Web 框架下不同版本之间也可能存在较大差异。

2）Java 代码的动态特征包括多态、反射、动态代理和 Spring 框架的动态对象加载，以及各组件的动态代码生成。为捕捉完整的代码逻辑，需处理这些动态特征。

为了解决上述问题，本文针对 Java Web 应用设计实现了一种采用前后端联合分析的漏洞挖掘方法，利用前端解析提取污点源信息帮助后端分析进行剪枝，在漏洞挖掘时利用程序的动静态信息进行代码建模，结合数据流分析、污点分析、符号执行以及轻量动态求解技术完成漏洞的挖掘和验证。本文实现的方法对常见的反序列化漏洞、SQL 注入漏洞以及文件上传漏洞进行检测，对于这 3 种漏洞识别的精确率达到 79%，漏报率低至 12%，和现有的 Java Web 漏洞自动化检测工具相比，具有较好的效果和性能。本文的贡献有 3 点：

1）提出一种基于前后端联合分析的 Java Web 漏洞自动化挖掘方法，通过前端特定的参数名称能够帮助定位后端的相关代码，实现剪枝的效果，大幅提高分析效率。

2）提出了一种轻量的动静态结合的后端代码分析方法，既能利用静态分析实现批量性能，又能通过

轻量的动态过程捕捉代码中的动态信息,在取得较好的检测效果下兼顾较高的性能和实用性。

3)选取了 CVE 漏洞、开源 CMS 以及开源社区应用中共 105 个 Java Web 漏洞对本文提出的方法进行了实验,漏洞识别正确率达到 79%,漏报率低至 12%,并发现了 14 个新漏洞。

1 相关工作

1.1 自动化 Web 漏洞挖掘

自动化 Web 漏洞挖掘技术可以分为黑盒测试[11-12]和白盒测试[13-14],灰盒检测技术一般不应用于 Web 安全。黑盒测试关注外部功能和用户视角,不考虑内部实现,多以动态测试为主[15];而白盒测试则深入内部代码、结构进行测试,兼顾静态与动态测试技术[16-17]。

黑盒检测技术通过捕获动态信息,评估系统在不同输入条件下的行为表现,从而发现潜在的漏洞。Pellegrino 等人[18]针对传统黑盒检测技术面对动态 Web 应用的不足,对客户端的 JS 代码进行动态分析,hook 利用事件处理和网络请求的 API(application programming interface),全面收集动态发出的 URL(uniform resource locator)请求和用户表单提交。Link 利用强化学习动态地调整修改攻击载荷,以检测目标系统中的反射型 XSS 漏洞,其状态、动作和奖励函数被设计成以黑盒和完全自动的方式发现反射型 XSS 漏洞[19]。Scanner++[20]通过将现有扫描器的能力与基于包的攻击意图同步相结合,生成更多样化的攻击向量,实现更好的漏洞检测性能。

白盒测试大都基于应用源代码进行检测,可以根据使用的分析技术选择静态或动态分析或二者结合的方式。Huang 等人[16]针对 PHP 应用的文件上传漏洞设计了一种自动化静态源代码检测方法,通过基于 AST(abstract syntax tree)的符号执行算法,引入漏洞代码段定位以减小执行开销,根据漏洞规则得到约束表达式并根据求解结果判断漏洞。之后,在 UChecker 基础上进行改进,提出了 UFuzzer[21],该方法引入了基于 Fuzzing 技术的动态求解方法,检测效果进一步提升。刘行波等人[22]提出一种细粒度的污点分析方法,在代码分析过程中生成更加精确的对象状态记录;对代码的执行路径进行判断并精确地记录污染情况,有效减少了过污染和欠污染。孙基男等人[23]采用符号值作为输入来模拟程序执行,提取执行路径上相应的约束条件,并构成可满足矩阵以及不可满足矩阵 2 个注入类漏洞。安全分析与检测模型求解结果可映射为注入类安全漏洞的安全状态。Khodayari 等人[24]的研究综合分析了客户端 CSRF(cross-site request forgery)漏洞并提出了检测方案 JAW,设计了一种混合代码属性图用于展现客户端 JS 代码的静态特征和程序与 DOM(document object model)交互的动态流程。使用符号执行工具对漏洞位置的可达性进行了验证 JAW,基于混合代码属性图遍历就能实现客户端 CSRF 漏洞的挖掘,在多个常用 Web 应用上实验效果良好。

但是,许多漏洞是前后端同时作用的结果,在现有的检测技术中对 Web 应用前后端的处理往往是割裂的,现有的黑盒检测方案往往只能看到前端,白盒检测则主要以后端代码为主,合理地利用前端功能点信息指导后端的代码分析是非常有用的。

1.2 Java Web 漏洞挖掘

Java Web 应用程序往往过于复杂,其消耗过多的系统资源,并且难以配置。因此,以往研究中 Java Web 漏洞检测方法往往依赖于静态代码审计和一般的黑盒 Web 漏洞检测方法(如 SQL 注入、XSS)。Livshits 等人[25]提出了一种基于精确指针分析的静态分析方法,将用户提供的漏洞规范自动转化为静态分析器,在静态分析的代码中找到与规范匹配的所有漏洞。JSEFuzz[26]结合了模糊测试和符号执行的方法,利用模糊测试方法查找漏洞触发条件和相应的输入数据,利用符号执行转换输入数据,在程序级验证漏洞。这些方法忽略了前端代码所能提供的有效信息,并且没有完全解决 Java Web 特性给漏洞检测带来的难题。

2 方法概述

本文主要研究 Java Web 应用的漏洞挖掘,为了能够实现较好的检测效果和较高的批量检测性能,创新地提出了一种前后端结合的分析方法,以静态源代码分析为主和轻量动态分析为辅,以实现高效的检测效果。本文提出的检测方法总体架构如图 1 所示,分为以下 4 个模块:

1)代码预处理。对输入的代码文件进行预处理,主要进行前后端代码分类和元信息识别。

2)前端分析。实现对前端框架与后端通信的 API 识别以及前端页面功能点的识别。

3)后端静态分析。是整个检测流程的核心部分,对应用后端 Java 代码进行建模和分析,得到可能的漏洞利用路径。

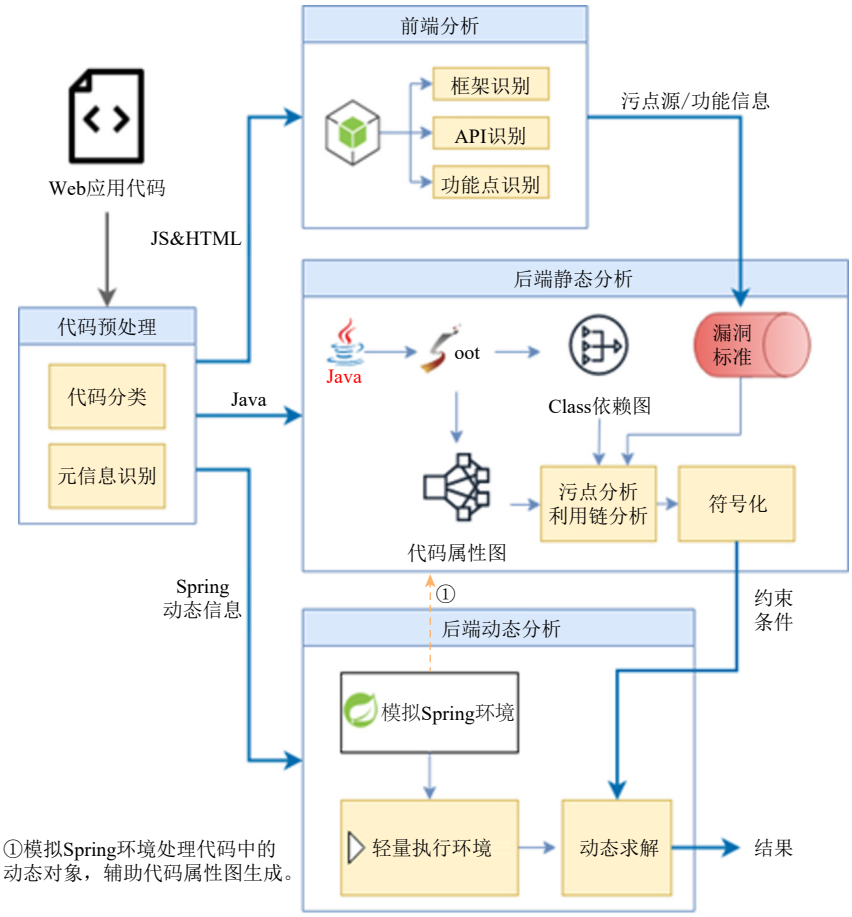


Fig. 1 Architecture diagram for Web vulnerability discovery method via joint front-end and back-end analysis

图 1 基于前后段联合分析的 Web 漏洞挖掘方法架构图

4)后端动态分析。模拟 Spring 应用的上下文来辅助静态建模，同时构建轻量的执行环境进行动态约束求解，判断漏洞是否可用。

总体来说，本文方法接收 Java Web 源代码作为输入，其中后端 Java 代码也可以是按 Spring 应用标准打包的 jar/war 包形式。该方法首先进行代码预处理，识别前后端代码和框架种类等元信息，然后同步进行前端代码分析和动态分析中的 Spring 上下文模拟(如果应用使用了 Spring 框架的话)。之后进入后端静态分析，首先结合部分动态分析的结果建立模型，再结合前端分析结果进行漏洞挖掘，输出约束条件，最后通过动态分析模块的轻量执行环境进行动态求解。

3 方法实现

3.1 代码预处理

本文漏洞挖掘的输入包括 Java Web 应用的前端代码和后端代码，以及依赖组件信息或代码 jar 包。考虑到 Java Web 开发中的各种框架、组件和版本可能导致代码规范差异和多样性，我们引入了预处理模块以统一处理这些差异，提高了设计的扩展性和可维护性。

代码预处理模块进行代码分类和应用元信息识别。代码分类主要对前后端代码进行分类，而应用元信息识别具体包括：代码目录结构、配置文件(如 Maven 和 Gradle 的配置文件)、组件依赖路径或者清单、Web 框架及组件版本、代码格式(如 Java, XML, Groovy)等信息。

代码分类的核心问题在于如何快速地定位前后端代码。针对后端代码，我们采用深度优先搜索算法进行目录遍历，寻找定位到 Java 或 class 文件并读取其中的完整包名，进而判断其路径是否符合包名规范。遍历过程还涉及组件依赖包名过滤，以防止包名与路径不匹配(通过代码文件包名检测过滤)或者避免不必要的遍历(通过对遍历路径名称进行过滤)。对于前端代码，我们利用广度优先搜索与深度优先搜索相结合的策略，特别针对基于 NodeJS 的实现，优

先搜索 NodeJS 的依赖描述文件——package.json。该文件通常位于前端目录的顶层,大多数情况下,只需进行 2 层遍历即可成功定位。若无法定位到 package.json,算法将转入深度优先搜索,寻找前端代码相关文件,主要匹配 html, css, js, ts, vue 文件。

应用元信息识别首先针对后端代码的目录结构进行解析,确定所用的包管理体系;然后读取并解析依赖信息中的框架与组件信息;最后分析 Web 框架和 ORM 组件,并检查代码文件格式,输出后续漏洞挖掘所需要的元信息。

3.2 前端分析

本文提出前端分析流程的目的是全面利用 Web 应用的信息提升漏洞检测的效果和性能。因此,前端分析需要快速准确地挖掘前端代码中的各种功能点以及它们是如何与后端交互的,从而指导后端代码分析,提高效果和性能。相比传统的纯后端检测方法,前端检测需要分析更多的代码,但如果在前端分析上多消耗的时间小于在后端分析中所节省的时间,即可认为有性能提升。同时,前端分析的另一挑战在于处理多样的前端框架(如 Vue, React, Angular)、UI 组件和通信方式,尤其是 React 结构的复杂性。

前端分析的目标在于定位污点源和功能点识别。前端分析模块接收预处理的前端代码作为输入,首先通过依赖管理文件 package.json 识别框架类型,然后利用框架特定方法解析路由文件以识别页面和组件。接着,针对性地解析组件,通过元素信息推断功能、定位与后端通信的代码部分,并识别后端 API 信息。最后,将与用户输入以及后端通信相关的部分作为污点源输出给后端分析模块。前端分析的关键点是路由识别、组件定位以及 API 识别。

3.2.1 路由识别

在识别框架后,我们针对 Vue, React, Angular 各自进行解析。通过文件搜索和索引建立,我们定位到路由文件,并优先关注含有“router”字段的路径,同时过滤掉像“assets”这样的资源文件夹以提高搜索性能。如果无法通过路径名称定位,我们使用模式匹配和已建立的索引进行代码匹配。路由代码定位后,解析其内容以获取 URL 路径对应的组件列表。考虑到 React 应用的灵活性和解析成本,我们选择直接根据代码文件索引进行分析,虽可能损失一些信息,但可保持良好的性能且其对后续分析的影响可接受。

3.2.2 组件定位

路由信息解析后,我们通过组件路径、组件名称以及对应的 import 语句定位代码位置,以建立 URL 路径和代码文件的对应关系。对于 React 应用,我们直接使用先前建立的文件索引查找组件。对组件代码提取特征信息以进行功能点识别,包括:URL 路径(React 应用除外)、组件命名或关键字信息(与 URL 路径高度相关)以及组件对应的前端页面元素,即组件的 HTML 内容。

3.2.3 组件功能识别

我们按照常见网站功能及 Java Web 漏洞涉及的功能,将组件功能划分为登录、注册、搜索、文件和数据获取 5 类,并设计了一套启发式算法对组件功能进行分类。算法根据前文提及的 URL 路径、组件命名或关键字信息以及组件对应的前端页面元素 3 个因素计算各类别可能性分数,取最大分数超过阈值的类别。对于 URL 路径和组件名称,使用关键字匹配计算分数,例如 login, search 等意义明确的关键字,对于每个类别收集常用的关键字列表,并根据相关程度给予不同的分数。对于页面内容,我们识别页面元素种类,如输入框、按钮等,通过元素存在情况推断页面类型。同时,我们支持 AntdDesign 和 Element-UI 的识别,通过匹配它们在不同框架上的标签关键字来识别页面元素作用,并根据元素组合进行类型打分。

3.2.4 API 识别

除组件功能点识别外,前后端通信识别也很关键,也就是在前端代码中识别出与后端通信的 API。图 2 是 Vue 使用 axios 和后端 springboot 应用通信的一个例子,其中黄色部分是通信所用的 API 信息。我们通过分析 Vue 代码,提取 API 的 URL、请求方法和数据格式等信息,这些构成前后端交互的入口或攻击者可控的污点源。首先通过 package.json 判断应用使用的方式,然后在组件代码中寻找对应 API 调用点,提取文件的 API 调用,我们采取先定位接口,再在组件中寻找调用情况的方式,并仅考虑一级文件调用。在完成组件功能识别和 API 调用识别后,我们可以将组件功能和对应的 API 关联起来,如此可通过简单的前端分析确定后端代码的入口,提升漏洞挖掘效率。此外,我们还输出组件功能的特征数据,包括组件名称、功能、输入框属性等关键信息,对后续代码分析有用。

3.3 后端静态分析

后端静态分析为整个检测流程的核心部分,分为 3 个阶段:代码建模、路径挖掘、符号执行。首先基于中间语言对 Java 代码进行建模,生成代码属性

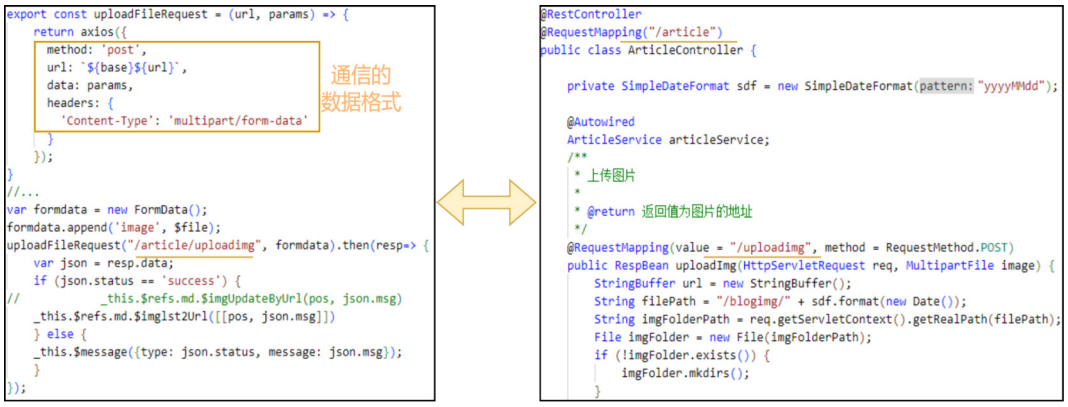


Fig. 2 Demonstration of Vue-backend communication using axios

图 2 Vue 使用 axios 与后端通信示例

图和 Java 类依赖图，并在该过程中利用前端分析的结果快速定位和剪枝；然后在代码属性图的基础上利用污点分析挖掘 SQL 注入和文件上传漏洞的利用路径，在类依赖图上通过利用链分析挖掘反序列化漏洞利用路径；最终通过符号执行生成各个路径的可达条件用于后续的求解。

3.3.1 代码建模

代码建模需要维护 Java 代码的组件依赖完整性以避免类缺失错误，解析非 Java 格式（如 XML, Groovy）的代码来捕获完整的程序逻辑以及处理 Java Web 的动态特性，如多态、反射、动态代理和动态代码生成等。所以，我们提出了一种前后端结合、动静态结合、代码属性图 *CPG*（code property graph）和类依赖图 *CDG*（class dependency graph）相结合的代码建模方式，能有效地对建模过程进行剪枝，同时在建模过程中尽可能体现应用的动态特征。为了提高分析的效率，我们选用基于三地址码中间语言 Jimple[27] 进行代码建模，将 Java 代码转为 Jimple 中间语言后，在其基础上生成代码属性图和类依赖图。代码属性图包含控制流图和函数调用图，控制流图是对代码执行的路径以及数据之间的传递关系进行呈现，函数调用图则是对代码中的显式和隐式的函数调用关系进行呈现。

然后，针对 Java 面向对象的特性以及反序列化利用链挖掘的需要构建 Java 类依赖图 *CDG*。该图包含各个类和接口之间的继承关系、维护可以反序列类的成员信息和函数调用信息、各个类方法签名的前缀索引、动态代理 Handler 的追踪信息 4 个方面。反序列化漏洞可能利用多个代码段，为缩小解析范围，根据前端分析，重点分析涉及用户输入的可反序列化的类，其他类型构建函数签名索引，待利用链需要时再深入分析，省去不必要解析。

3.3.2 路径挖掘

路径挖掘会根据代码建模的结果寻找可能存在的漏洞路径，即根据各种 Web 漏洞的特征去匹配对应的功能路径或者可用的利用链。对于正常控制流中的漏洞（文件上传和 SQL 注入），在代码属性图上进行数据流分析和污点分析来定位路径；对于劫持控制流的漏洞（反序列化），则基于类依赖图利用路径进行挖掘。

我们设计了基于前向数据流分析的污点追踪算法进行文件上传和 SQL 注入漏洞路径的检测。如算法 1 所示，该算法的目的是定位从污点源到脆弱 *sink* 函数的污点传播路径，定义 *CPG* 中每个节点的集合 *IN*, *OUT*, 其中集合 *IN* 表示污点来自变量，集合 *OUT* 表示污点能传播到的变量。根据漏洞特征，如果某个节点未对集合 *IN* 中污点变量进行重新赋值，则该节点集合 *OUT* 包含集合 *IN*；如果某个节点中非污点变量和污点变量相关（通过运算或函数调用关系相关），则将该变量加入集合 *OUT*；对于分支节点，记录各个分支的判定条件，并附加到集合 *OUT* 的变量上。

算法 1. 基于数据流分析的污点分析算法。

输入：代码属性图 *CPG*，污点源；

输出：可能的漏洞路径 *P*。

① 初始化集合 *IN*, *OUT*；

② 开始访问 *CPG*，污点变量加入 *IN*；

③ while 耗时小于阈值且未处理完入口方法 do

④ 处理当前节点 *node*, *OUT* = *IN*；

⑤ if *node* 为 *CPG/CDG* 交界 then 进入对方节点；

⑥ end if

⑦ for *v* in *IN* do

⑧ if *node* 为 *sink* 函数且参数与 *v* 相关 then

更新 *P*；

⑨ end if
⑩ if *v* 被赋值 then *OUT* – *v*;
⑪ else if *node* 中变量 *x* 与 *v* 相关 then *OUT* +*x*;
⑫ end if
⑬ if *node* 为分支且分支条件与 *v* 相关 then
为不同分支的 *v* 建立拷贝;
⑭ end if
⑮ end for
⑯ end while

针对反序列化利用链的挖掘, 我们提出了一种基于类依赖图的双向搜索算法, 如算法 2 所示。限于文章篇幅, 该算法呈现了前向搜索过程, 后向搜索的过程与之相似, 当前后路径相交时则说明找到了可能的利用链。反序列化漏洞的利用路径多样, 仅前向搜索从用户输入出发的效率低, 因此需从危险函数或现有利用链出发进行后向搜索。双向搜索相遇点表明可能的利用路径, 显著提升搜索效率。我们汇集了公开的 Apache commons-collections, JDK, Spring, Groovy, XStream, FastJson 等利用链及危险函数作为后向的 *sink* 点, 前向的污点源通过算法 1 定位用户输入至反序列化流程。以双向 BFS 算法在类依赖图中寻找可用利用链, 搜索过程实施剪枝, 忽略数据不匹配或函数参数不符路径。

算法 2. 双向利用链搜索算法的前向搜索部分。
输入: Java 类依赖图 *CDG*, 反序列化入口类 *K*;
输出: 可能的利用路径 *P*。
① 初始化上一跳结尾函数签名 *f*;
② while 耗时小于阈值 do
③ 处理 *CDG* 中当前类 *cls*;
④ if *cls* 不符合函数调用图 *f* 的 class 类型 then
略过;
⑤ end if
⑥ if *cls* 为接口且存在代理 *Handler* then
将 *Handler* 及 *invoke* 函数加入 *BFS* 队列;
⑦ else if *cls* 为接口且无代理类 then 略过;
⑧ else
⑨ if *cls* 为可反序列类 then 直接构造该类实例;
⑩ else if *cls* 不可反序列化 then
通过反射构造, 加入 *f*;
⑪ for *f* 中的每次函数调用 *x* do
⑫ if *x* 不能构造 then 略;
⑬ 查找 *x* 中调用的函数 *x.func* 的函数签名索引, 找到类 *newcls*;
⑭ end if
⑮ if *newcls* 存在符合 *x.func* 的签名函数 then
⑯ 将 *newcls* 及函数 *x.func* 加入 *BFS* 队列;
⑰ end if
⑱ end for
⑲ end if
⑳ end if
㉑ end while

3.3.3 符号执行

符号执行旨在制定漏洞路径的约束条件。在符号执行中, 需以符号变量模拟执行过程, 理论上完全符号化所有变量的方案效果最佳。然而, 实际程序中变量众多, 部分类型如多维数组难以完全符号化, 故需权衡取舍。目前选择符号化数字和字符串类型变量, 在保证检测效果的前提下获得良好性能。针对路径爆炸问题, 仅需计算达到函数 *sink* 路径的约束, 并减少路径分析量且限制循环情况, 避免过多迭代。

具体来说, 符号执行阶段会根据之前挖掘的漏洞路径在 *CPG* 上进行对应路径的符号执行, 从而得到抵达函数 *sink* 的约束条件。漏洞路径当中会包括各个分支的条件, 根据这些信息可以快速地定位需要执行的路径。对于 SQL 注入和文件上传漏洞, 我们关注从用户输入到危险函数的整条路径; 对于反序列化漏洞, 我们则是关注每个 gadget 类当中执行到与下一个 gadget 相接的函数的路径。在输出约束条件时, 对于字符串相关操作会进行处理, 将它们转为逻辑表达式的形式, 对于未建模的函数(例如某个组件中的 API 调用出现在分支约束中)则不进行处理, 交给后续的动态分析模块求解。

3.4 后端动态分析

后端动态分析模块是实现轻量动态检测的关键。本模块可以辅助后端静态分析模块进行更为精确的代码建模, 对漏洞路径约束条件也可以进行更精确的动态求解, 提高检测的精度。传统动态分析效率低的主要原因在于需要人工运行目标应用。Java 程序中, 不同应用使用不同环境、依赖, Web 应用需与前端、数据库、缓存等交互。如何实现一种不需要考虑运行环境且能够快速地进行轻量动态执行是本模块需要解决的问题。其次, 后端静态分析得到漏洞路径后, 通过符号执行获取路径约束, 由于约束中可能会包含动态逻辑, 使用静态求解器的求解精度较差, 所以在动态分析阶段必须对其进行动态求解,

如何将这些约束转化为适合动态执行的方式是本模块需要解决的另一个问题。

整个后端动态分析模块整体分为 Spring 上下文模拟和轻量执行环境 2 个部分。Spring 模拟部分负责处理程序中的 XML 代码以及 Spring 框架下的动态生成对象；轻量执行环境则负责将约束条件转为轻量的 Java 代码形式并进行执行求解。

3.4.1 Spring 上下文模拟

Spring 框架的特点之一是 IOC（控制反转）概念，无需构建 JavaBean 对象，由 Spring 集中管理。传统静态分析难解析动态 Bean 对象，需模拟 Spring 上下文加载，才能精准分析基于 Spring 的应用。经过对 Spring 加载 Bean 过程源代码的解析，我们复用其读取 XML 文件以及使用反射解析注解的方法，实现较为简单的 Bean 加载模拟。由于目前无法完全还原执行环境，因此在 Bean 对象的初始化过程中会大概率出现依赖缺失的错误，对此进行忽略处理。在加载完成 Bean 对象之后，针对静态代码建模的问题，使用 Spring 上下文的接口和反射机制获取所有 Bean 的信息，使得代码建模时可以利用 Spring 动态特征。

3.4.2 轻量执行环境

将静态约束转为动态执行，首先以属性图缩减方法实现约束转化为 Java 代码的形式。定位约束对应的 *CPG* 路径，去除路径上与约束中无关的符号变量状态节点，构建新的局部 *CPG*。由此得出 Jimple 代码段，将其转化为可执行字节码。实现轻量的关键在于转换后代码片段仅与约束中的变量状态相关，需确保这些代码逻辑正常执行，无需保证完整执行环境。通过调用依赖和类依赖图保证局部代码的依赖存在。

将静态约束转为可执行字节码后，我们设计了一种动态约束求解算法验证其可解性，如算法 3 所示。该算法基于模糊测试思想，通过不断变异涉及的符号变量来判定约束是否有解。与模糊测试不同，用户输入的污点变量值源自漏洞特定的 payload，如文件漏洞的文件名和 SQL 注入的查询关键词。这些 payload 由历史漏洞特征和检测工具字典制定。对其他符号变量变异，我们采用“随机变异+特殊变异”策略，优先使用已有常量作为变异值。综合而言，该动态求解方法结合了动态特征的利用和优良的执行性能。

算法 3. 动态约束求解算法。

输入：约束代码；

输出：是否可达。

①导入 class 依赖；

② Spring 模拟加载 Bean；

③ for *payload* in 漏洞字典 do

④ 污点变量等于 *payload*；

⑤ while 耗时小于阈值 do

⑥ 变异其他符号变量；

⑦ if 约束代码能执行到底 then

⑧ 该 *payload* 可达；

⑨ break；

⑩ end if

⑪ end while

⑫ end for

4 实 验

为了评估本文方法，我们设计了相应的实验来回答下面 3 个问题。

RQ1：本文方法对 Java Web 漏洞的检测效果如何。

RQ2：前后端联合分析和动静态结合分析对漏洞检测是否有效。

RQ3：与现有方法相比，本文方法是否有优势。

4.1 实验设置

4.1.1 实验环境及数据集

实验所使用的 Java JDK 版本为 build 1.8.0_301-b09。漏洞检测实验的样本来自 Java Web 组件历史版本的公开漏洞（CVE）、开源的 Java CMS 框架以及 Github 等开源社区上受欢迎的 Java Web 项目。收集了不同 Java Web 组件（包括 Log4j, fastJson, Xstream, Mybatis 等）的 61 个 CVE 样本、11 个常用的国内外 CMS 的开源版本以及开源社区中 15 个 Star 较多的 Java Web 应用作为实验的测试样本。代码预处理以及前端分析的测试样本则主要来自 Github, Gitee 等开源社区中的大中型 Java Web 项目，它们包含各种 CMS 系统和网站应用。

4.1.2 评价指标

我们将实验分为分类问题和重点识别问题 2 类，分类问题会关注各个不同类型识别的效果，如代码分类，在实验中更看重准确率（accuracy），即所有正确分类样本占全部样本数量的比例。漏洞识别问题则关注漏洞检测的精确率（$Precision = TP / (TP + FP)$）、漏报率（$Missing\text{-}Rate = FN / (TP + FN)$）、误报率（$False\ Positive\ Rate = FP / (FP + TN)$）。另外我们还会对运行的性能进行评估，主要指标为各个阶段分析所消耗的时间。

4.2 检测效果实验（RQ1）

我们选用了 CVE 漏洞、CMS 以及社区开源项目

等 87 个检测样本,存在 105 个漏洞,均为文件上传、SQL 注入或反序列化漏洞,其中包含 14 个漏洞是由本文方法在实验中新发现的。漏洞检测结果如表 1 所示。

Table 1 Overall Results of Vulnerability Detection Experiments

表 1 漏洞检测实验总体结果

样本来源	漏洞类型	已知数量①	识别数量②	精确率	漏报率
CVE	文件上传	15	14/17	0.82	0.07
	SQL 注入	20	16/22	0.73	0.20
	反序列化	26	23/24	0.96	0.12
CMS	文件上传	7	7/10	0.70	0.00
	SQL 注入	9	8/13	0.63	0.11
	反序列化	7	7/9	0.78	0.00
开源社区	文件上传	6(3)	5/7	0.71	0.33
	SQL 注入	9(6)	9/13	0.69	0.00
	反序列化	6(5)	6/6	1.00	0.00
总计	文件上传	28(3)	26/34	0.76	0.08
	SQL 注入	38(6)	33/48	0.69	0.16
	反序列化	39(5)	36/39	0.92	0.09
	ALL	105(14)	95/121	0.79	0.12

注:①括号内表示实验中新漏洞数量,②“/”前后表示真实漏洞总数和识别到的漏洞总数。

1)检测效果分析。从总体上看,所有漏洞的识别精确率达到了 0.79,漏报率则只有 0.12,这个表现相对来说还是比较好的。目前实现的方法在反序列化漏洞挖掘上的表现最好,在测试集中精确率达到了 0.92,漏报率低至 0.09,这说明本文设计的 Java 类依赖图建模的方法能够有效呈现代码中的 gadget 特征,较为全面地覆盖各种反序列化利用链的挖掘,较低的漏报率则说明了漏洞利用链挖掘算法以及动态约束求解的有效性。对于文件上传和 SQL 注入漏洞,本文方法分别达到了 0.76 和 0.69 的精确率。经过对识别结果的分析,精确率的下降主要有 2 方面原因:一是针对这 2 种漏洞的规则收集还不够细致和完善,导致在识别时会产生误报;二是部分复杂约束求解错误导致 SQL 注入漏洞的利用路径和方法相比文件上传漏洞更为复杂。

我们也对检测方法在不同类型的样本上的表现情况进行了比较。对于已知漏洞,检测效果在 CVE 样本上的表现较好,在 CMS 样本上的表现最差。我们认为这是由于已有的 CVE 样本所涉及的漏洞一般更符合公认的漏洞模型,且需要检测的代码量也更少。另外,在实验中,针对开源社区的检测样本,本文方法还识别出了 19 个新的 Web 漏洞,其中 14 个漏洞通过人工验证,包括 3 个文件上传漏洞、6 个 SQL 注入漏洞和 5 个反序列化漏洞,这一点证明了本文方法发掘新漏洞的能力。

2)执行性能分析。我们对本文方法的各个阶段的耗时进行了记录,并根据待分析的代码文件总大小进行了统计,结果如图 3 所示。图中记录了实验过程中,预处理、前端分析、后端建模、漏洞挖掘以及动态约束求解这 5 个阶段的执行耗时随输入的 Web 应用大小的变化趋势。应用大小在 8 MB 以下时,各个阶段的耗时都相对较短,漏洞挖掘阶段的耗时最长;应用大小超过 9 MB 后,后端建模过程的耗时显著增长,这是因为随着应用大小增长,前端识别到的污点源和功能点越来越多,因此建模阶段需要处理的代码也会越来越多,进而导致建模的耗时呈现指数级增长趋势。而其他阶段的执行复杂度是低于建模过程的,因此均呈现接近线性增长的趋势。在应用大小超过 50 MB 乃至更大规模时,总体耗时逐渐趋于平缓。总体耗时为预处理、前端分析、后端建模、漏洞挖掘以及动态约束求解的耗时之和,其中由于后端建模涉及复杂的计算开销,耗时占比最大,是总耗时的主要影响因素。实际中,预处理和前端分析则需要处理所有代码,因此随着应用大小的增长耗时尽管保持较为稳定的线性增长,但由于其耗时占比较少,对总耗时影响并不大。但是后端建模随着应用大小增长到一定规模时,涉及漏洞的功能点数量的后端需要分析代码增长会逐渐趋于平缓,由于其耗时占比大,因此总体耗时表现趋于平缓。这说明,本文方法在处理大规模的代码库时,仍然可以有较少的时间开销。

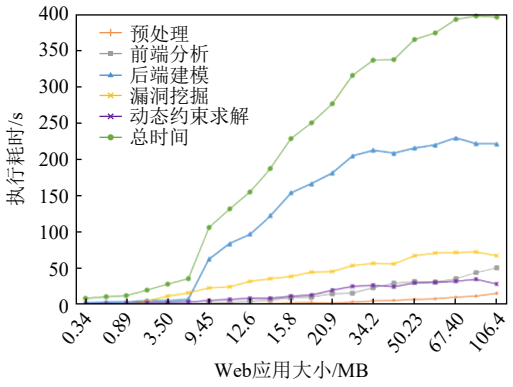


Fig. 3 Execution performance at each stage of the proposed method

图 3 本文方法各阶段执行性能

针对 RQ1,本文提出的前后端联合分析的 Jave 漏洞挖掘方法在实验中的总体表现较好,识别漏洞

的精确率达到了 0.79, 漏报率低至 0.12, 并在实验中发现了 14 个新漏洞, 能够真实有效地发现漏洞。对于大部分 Web 应用能够在 400 s 内完成所有的挖掘流程, 具有较好的性能。

4.3 消融实验（RQ2）

为了更好地说明前后端联合分析和动静态结合分析对于提升整体漏洞检测效果和性能具有重要作用, 我们设计相关实验对本文的创新模块进行评估。

4.3.1 前后端联合分析

1)检测效果分析。实验使用相同的漏洞样本作为输入, 分别测试在前后端联合分析以及单独后端检测的情况下的检测效果和性能情况。在无前端分析的检测流程中, 后端检测根据代码的特征来判断污点源。如图 4 所示, 由于本文检测方法的漏洞判断和验证的过程主要在后端漏洞挖掘和动态约束求解阶段完成, 因此前端分析对漏洞识别精确率的影响较小, 在关闭前端分析后, 漏洞识别的精确率和误报率没有产生很大变化。但是对于漏报率而言, 无前端分析的漏报率几乎是前后端联合分析的 3 倍, 因此证明通过前端分析发掘污点源和功能点能够很好地提高漏洞挖掘的覆盖率。

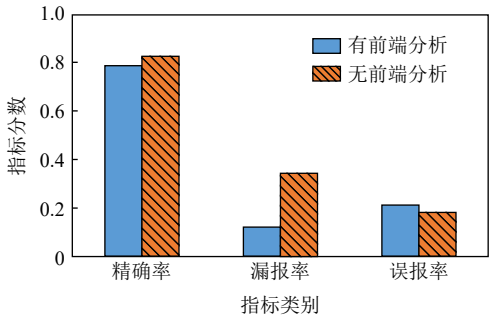


Fig. 4 Comparative chart of detection effect

图 4 检测效果对比图

如图 5 所示, 有无前端分析对于不同漏洞影响有所不同。针对反序列化漏洞和 SQL 注入漏洞这 2 类

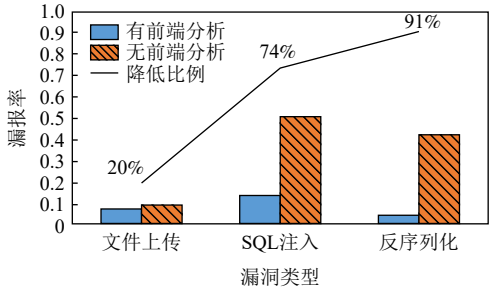


Fig. 5 Impact of the analysis with and without front-end on the leakage rate of different types of vulnerabilities

图 5 有无前端分析对不同类型漏洞漏报率影响

与数据流处理, 尤其是前后端通信密切相关的漏洞, 前端分析通过识别前端框架和与后端通信的 API, 可以帮助检测前端与后端交互过程中潜在的漏洞风险, 因此在漏报率的提升上更为明显。而文件上传漏洞通常有固定的接口函数, 尽管前端也能识别文件上传接口、上传限制等功能点, 但文件上传过程的核心依然是后端的验证和处理, 即文件上传的防范核心在后端, 因此前端分析对于漏报率的提升较为有限。

2)执行性能分析。实验的性能结果如图 6 所示。根据图中结果, 有无前端分析流程对总体的漏洞挖掘性能影响是很大的。当应用大小在 2 MB 以下时, 引入前端分析的性能提升较小, 当应用大小在 2 MB 以上时, 前端分析能够大幅缩短漏洞挖掘的耗时。前端分析可以有效地对后端分析模块代码建模和漏洞挖掘的流程进行剪枝, 显著提升性能, 尤其是针对 2 MB 以上的 Web 应用。

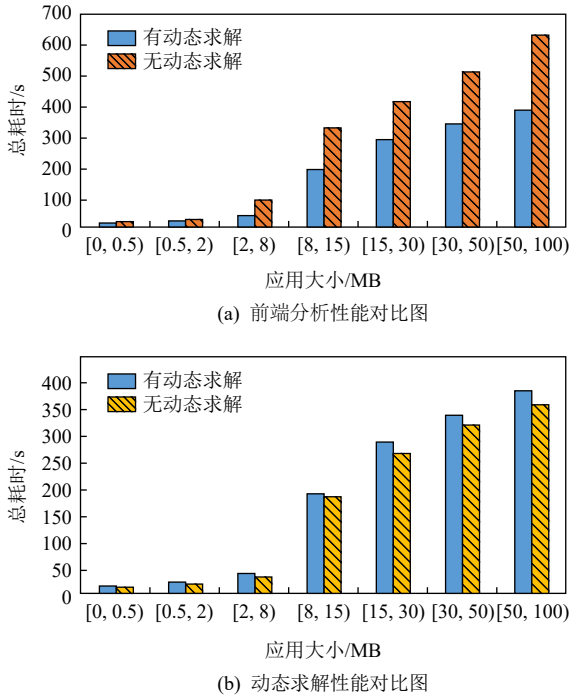


Fig. 6 Comparative chart of performance

图 6 性能对比图

4.3.2 动态约束求解

1)检测效果分析。使用漏洞样本作为输入, 分别测试有动态约束求解和无动态约束求解的检测效果。后者在静态分析过程得出约束条件后, 直接使用 Z3 求解器进行求解, 不进行轻量动态执行。漏洞检测效果如图 7 所示, 动态求解主要影响漏洞识别的精确率和误报率, 非动态求解会导致 0.40 以上的误报率, 这是由于 Java Web 漏洞的约束条件中往往

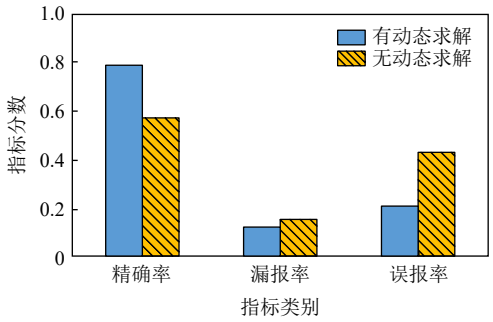


Fig. 7 Comparative chart of testing effect

图 7 检测效果对比图

包含许多动态信息,使用传统符号执行的静态方式进行求解是无法准确地执行的。

2)执行性能分析。由于采用了轻量执行的动态求解机制,因此能够在小幅提升开销的情况下带来比较大的检测效果的提升。

针对 RQ2,得出了 2 点结论:1)本文提出的前后端联合分析的方式能够有效提高漏洞挖掘的覆盖能力,降低漏报率,并通过前端指导后端分析的方式大幅提高执行的性能;2)本文方法能够在引入较小性能开销的前提下,较大地提升漏洞识别的精确率,降低误报率。

4.4 对比实验(RQ3)

目前在学术界关于 Java Web 漏洞自动化挖掘的研究较少,因此选择工业界比较常用的 Java 漏洞挖掘工具 FindBugs[28], CodeQL[29], ysoserial[30] 作为比较对象。这 3 个工具都是纯静态分析,且一般只考虑后端代码。本文目前只对文件上传、SQL 注入、反序列化这 3 种较为常见的漏洞进行了研究,因此后续的对比实验也在这 3 种漏洞范围内。表 2 显示了对比实验的结果。

1)检测效果分析。本文方法在漏洞识别精确率和漏报率上均优于其他方法。尽管 ysoserial 的精确率高达 0.94,但漏报率较高,而本文通过结合前端信息和 Java 动态特性覆盖了更多利用链,提升了检测效果。

2)执行性能分析。图 8 表明,当应用大小小于 50 MB 时,FindBugs 因实现简单而执行最快;但应用大小大于 50 MB 时,本文方法因采用前端指导后端分析机制,在处理大量代码时表现出更优性能。

针对 RQ3,本文方法总体上的漏洞检测效果和性能是最好的。FindBugs 虽然性能高,但识别效果差;CodeQL 在检测效果和性能上都不如本文方法;ysoserial 虽然对于反序列化漏洞具有很高的识别精确率,但是在利用链的覆盖率上也要低于本文方法。

Table 2 Overall Results of Comparative Experiments

表 2 对比实验总体结果

样本来源	漏洞类型	已知数量	识别数量①	精确率	漏报率
FindBugs[28]	文件上传	28	13/25	0.52	0.54
	SQL 注入	38	22/51	0.43	0.42
	反序列化	39	23/66	0.35	0.41
	ALL	105	58/142	0.41	0.45
CodeQL[29]	文件上传	28	21/36	0.58	0.25
	SQL 注入	38	29/42	0.69	0.24
	反序列化	39	27/34	0.79	0.31
	ALL	105	77/112	0.69	0.27
ysoserial[30]	反序列化	39	31/33	0.94	0.21
本文方法	文件上传	28	26/34	0.76	0.08
	SQL 注入	38	33/48	0.69	0.16
	反序列化	39	36/39	0.92	0.09
	ALL	105	95/121	0.79	0.12

注:①真实漏洞数/识别到的漏洞总数。

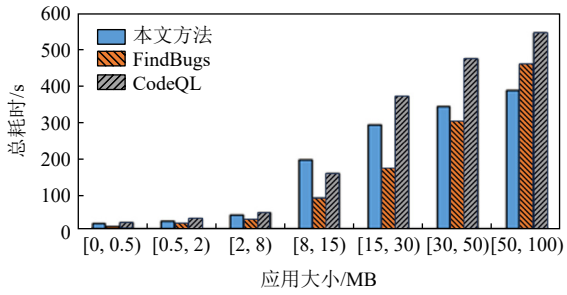


Fig. 8 Results of execution performance in comparative experiments

图 8 对比实验执行性能结果

5 讨 论

尽管本文提出的方法主要针对 Java Web 中最常见的 3 种漏洞进行了讨论,但是本文方法并不限于此。针对其他类型的漏洞,尤其是那些涉及数据流和输入输出过程的漏洞,都可以通过本文方法进行检测。例如,针对命令注入漏洞,数据流分析和污点追踪可以帮助发现用户输入是否通过后端不安全地传递给系统命令执行,进而提升识别潜在的命令注入漏洞的准确性;针对跨站脚本漏洞,通过前端参数名称的特定信息和后端的代码分析,可以帮助识别不当的用户输入和输出过程中的潜在漏洞。特别地,本文方法在结合静态分析和轻量级动态分析的基础上优化了性能,尤其适用于复杂的 Web 应用程序。未来我们考虑在更多的漏洞类型上应用本文方法进

行实证探索。

另外，本文方法在动态约束求解模块中用漏洞特定的 payload 字典来辅助求解。这是因为 Java 漏洞一般具有固定攻击模式，因此我们依据历史漏洞特征和漏洞检测工具模式来制定 payload 字典，并通过将 payload 赋值给污点变量而变异其他条件变量的策略，我们可以实现比模糊测试更加高效的动态求解，从而对漏洞进行充分的挖掘。虽然 payload 来自字典，但本文方法仍然具有对于零日漏洞和未知攻击的挖掘能力。在 4.2 节中，本文方法在开源社区数据集中发现了 19 个新漏洞，证明了本文方法识别新漏洞的潜力。尽管如此，对于一些具有全新或者未被广泛记录攻击模式的漏洞，本文方法仍然有所局限。未来我们考虑引入人工智能模型自动分析程序行为，挖掘潜在的、未知的攻击模式来进行改进。

6 结　论

本文设计了一种 Java Web 漏洞挖掘方法，通过前后端联合分析定位漏洞点，动静态分析结合通过轻量执行环境对漏洞约束进行动态求解，实现了较高精确率和性能的漏洞挖掘。通过实验证明了本文提出的检测方法的有效性，对于各种漏洞的识别精确率达到 0.79，漏报率低至 0.12，并发现了 14 个新漏洞，同时具有良好的批量检测性能。

作者贡献声明：邹福泰、姜开达、张亮参与方案设计；邹福泰、施纬负责方法设计；邹福泰、占天越、施纬撰写论文；邹福泰、姜开达负责审稿。

参　考　文　献

[1] Yang Yi, Li Ying, Chen Kai. Vulnerability detection methods based on natural language processing[J]. Journal of Computer Research and Development, 2022, 59(12): 2649−2666 (in Chinese)
(杨伊, 李滢, 陈恺. 基于自然语言处理的漏洞检测方法综述[J]. 计算机研究与发展, 2022, 59(12): 2649−2666)

[2] Cyber Safety Review Board. CSRB review: December 2021 Log4j event[R/OL]. [2024-08-23]. https://www.cisa.gov/cyber-safety-review-board

[3] Güler E, Schumilo S, Schloegel M, et al. Atropos: Effective fuzzing of Web applications for server-side vulnerabilities[C]//Proc of the 33rd USENIX Security Symp. Berkeley, CA: USENIX Association, 2024: 4765−4782

[4] Park S, Kim D, Jana S, et al. FUGIO: Automatic exploit generation for PHP object injection vulnerabilities[C]// Proc of the 31st USENIX Security Symp (USENIX Security 22). Berkeley, CA: USENIX Association, 2022: 197−214

[5] Zhang Guodong, Liu Zilong, Yao Tianyu, et al. Research on PHP injection vulnerability detection method based on intermediate language[J]. Journal of Cyber Security, 2024, 9(6): 114−128 (in Chinese)
(张国栋, 刘子龙, 姚天宇, 等. 基于中间语言的 PHP 注入漏洞检测方法研究[J]. 信息安全学报, 2024, 9(6): 114−128)

[6] Dahse J, Holz T. Simulation of built-in PHP features for precise static code analysis[C]// Proc of the 21st Network and Distributed System Security Symp (NDSS). Reston, Virginia: The Internet Society, 2014: 23−26

[7] Dahse J. RIPS─Free PHP security scanner using static code analysis[EB/OL]. [2024-12-18]. https://rips-scanner.sourceforge.net/

[8] The OpenJS Foundation. Node. js[EB/OL]. [2024-11-23]. https://nodejs.org/en/

[9] VMware Tanzu. Spring | home[EB/OL]. https://spring.io/

[10] Wang Juan, Gong Jiaxin, Lin Ziqing, et al. Multidimensional depth oriented fuzzing method of Java Web applications[J]. Netinfo Security, 2024, 24(2): 282−292 (in Chinese)
(王鹃, 龚家新, 蔺子卿, 等. 多维深度导向的 Java Web 模糊测试方法[J]. 信息网络安全, 2024, 24(2): 282−292)

[11] Doupé A, Cavedon L, Kruegel C, et al. Enemy of the state: A state-aware black-box Web vulnerability scanner[C]// Proc of the 33rd USENIX Security Symp (USENIX Security). Berkeley, CA: USENIX Association, 2012: 523−538

[12] Corradini D, Pasqua M, Ceccato M. Automated black-box testing of mass assignment vulnerabilities in RESTful APIs[C]//Proc of the 45th IEEE/ACM Int Conf on Software Engineering (ICSE). Los Alamitos, CA: IEEE Computer Society, 2023: 2553−2564

[13] Alhuzali A, Gjomemo R, Eshete B, et al. NAVEX: Precise and scalable exploit generation for dynamic Web applications[C]//Proc of the 27th USENIX Security Symp (USENIX Security). Berkeley, CA: USENIX Association, 2018: 377−392

[14] Gan Shuitao, Zhang Chao, Qin Xuwen, et al. CollAFL: Path sensitive fuzzing[C]//Proc of the 39th IEEE Symp on Security and Privacy (S&P). Los Alamitos, CA: IEEE Computer Society, 2018: 679−696

[15] Lee T, Wi S, Lee S, et al. FUSE: Finding file upload bugs via penetration testing[C]//Proc of the 27th Network and Distributed System Security Symp (NDSS). Reston, Virginia: The Internet Society, 2020: 1−17

[16] Huang Jin, Li Yu, Zhang Junjie, et al. UChecker: Automatically detecting php-based unrestricted file upload vulnerabilities[C]//Proc of the 49th IEEE/IFIP Int Conf on Dependable Systems and Networks (DSN). Los Alamitos, CA: IEEE Computer Society, 2019: 581−592

[17] Pellegrino G, Johns M, Koch S, et al. Deemon: Detecting CSRF with dynamic analysis and property graphs[C]//Proc of the 30th ACM Conf on Computer and Communications Security (CCS). New York: ACM, 2017: 1757−1771

[18] Pellegrino G, Constantin T, Bodden E, et al. Jak: Using dynamic analysis to crawl and test modern Web applications[C]//Proc of the 18th Int Symp on Recent Advances in Intrusion Detection (RAID). Berlin: Springer, 2015: 295−316

[19] Lee S, Wi S, Son S. Link: Black-box detection of cross-site scripting vulnerabilities using reinforcement learning[C]//Proc of the 31st ACM Web Conf. New York: ACM, 2022: 743−754

[20] Yin Zijing, Xu Yiwen, Ma Fuchen, et al. Scanner++: Enhanced vulnerability detection of Web applications with attack intent synchronization[J]. ACM Transactions on Software Engineering and Methodology, 2023, 32(1): 1−30

[21] Huang Jin, Zhang Junjie, Liu Jialun, et al. UFuzzer: Lightweight detection of PHP-based unrestricted file upload vulnerabilities via static-fuzzing co-analysis[C]//Proc of the 24th Int Symp on Recent Advances in Intrusion Detection (RAID). Berlin: Springer, 2021: 78−90

[22] Liu Xingbo, Li Yuanlin, Yu Mingjun, et al. Web vulnerability detection based on taint analysis and symbolic execution[J]. Computer Applications and Software, 2022, 39(11): 297−303 (in Chinese)
(刘行波, 李源林, 余明俊, 等. 基于污点分析与符号执行的 Web 漏洞检测[J]. 计算机应用与软件, 2022, 39(11): 297−303)

[23] Sun Jinan, Pan Kefeng, Chen Xuefeng, et al. Static analysis of injection security vulnerabilities based on symbolic execution[J]. Acta Scientiarum Naturalium Universitatis Pekinensis, 2018, 54(1): 1−13 (in Chinese)
(孙基男, 潘克峰, 陈雪峰, 等. 基于符号执行的注入类安全漏洞的分析技术[J]. 北京大学学报: 自然科学版, 2018, 54(1): 1−13)

[24] Khodayari S, Pellegrino G. JAW: Studying client-side CSRF with hybrid property graphs and declarative Traversals[C]// Proc of the 30th USENIX Security Symp (USENIX Security). Berkeley, CA: USENIX Association, 2021: 2525−2542

[25] Livshits V B, Lam M S. Finding security vulnerabilities in Java applications with static analysis[C]// Proc of the 14th USENIX Security Symp. Berkeley, CA: USENIX Association, 2005: 271−286

[26] Man Hongpeng, An Jing, Huang Wei, et al. JSEFuzz: Vulnerability detection method for Java Web application[C]//Proc of the 3rd Int Conf on System Reliability and Safety (ICSRS). Los Alamitos, CA: IEEE Computer Society, 2018: 92−96

[27] Vallee-rai R, Hendren L J. Jimple: Simplifying Java bytecode for analyses and transformations[EB/OL]. [2025-05-05]. https://api.semanticscholar.org/CorpusID:10529361

[28] The University of Maryland. FindBugs—FindBugs in Java programs [EB/OL]. [2024-09-08]. https://findbugs.sourceforge.net/

[29] GitHub. CodeQL[EB/OL]. [2024-08-12]. https://codeql.github.com/

[30] Frohoff C, Lawrence G. Marshalling pickles: How deserializing objects will ruin your day[EB/OL]. [2025-03-11]. https://frohoff.github.io/appseccali-marshalling-pickles/



Zou Futai, born in 1973. PhD, associate professor. Senior member of CCF. His main research interest includes network attack and defense technologies.
邹福泰, 1973 年生。博士, 副教授。CCF 高级会员。主要研究方向为网络攻防技术。



Jiang Kaida, born in 1980. Master, senior engineer. Member of CCF. His main research interests include educational informatization and cybersecurity.
姜开达, 1980 年生。硕士, 高级工程师。CCF 会员。主要研究方向为教育信息化、网络空间安全。



Zhan Tianyue, born in 2004. Undergraduate. Her main research interests include AI security and embodied intelligence.
占天越, 2004 年生。本科。主要研究方向为人工智能安全、具身智能。



Shi Wei, born in 1998. Master. His main research interests include Web security and machine learning.
施　纬, 1998 年生。硕士。主要研究方向为 Web 安全、机器学习。



Zhang Liang, born in 1974. PhD, professor of engineer. His main research interests include power dispatching automation and cybersecurity.
张　亮, 1974 年生。博士, 正高级工程师。主要研究方向为电力调度自动化、网络空间安全。