软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(7):2886−2910 [doi: 10.13328/j.cnki.jos.007581] [CSTR: 32375.14.jos.007581]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向 Web 应用漏洞检测的多数据流静态分析方法*

毛祥煜[1], 肖 庆[1], 戴嘉润[2], 何君尧[1], 谭 杰[1]

[1](杭州橙盾信息科技有限公司, 浙江 杭州 311121)

[2](复旦大学 计算与智能创新学院, 上海 200437)

通信作者: 谭杰, E-mail: tanjie.tj@alibaba-inc.com

摘　要: 作为 Web 应用安全漏洞检测的核心技术之一, 静态应用安全测试 (static application security testing, SAST) 具备广泛的业界应用场景. 然而, 现有静态分析工具受限于底层污点分析算法设计, 难以应对现代 Web 应用中的异步请求模式和多源输入语义等复杂逻辑, 直接影响其漏洞检测性能. 对此, 提出了一种面向 Web 安全漏洞检测的多数据流静态分析方法, 旨在对传统污点分析算法能力进行多维度扩展, 以提升其检测能力与泛用性: 在纵向维度上, 引入多段数据流分析, 通过关联与迭代算法综合考虑不同控制流路径上的数据依赖关系, 有效支撑需要多次异步调用触发的深层漏洞检测需求; 在横向维度上, 引入多标签数据流分析, 利用污染标签区分不同输入来源, 获取更细致的程序上下文语义信息, 提升与复杂语义相关的漏洞检测精确率. 基于上述方法实现了面向 Java/JavaScript Web 应用的漏洞检测原型系统 MultiFlow, 实验评估结果表明, 在包含 60 个真实 Web 应用与第三方组件的数据集中, MultiFlow 的多数据流分析方法具备良好的有效性, 在存储型、越权、原型链污染等复杂 Web 漏洞检测任务上分别取得了 87.18%、75.00% 与 83.72% 的精确率, 已获得 8 个 CVE 编号; 与现有方法相比, MultiFlow 以更少的分析开销实现了更高的漏洞检测准召率, 验证了其实用价值.

关键词: Web 应用安全; 漏洞检测; 静态分析; 污点分析; 数据流分析

中图法分类号: TP311

中文引用格式: 毛祥煜, 肖庆, 戴嘉润, 何君尧, 谭杰. 面向Web应用漏洞检测的多数据流静态分析方法. 软件学报, 2026, 37(7): 2886–2910. http://www.jos.org.cn/1000-9825/7581.htm

英文引用格式: Mao XY, Xiao Q, Dai JR, He JY, Tan J. Multi-data Flow Static Analysis Method for Vulnerability Detection in Web Applications. Ruan Jian Xue Bao/Journal of Software, 2026, 37(7): 2886–2910 (in Chinese). http://www.jos.org.cn/1000-9825/7581.htm

Multi-data Flow Static Analysis Method for Vulnerability Detection in Web Applications

MAO Xiang-Yu[1], XIAO Qing[1], DAI Jia-Run[2], HE Jun-Yao[1], TAN Jie[1]

[1](Hangzhou Orange Shield Information Technology Co. Ltd., Hangzhou 311121, China)

[2](College of Computer Science and Artificial Intelligence, Fudan University, Shanghai 200437, China)

Abstract: As a core technology for vulnerability detection in Web applications, static application security testing (SAST) holds widespread industrial applications. However, existing static analysis tools face challenges in handling complex logical structures in modern Web applications, such as asynchronous request patterns and multi-source input semantics, due to limitations in the underlying design of taint analysis algorithms. To this end, this study proposes a multi-data flow static analysis method for security vulnerability detection in Web applications, which is aimed at extending the ability of traditional taint analysis algorithms in multiple dimensions to improve both detection performance and generalization. Vertically, the multi-stage data flow analysis is introduced to comprehensively consider the data dependency across different control flow paths via correlation and iterative algorithms, thereby effectively supporting the detection of deep

* 本文由“智能化基础软件可信构造和供应链安全”专题特约编辑王林章教授、司徒凌云研究员、钟浩研究员、 向剑文教授、张路教授推荐.
收稿时间: 2025-08-29; 修改时间: 2025-10-20; 采用时间: 2025-12-08; jos 在线出版时间: 2025-12-26
CNKI 网络首发时间: 2026-06-02

vulnerabilities that require multiple asynchronous calls to be triggered. Horizontally, the multi-tag data flow analysis is introduced to distinguish different input sources via taint tags, thereby obtaining more detailed program context semantic information and enhancing vulnerability detection *Precision* related to complex semantics. Based on the above-mentioned method, a vulnerability detection prototype system named MultiFlow is developed for Java/JavaScript Web applications. Experimental evaluations demonstrate that MultiFlow's multi-data flow analysis method features sound effectiveness on a dataset containing 60 real-world Web applications and third-party components, yielding *Precision* of 87.18%, 75.00%, and 83.72% respectively on complex Web vulnerability detection tasks including stored vulnerabilities, broken access control, and prototype pollution, with eight CVE IDs obtained. Compared with the existing methods, MultiFlow achieves higher *Precision* and *Recall* with less analysis overhead, thereby validating its practical significance.

Key words: Web application security; vulnerability detection; static analysis; taint analysis; data flow analysis

随着互联网技术的快速发展, Web 应用程序已成为各行业数字化转型的核心载体, 广泛应用于金融、医疗、政务及电子商务等领域. 然而, 其开放性和复杂性使得 Web 应用面临日益严峻的安全威胁. 根据开放式 Web 应用程序安全项目 (Open Web Application Security Project, OWASP) 发布的 Top 10 Web 安全风险报告[1], 越权 (broken access control)、敏感数据泄露 (sensitive data exposure)、注入式攻击 (injection) 等安全问题位居前列, 成为最主要的 Web 漏洞类型. 若未能及时检测并修复这些漏洞, 将导致敏感数据大规模泄露、关键业务系统瘫痪乃至基础设施被恶意接管, 持续威胁数据资产安全与数字服务的完整性.

Web 漏洞检测的主要技术包括静态应用安全测试 (static application security testing, SAST)、动态应用安全测试 (dynamic application security testing, DAST)、交互式应用安全测试 (interactive application security testing, IAST)、渗透测试 (penetration testing) 与模糊测试 (fuzz testing) 等. 其中, DAST 与 IAST 技术通过运行时监控机制, 对程序执行路径、代码行为及数据进行实时观测, 从而识别可外部触发的安全缺陷. 但两类技术的检测效果高度依赖测试用例的完备性, 难以覆盖所有代码分支, 且需构建完整运行环境导致检测开销显著增加; 渗透测试与模糊测试技术采用模拟攻击者视角的策略, 通过人工构造攻击载荷或自动化生成变异输入数据, 触发潜在的程序异常. 二者虽能准确识别可实际触发的漏洞, 但存在计算资源消耗大、分析效率低下的缺陷, 尤其在处理复杂程序时检测成本呈指数级增长. 相较之下, SAST 技术通过分析源代码、字节码或二进制文件检测漏洞, 无须运行程序即可执行分析, 显著降低检测开销; 此外, SAST 技术能够实现更加全面的代码路径覆盖, 可给出漏洞代码精确位置, 适合在软件开发早期阶段快速定位潜在风险, 减少修复成本, 符合"安全左移 (shift-left security)"的开发理念, 这些特性使得 SAST 技术被大型开发团队广泛应用于持续集成/持续交付 (CI/CD) 流程中. 据相关报告显示, 全球 SAST 工具市场规模在 2023 年已达到 6.21 亿美元[2], 并以 7.1% 的复合年增长率持续扩张.

现有 SAST 技术的底层原理通常采用基于污点分析 (taint analysis) 的漏洞检测算法, 即通过追踪程序中污点源 (*Source*) 至污点汇 (*Sink*) 的污点传播路径, 判断是否存在安全漏洞. 然而, 传统污点分析在设计上仅关注单次程序执行路径中的单一类型污染数据流, 无法有效处理现代 Web 应用中的异步请求过程 (例如存储型 XSS (cross-site scripting) 漏洞场景下, 对输入内容的持久化存储与读取)、多类型输入数据源 (例如越权访问场景下, 用户输入与 Cookie/Session 信息的交互验证) 等复杂程序逻辑, 导致其在面对复杂类型 Web 漏洞场景时能力受限, 存在较高的漏报率与误报率. 近年来, Web SAST 领域的相关研究普遍聚焦于改进污点分析算法的具体实现[3−9], 而对于算法底层设计及 Web 场景下的局限性缺少充分认知; 针对异步 Web 请求间的隐式数据依赖问题, 部分工作采用启发式建模策略[10−12]对持久化存储前后的污染数据流进行关联, 来检测存储型 XSS 等漏洞, 但此类方案大多数仅能在特定语言环境下生效, 跨语言迁移成本较高, 不具备通用性; 针对 Web 应用的多源输入语义差异问题, 现有 Web 静态分析研究中同样缺乏有效的解决方案, 而二进制领域的相关工作[13−18]多基于内存单元或寄存器执行细致化污染标记, 不符合 Web 应用安全特性, 在工业级规模 Web 应用场景下将面临分析开销爆炸问题.

鉴于此, 本文提出一种基于多数据流分析的 Web 应用静态检测方法. 该方法通过纵向和横向扩展传统污点分析算法, 使其具备检测复杂类型 Web 安全漏洞的能力. 具体而言, 在纵向维度上, 本文增强了污染传播分析流程, 通过引入多段数据流分析技术, 综合考虑多条异步控制流路径上的数据共享与依赖关系; 在此基础上, 设计了关联与迭代分析算法, 能够精准检测异步控制流中潜在的存储型安全漏洞. 在横向维度上, 本文对污点概念进行了精细

化改进, 提出了多标签数据流分析技术, 该技术在同一控制流中对污染数据流进行细致化分类, 使用污染标签表征程序中不同输入变量, 捕获更加丰富的程序执行上下文语义信息, 进而实现对复杂 Web 漏洞的精确检测. 基于多数据流分析方法, 本文实现了面向 Java Web 和 JavaScript Web 应用的安全漏洞检测原型系统 MultiFlow, 支持检测数据库存储型漏洞、越权访问与原型链污染等 Web 漏洞.

本文构建了包含 30 个 Java Web 应用和 30 个 JavaScript Web 应用的真实应用数据集, 结合 OWASP benchmark 基准漏洞数据集, 从多方面对 MultiFlow 开展详细的实验评估. 实验结果表明, MultiFlow 具备良好的 Web 漏洞检测性能, 相较于 3 种现有 SAST 工具在检测准召率 (*Precision* and *Recall*) 与分析效率上均取得更优秀的表现; MultiFlow 的多段数据流分析方法共检测到 39 例数据库存储型漏洞, 精确率为 87.18%, 涉及 XSS、SSRF (server-side request forgery)、远程代码执行 (remote code execution, RCE) 等多种 Web 漏洞类型; MultiFlow 的多标签数据流分析方法以较小的额外开销, 实现对各类复杂 Web 漏洞的精准检测能力, 共识别到 40 例越权访问漏洞 (精确率为 75.00%)、43 例原型链污染漏洞 (精确率为 83.72%), 获得 8 个 CVE 编号.

本文的主要贡献与创新点如下.

(1) 提出面向 Web 应用安全场景的多数据流静态分析方法, 通过引入多段数据流分析与多标签数据流分析两类子方法, 在纵、横两个维度扩展了传统的污点分析能力, 显著提升了其对复杂漏洞场景的覆盖能力与检测精确率.

(2) 基于多数据流分析方法, 实现了面向 Java Web 与 JavaScript Web 应用的安全漏洞检测原型系统 MultiFlow, 并针对 Web 应用场景进行了多项优化, 提升了实用性与可扩展性.

(3) 在真实 Web 应用与漏洞基准数据集上对 MultiFlow 进行全面评估, 验证了多数据流分析方法的有效性, 同时表明 MultiFlow 在 Web 漏洞检测准召率与分析效率上, 相较于现有静态分析技术具备明显优势.

本文第 1 节介绍相关研究背景. 第 2 节介绍本文提出的多数据流分析方法设计及其应用场景. 第 3 节介绍分析原型系统 MultiFlow 的具体实现细节. 第 4 节对 MultiFlow 开展实验评估, 并对评估结果进行分析. 第 5 节对研究做进一步讨论. 第 6 节介绍与本文相关的其他工作. 第 7 节总结全文.

1 研究背景

1.1 控制流与数据流

程序分析[19]是指软件工程领域中通过形式化方法, 对程序结构、行为及正确性、健壮性和安全性等特征进行系统性研究的过程. 控制流 (control flow) 和数据流 (data flow) 是程序分析领域的两个核心概念, 用于描述程序的不同行为特性.

控制流是指程序执行过程中指令的执行顺序, 反映了程序逻辑的结构和流程. 它决定了程序中哪些代码会被执行, 以及执行的先后顺序. 控制流分析通常基于控制流图 (control flow graph, CFG) 开展, 控制流图由基本块 (basic block) 和边 (edge) 构成. 基本块代表一段顺序执行的代码序列, 具有单一入口和出口, 内部无分支或跳转; 边表示基本块间的控制转移关系 (如条件跳转、循环跳转). 在图 1 展示的代码片段中, 程序在执行至 if-else 分支语句时, 将根据 condition 变量的取值情况进入两条不同的控制流分支, 并在 x = call(c) 语句处汇合.

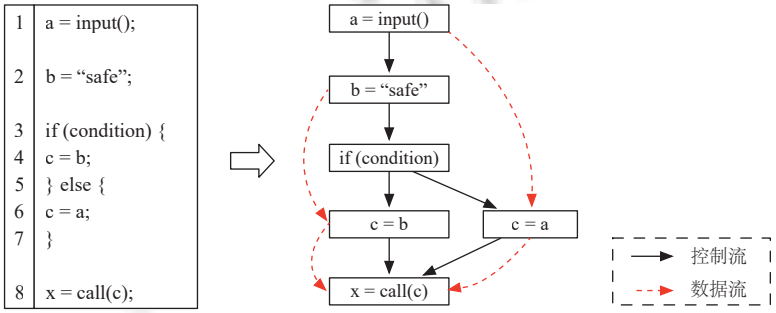


图 1 代码控制流与数据流示例

当程序执行时, 数据会在变量赋值、函数调用、模块之间传递, 形成数据流动的路径, 即数据流. 数据流描述程序中数据的流动与变换过程, 其伴随着控制流而发生. 图 1 所示的两条控制流路径上分别包含对应的数据流传播过程, 二者最终交汇于 x = call(c) 语句, 表明变量 c 可能来自用户输入变量 a 或字符串常量 b.

1.2 Web 应用安全与污点分析技术

Web 应用 (Web application) 是基于 Web 技术开发的应用程序, 其核心逻辑运行于服务器端, 通过 HTTP/HTTPS 协议与客户端代理交互, 动态处理请求并生成响应. Web 应用安全旨在保护此类应用免受恶意攻击和漏洞利用, 其焦点集中于应用层风险, 典型例子如 OWASP Top 10 漏洞类型. Web 应用安全通常以接口为分析基本单位——每个接口在代码层面对应一个入口函数 (entry point), 即负责接收客户端 HTTP 请求并返回响应的函数或方法, 作为 Web 程序的实际执行入口. 此外, 相较于内存安全而言, Web 应用安全更关注应用层语义, 如重点分析变量是否可能被恶意输入污染, 而无须精确追踪其具体取值.

污点分析 (taint analysis) 是一种程序分析技术, 其核心思想是将特定的数据源标记为“污点源”, 并监控这些数据在程序中的流动, 当未经安全处理的污染数据流入敏感逻辑时, 报告潜在安全漏洞. 早在由 Liskov[20]提出的 CLU 语言中, “数据污染”概念便已经出现, 其用于验证程序的类型安全性和数据完整性. 污点分析技术首次被明确用于安全领域, 则可追溯到 Perl 语言中引入的“Taint mode”安全检查机制[21]. 图 2 展示了静态污点分析算法的基本工作流程. 算法以待分析的程序中间表示 *P* 和漏洞规则 *Rule* 作为输入, 规则中配置有污点源模式集合 *Source*、污点汇模式集合 *Sink* 和净化函数集合 *Sanitizer* 等安全相关方法 (security-relevant method, SRM) 信息. 算法包含 3 个主要步骤: (1) 基于规则中预设的污点源模式, 匹配程序中指定类型的输入参数, 将其标记为污染变量; (2) 执行数据流分析, 追踪污点数据随程序代码执行的传播与净化过程; (3) 当检测到未被净化的污染数据流入敏感操作时, 报告潜在漏洞.

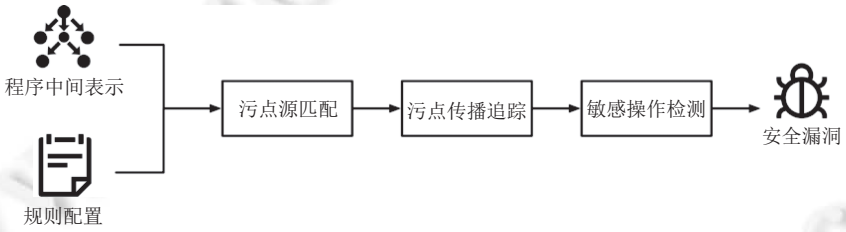


图 2 传统污点分析流程

然而, Web 应用程序的逻辑复杂性和独特交互模式, 使得传统污点分析算法在 Web 安全领域面临双重挑战.

(1) 缺少跨控制流的异步追踪能力. Web 应用程序多采用请求-响应架构, 依赖会话保持与异步 I/O 机制, 导致恶意载荷可通过 HTTP 会话、数据库持久化、线程共享数据等中间介质跨执行过程传播. 传统污点分析基于单次程序控制流轨迹, 难以追踪此类隐式数据流. 例如图 3(a) 展示的存储型 XSS 漏洞中, 攻击载荷进入数据库存储 (第 3 行) 的请求与内容渲染请求 (第 10 行) 分属于不同控制流, 且二者间无显式调用关系, 传统方法因忽略跨 HTTP 事务的数据依赖而产生漏报.

(2) 细粒度语义表征能力不足. Web 应用的输入空间具有显著异构性, 涵盖结构化数据 (HTTP 请求参数、JSON/XML 请求体)、非结构化数据 (文件内容) 及隐式输入来源 (Cookie/Session 变量) 等. 而现有方法采用粗粒度污染标记策略, 即将不同输入简单划分为同类型污染/非污染变量, 未充分考虑输入语义差异性, 可能导致执行上下文信息丢失, 影响复杂漏洞检测精确度. 例如图 3(b) 中, 第 7 行的密码重置操作虽然接收用户控制的 password 参数, 但在执行时通过不可伪造 Session 凭证完成身份校验. 传统方法忽视该语义关联性, 将产生越权漏洞误报.

近年来, Web 静态应用安全测试技术的相关研究工作多聚焦于改进污点分析算法的具体实现[3−9], 而在底层检测原理上均遵循传统污点分析算法框架, 故在分析图 3 所示程序代码逻辑时, 仍然无法避免其局限性; 部分工作采用启发式建模策略[10−12]处理异步执行过程间的数据依赖关系, 但方案适用范围较窄, 缺乏普遍性; 对于多源输入语义差异特性, 同样少有研究者进行关注. 因此, 亟需一种通用的污点分析算法增强方案, 以扩展现有 Web 应用静态分析技术的能力与适用场景.

```
app.post("/comment", (req, res) => {
  const {content} = req.body;
  db.query(
  "INSERT INTO comments (content) VALUES (?)",
  [content]
  );
  res.redirect("/");
});
app.get("/", (req, res) => {
  db.query(
  "SELECT * FROM comments",
   (err, results) => res.render("index", {comments: results});
  );
});
```

(a) 跨控制流异步场景

```
@RequestMapping("updatePassword")
public void updatePassword(String password) {
    int userId = Integer.parseInt(SessionHelper.getOperateId());
    User user = new User();
    user.setId(userId);
    user.setPassword(password);
    userMapper.update(user);
}
<!-- UserMapper.xml -->
<mapper namespace="com.example.mapper.UserMapper">
  <update id="update" parameterType="com.example.entity.User">
     UPDATE user SET password = #{password} WHERE id = #{id}
  </update>
</mapper>
```

(b) 细粒度输入场景

图 3　复杂 Web 应用程序逻辑示例

2　面向 Web 应用安全的多数据流分析方法

针对现有 Web 静态分析技术因底层污点分析局限而导致的适用性不足、检测准召率低等问题, 本文提出了一种面向 Web 应用安全场景的多数据流静态分析方法. 该方法通过引入多段数据流分析与多标签数据流分析两类子方法, 从多个维度对图 2 所示的传统污点分析算法进行增强, 显著提升其在复杂 Web 应用程序场景下的检测能力. 同时, 该方法不依赖于具体的编程语言, 具备良好的泛用性. 以下将详细阐述多数据流分析方法的设计思路及其应用场景.

2.1　方法设计

2.1.1　多段数据流分析

多段数据流分析方法的整体流程如图 4(b) 所示, 将传统的单次污点传播追踪过程扩展为多轮次 (multi-pass) 迭代分析形式, 从而实现对污点分析算法的纵向能力增强. 其核心思想是关注 Web 应用程序中常见的跨过程持久化数据 (如数据库表、线程共享内存、全局会话变量, 以下简称共享数据). 在污点分析过程中, 方法通过记录污染变量流入共享数据并存储的操作, 捕捉不同执行路径间的污染数据流关联, 从而实现对异步执行过程中的 Web 安全漏洞检测能力.

算法 1 为多段数据流迭代分析的伪代码实现. 算法以待分析应用程序的中间表示 P 和目标漏洞规则 *Rule* 为输入, 在实际分析中, P 可根据需求选用程序的抽象语法树、控制流图或其他中间表示形式. 算法首先基于规则中配置的污点源模式信息, 初始化数据源模式集合 *SourceSet* 并执行首轮污点分析. 具体而言, 对于 *SourceSet* 中的每个污点源模式, 在 P 中匹配符合该模式的输入变量并标记为污点, 之后遵循传统污点分析的工作原理, 沿控制流跟踪污染变量在程序中的传播与净化过程, 将污染数据流经过的所有代码语句添加至 *TaintStmts* 集合中.

算法 1. 多段数据流迭代分析算法.

输入: 待分析应用 *P*, 漏洞规则 *Rule*;
输出: 安全漏洞集合 *Vulns*.

1. *Vulns* ← ∅;
2. *SourceSet* ← *Rule.Source*;
3. *VisitedOps* ← ∅;
4. **repeat**
5.　　*TaintStores* ← ∅;
6.　　*TaintStmts* = *taintPropagation*(*P*, *SourceSet*);

```
7.    for each s ∈ TaintStmts do
8.      if hasSink(Rule.Sink, s) then
9.        Vulns ← Vulns ∪ report(s);
10.     end if
11.     if hasDataStore(s) and s ∉ VisitedOps then
12.       VisitedOps ← VisitedOps ∪ {s};
13.       TaintStores ← TaintStores ∪ getStoredPath(s);
14.     end if
15.   end for
16.   TaintReads ← calculateReadLink(TaintStores);
17.   SourceSet ← TaintReads \ SourceSet;
18. until TaintReads == ∅;
19. return Vulns;
```

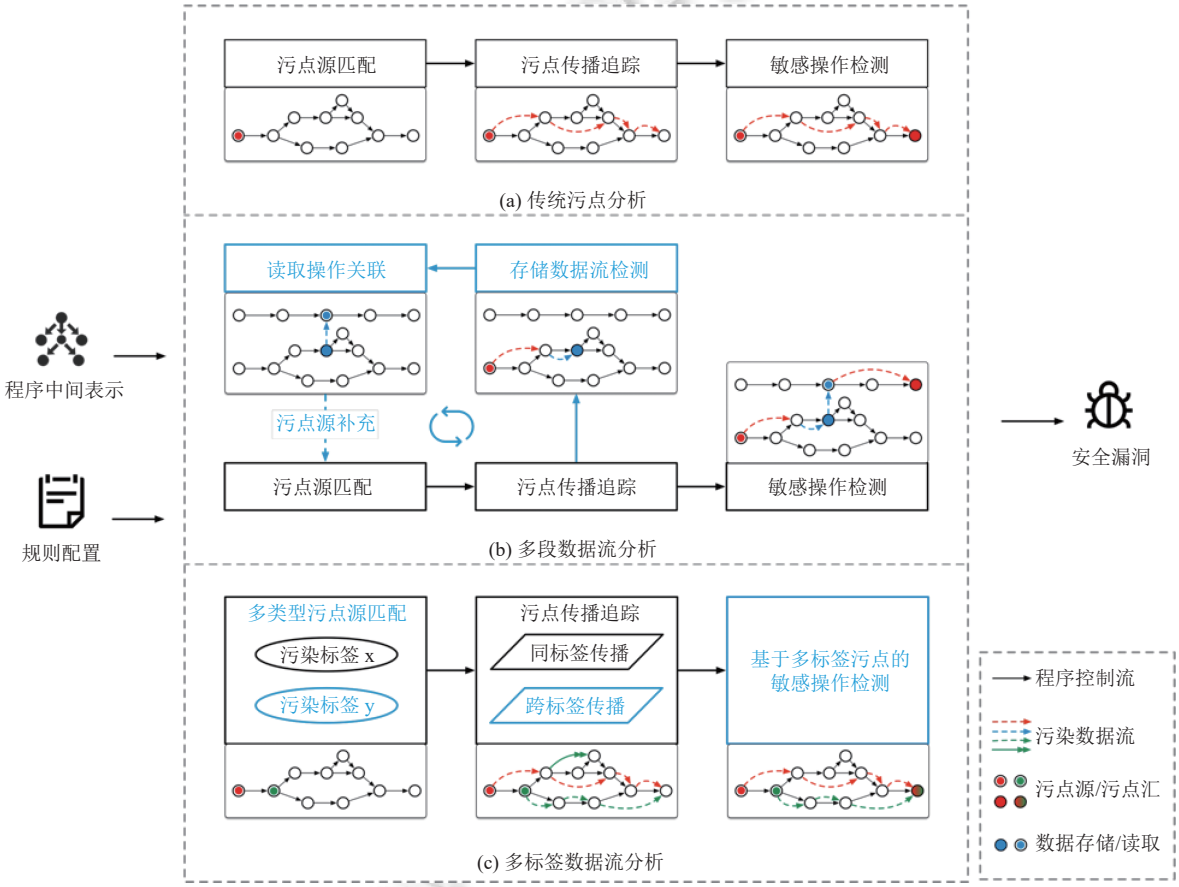


图 4　多数据流分析方法概览

在完成单个轮次的污点传播追踪分析后, 算法将检查 *TaintStmts* 中的语句是否满足如下两类情况: (1) 若语句包含与目标漏洞相关的敏感操作, 则报告安全漏洞; (2) 若语句包含针对共享数据的写入操作, 且该操作在前序迭代轮次中未被分析过, 则执行对数据存储位置的细粒度域敏感分析 (如写入数据表的数据列、对象字段名等), 将

其添加至存储数据流集合 *TaintStores* 中. 对于 *TaintStores* 中收集到的所有共享数据, 算法根据共享数据类型、框架等信息, 计算其对应的数据读取操作映射, 得到共享数据读取操作集合 *TaintReads*, 实现对存储段污染数据流与读取段污染数据流的关联; 之后将未被分析过的所有读取操作转换为污点源模式配置, 添加至下一轮次分析的数据源模式集合中. 重复迭代此过程, 直至分析结果中不再出现新的共享数据写入操作时终止.

算法的迭代次数为 n 时, 理论上可检测到首尾相接的 n 段程序控制流中污染数据流的传播过程. 由程序中共享数据写入和读取操作等元素的有限性, 易知该迭代过程必然在有限轮次内收敛. 同时不难发现, 当程序中不存在共享数据写入操作时, 该算法退化为传统污点分析的等价形式, 这保证了多段数据流分析相较于污点分析在漏洞检测能力上的后向兼容性. 算法的分析复杂度与代码中的共享数据分布密度、迭代轮次数量存在关联, 相关细节讨论见第 5.2 节.

2.1.2 多标签数据流分析

多标签数据流分析算法的整体流程如图 4(c) 所示, 其核心思路为引入污染标签概念, 用于区分不同语义类型的污染变量, 进而横向增强现有污点分析算法对代码语义的理解与分析能力. 形式化地, 对于给定的污染标签集合 $T=\{t_1,t_2,t_3,\ldots\}$, 多标签数据流分析将对传统污点分析算法的规则配置与流程中各个阶段执行如下增强.

- 漏洞规则中, 引入多标签污点源集合 $\mathbb{S}_{\text{source}}\subseteq Source\times T$, 其满足如下条件:

$$\forall s\in Source,\exists t\in T,(s,t)\in\mathbb{S}_{\text{source}} \tag{1}$$

即需要为每个污点源模式指定一种或多种类型的污染标签信息.

- 污点传播追踪阶段, 算法将维护如下形式的多标签污染变量集合: $\mathbb{V}_{\text{taint}}=\{\langle v,t\rangle|t\in T\}$, 其中 v 代表污染变量, t 代表其对应的污染标签. 在污点分析过程中, 多标签污染变量集合存在如下两类元素更新方式.

(1) 通过污点源模式匹配, 将匹配到的污染变量加入集合时, 为其赋予与污点源模式对应的所有污染标签:

$$(s,t)\in\mathbb{S}_{\text{source}}\wedge s\xrightarrow{\text{match}}v\Rightarrow\mathbb{V}_{\text{taint}}\leftarrow\mathbb{V}_{\text{taint}}\cup\{\langle v,t\rangle\} \tag{2}$$

(2) 通过已有污染变量传播至新变量 v', 此时算法将遵循传统污点分析策略, 即在默认条件下, 独立处理不同类型污点数据的传播与净化过程. 此外, 漏洞规则中支持自定义跨类型的污染传播谓词 isCrossTag(·), 若满足谓词条件, 算法将在传播污点时对不同类型的污染标签进行相互转化, 以进一步提高对程序语义的表征能力, 增加方法扩展性.

$$\begin{cases}\langle v,t\rangle\in\mathbb{V}_{\text{taint}}\wedge v\xrightarrow{\text{taint}}v'\Rightarrow\mathbb{V}_{\text{taint}}\leftarrow\mathbb{V}_{\text{taint}}\cup\{\langle v',t\rangle\}\\ \langle v,t\rangle\in\mathbb{V}_{\text{taint}}\wedge\text{isCrossTag}(\langle v,t\rangle,\langle v',t'\rangle)\Rightarrow\mathbb{V}_{\text{taint}}\leftarrow\mathbb{V}_{\text{taint}}\cup\{\langle v',t'\rangle\}\end{cases} \tag{3}$$

- 敏感操作检测阶段, 将基于如下增强污点汇集合定义进行分析:

$$\mathbb{S}_{\text{sink}}=\{\langle s,\text{isVuln}(\mathbb{V}_{\text{taint}})\rangle|s\in Sink\} \tag{4}$$

其中, s 表示污点汇模式, 用于匹配与漏洞相关的敏感操作; isVuln(·) 表示漏洞约束条件, 其功能除分析污染变量是否流入敏感操作外, 还支持根据敏感操作中各个污染变量的污染标签信息进行综合分析, 由此实现更强大的安全漏洞检测能力.

若多标签数据流分析规则配置中仅包含单个污染标签, 则其分析过程与传统污点分析等价, 这表明多标签数据流分析在传统 Web 漏洞检测任务中同样具备后向兼容性. 实际分析过程中, 污染标签数量为有限值, 且绝大多数变量仅对应单一类型的污染标签, 这些因素使得多标签数据流分析在性能上可接近传统污点分析的表现, 相关细节将在第 5.2 节中详细讨论.

2.2 漏洞检测场景应用

本节将基于多种复杂类型 Web 安全漏洞, 介绍两类多数据流分析方法的实际应用场景与检测技术实现.

2.2.1 数据库存储型漏洞

作为 Web 应用中与业务逻辑和用户交互密切相关的核心组件, 数据库一直是 Web 安全领域的重要关注对象. 本文使用“数据库存储型漏洞”这一概念, 指代所有利用过程中涉及数据库读写及内容持久化存储的漏洞类型, 包括但不限于存储型 XSS、二次注入等. 在数据库存储型漏洞场景中, 各个数据表中存放的元组对应共享数据概念, INSERT 和 UPDATE 查询对应共享数据存储操作, 而 SELECT 查询则对应共享数据读取操作. 下文以 Java Web

应用中常用的数据库框架 MyBatis[22]为例, 介绍多段数据流分析方法在检测存储型 Web 漏洞中的具体应用方式.

在图 5 所示的代码片段示例中, 多段数据流分析方法在首轮迭代时检测到 commentMapper.insertComment 调用了数据库插入语句 (即共享数据存储操作). 结合域敏感的污染传播分析结果可知, 用户输入参数 request 将传播至 comment.content 字段, 进一步分析得到该字段对应的存储位置为 comment 表的 content 列, 从而生成如下共享数据存储记录: {table: “comment”, column: “content”}.

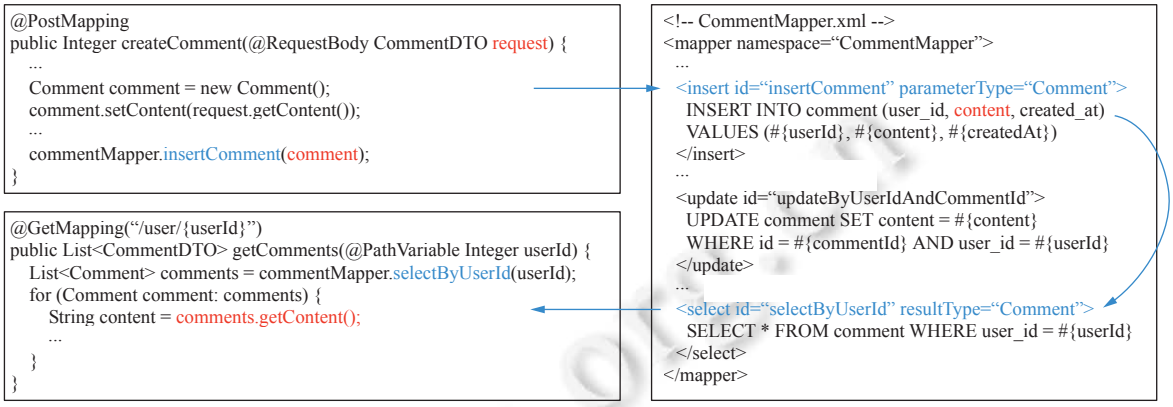


图 5 共享数据分析过程

随后, 算法扫描同一命名空间下的所有数据库操作, 关联该记录对应的共享数据读取操作. 此处存在一个基本假设: 同一命名空间下配置的所有 SQL 查询均针对同一数据库. 通过关联分析, 算法发现 selectByUserId 调用将读取上述污染数据, 因此在下一轮分析中将其标记为污点源. 后续轮次分析中, 在执行污点源匹配调用操作时, 方法同样支持域敏感的污点分析能力, 即标记 selectByUserId 调用返回对象的 content 字段为污染变量, 保证后续污染传播追踪的精确率.

2.2.2 越权访问

访问控制缺陷 (broken access control), 又称越权访问漏洞, 指系统在身份认证与权限校验机制设计或实现层面存在安全隐患, 导致攻击者可通过非授权路径绕过访问控制约束, 执行超出当前权限的操作. 漏洞产生根源为程序代码逻辑在处理敏感请求时, 未对请求主体的权限属性与请求客体的所需权限实现完善的一致性验证. 根据实际攻击方式, 越权访问漏洞可被进一步细分为垂直越权 (vertical privilege escalation)、水平越权 (horizontal privilege escalation) 等子类型. 前者主要体现为低权限实体通过未授权接口调用获取更高特权, 如普通用户访问管理后台; 后者的常见表现形式为攻击者通过篡改请求参数, 访问其他同等级实体的私有数据.

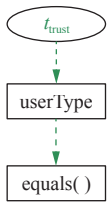
越权访问漏洞检测的核心挑战源于其漏洞成因与业务逻辑的高度耦合性, 即漏洞触发条件通常依赖特定业务逻辑中的上下文信息, 而传统污点分析算法在建模复杂语义关联方面的能力较弱, 导致在检测越权访问漏洞时面临较大困难. 具体而言, 能否准确识别现代 Web 应用中多样化的鉴权实现方式, 进而判断未受鉴权逻辑保护的敏感操作, 直接决定了静态分析技术对于越权访问的检测性能. 本文通过对大量真实应用分析发现, 许多鉴权逻辑的实现中都涉及由 Web 应用服务端安全框架或会话管理机制生成的用户身份信息相关变量, 例如会话信息中的登录态用户标识参数. 为便于描述, 下文统一使用“可信源 (trusted source)”概念指代此类变量. 可信源的特点在于其数据来源不可篡改, 通常与用户身份认证状态和访问控制策略直接关联, 可作为权限判定的核心依据. 因此, 在分析过程中重点关注可信源与用户输入参数的交互逻辑, 即可有效识别此类鉴权操作. 以下将介绍基于多标签数据流分析的越权访问漏洞检测算法实现.

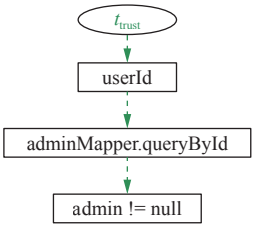
● 污染标签配置. 越权访问漏洞规则使用如下污染标签集合: $T = \{t_{\text{input}}, t_{\text{trust}}\}$, 除基本的用户输入污染标签 t_{input} 外, 为可信源变量设置一类独立的污染标签 t_{trust}.

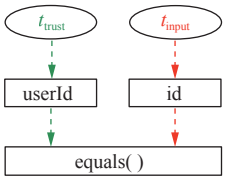
● 污点源匹配. 对于用户输入类型污点, 采取 Web 应用中常见的用户输入模式 (如 HTTP 请求参数); 可信源的来源模式与应用具体实现相关, 且单个应用中通常采取若干固定的可信源类型. 在实际分析中, 需结合 Web 应

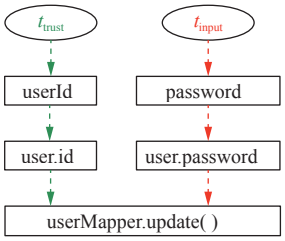
用的开发框架与协议等信息, 灵活确定程序中与可信源相关的污点源模式, 例如图 6(a) 中, 将 SessionHelper.getOperateUserType 调用语句返回值作为可信源模式.

分类	鉴权代码相关片段	多标签污点分析过程

垂直鉴权 I	@RequestMapping(“/user/update”)
	public void updateUserInfo(String id, String password) {
	String userType = SessionHelper.getOperateUserType();
	if (userType.equals (“admin”)) {
	User user = new User();
	user.setId(id);
	user.setPassword(password);
	userMapper.update(user);
	}
	} (a)
		

垂直鉴权 II	@RequestMapping(“/user/update”)
	public void updateUserInfo(String id, String password) {
	String userId = SessionHelper.getOperateId();
	User admin = adminMapper.queryById(userId);
	if (admin != null) {
	...
	userMapper.update(user);
	}
	} (b)
		

水平鉴权 I	@RequestMapping(“/user/update”)
	public void updateUserInfo(String id, String password) {
	String userId = SessionHelper.getOperateId();
	if (userId.equals(id)) {
	...
	userMapper.update(user);
	}
	} (c)
		

水平鉴权 II	@RequestMapping(“/user/update”)
	public void updateUserInfo(String id, String password) {
	String userId = SessionHelper.getOperateId();
	User user = new User();
	user.setId(userId);
	user.setPassword(password);
	userMapper.update(user);
	// update user set password = #{password} where id = #{id}
	} (d)
		

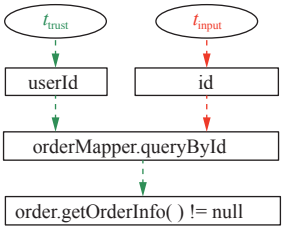
水平鉴权 III	@RequestMapping(“/order/update”)
	public void updateOrder(String orderId, String pay State) {
	String userId = SessionHelper.getOperateId();
	Order order = orderMapper.queryById(userId, orderId);
	if (order.getOrderInfo() != null) {
	order.setPayState(payState);
	orderMapper.update(order);
	}
	} (e)
		

图 6 共享数据分析过程

● 污点传播追踪. 该阶段为两类污染变量独立执行污点传播与净化分析; 此外, 为实现鉴权操作识别, 算法将关注代码中某些特定的操作模式, 并基于可信源与用户输入污染变量信息, 综合判断相关操作是否实现了对用户身份的验证逻辑. 一般来说, 若鉴权操作中对可信源进行独立验证, 通常代表该操作为垂直鉴权; 将可信源与用户输入变量进行共同验证, 则通常为水平鉴权逻辑. 具体而言, 算法总结了如下几类常见的鉴权模式.

(1) 垂直鉴权 I. t_{trust} 流入条件语句比较逻辑. 该类型鉴权逻辑实现方式为直接使用可信源验证用户身份信息. 例如图 6(a) 展示的代码片段中, 可信源变量 userType 被用于验证当前用户是否拥有管理员权限.

(2) 垂直鉴权 II. t_{trust} 流入数据库查询, 并通过验证查询返回结果实现权限校验. 例如图 6(b) 所示代码片段, 其语义为根据可信源变量 userId, 在管理员表中查询是否存在当前用户.

(3) 水平鉴权 I. t_{trust} 与 t_{input} 同时流入比较逻辑, 鉴权实现方式为验证用户输入与可信源中相关信息的一致性. 例如图 6(c) 代码片段中, 比较 userId 与 id 的值是否相等.

(4) 水平鉴权 II. t_{trust} 和 t_{input} 同时流入数据库查询, 且要求 t_{trust} 在查询的条件子句中被使用. 该类型鉴权逻辑意在执行敏感操作时, 通过可信源直接限制操作的目标数据, 达到水平鉴权目的. 例如图 6(d) 代码片段中, 用户输入变量 password 与可信源变量 userId 同时流入了数据库操作, 在查询时通过 where 子句, 限制了该操作仅可修改当前用户自身信息.

(5) 水平鉴权 III. t_{trust} 和 t_{input} 同时流入数据库查询后验证, 其原理类似垂直鉴权 II 与水平鉴权 II 的结合形式.

对于符合以上模式的程序代码片段, 算法在识别后将其加入鉴权操作集合 *AuthOps* 中. 需指出的是, 此处列出的鉴权模式样例均为应用业务逻辑执行时的显式鉴权. 对于现代 Web 应用中常见的框架鉴权、切面鉴权等其他鉴权实现方式, 与显式鉴权的主要差异在于鉴权行为在代码执行流上的具体介入时机 (例如在请求路由匹配后、业务逻辑执行前完成鉴权). 考虑其鉴权机制核心实现, 通常仍涉及可信源与用户输入信息交互的相关代码逻辑, 故可将这些鉴权方式进一步归纳为相似建模处理.

● 敏感操作检测. 根据越权访问漏洞的形成原理, 可将 isVuln 约束谓词定义为如下形式:

$$\text{isVuln}(s) \equiv \exists \langle v, t_{\text{input}} \rangle \in \mathbb{V}_{\text{taint}}, v \xrightarrow{\text{flow}} s \wedge \neg \exists a \in \mathit{AuthOps}, (\text{dominate}(\text{Entry}, a) \wedge \text{dominate}(a, s)) \tag{5}$$

即在检测越权访问时, 算法除关注带有 t_{input} 标签的污染变量是否流入数据库增删改等敏感操作外, 还关注从程序入口到达敏感操作的控制流路径上, 是否存在控制流支配关系的鉴权操作 (例如图 6(a) 所示代码片段, 易知在执行用户表更新操作前, 必然执行了用户身份验证), 不存在则报告漏洞.

2.2.3 原型链污染

原型链污染 (prototype pollution) 是 JavaScript 语言独有的一类安全漏洞, 其产生根源为 JavaScript 中的动态原型继承机制. 在 JavaScript 应用程序中, 所有对象通过隐式的__proto__属性与其构造函数的原型对象关联, 形成一条原型链. 当访问对象属性时, JavaScript 会沿着原型链逐级向上查找, 直至找到目标属性或到达原型链顶端. 由于 JavaScript 未限制对原型对象的修改, 攻击者可以通过某些不安全的操作, 将恶意属性注入顶层原型 Object.prototype 中, 进而在后续程序逻辑中影响所有继承该原型的对象, 可能导致拒绝服务 (denial of service, DoS)、远程代码执行或其他高危后果.

与传统注入类 Web 安全漏洞相比, 原型链污染不存在明确的敏感操作锚点, 且与 JavaScript 引擎原型链解析机制存在高度关联、污染逻辑与正常业务逻辑交织紧密、涉及隐式传播等诸多特性, 使得无法直接采用传统污点分析算法进行检测. 现有的原型链污染检测相关工作[4,5]多从 JavaScript 语言底层特性出发, 通过对象属性图等复杂数据结构模拟代码中的隐式数据依赖与传播关系, 其检测方法开销极大, 在工业级场景下难以应用.

本文通过分析若干真实漏洞样例发现, 从语义层面来看, 典型场景下原型链污染漏洞的利用过程可概括为两个阶段: (1) 通过恶意操作劫持顶层原型对象 Object.prototype; (2) 向原型对象中注入污染属性, 或覆盖其原有属性. 基于这一特性, 可利用多标签数据流分析方法, 通过独立的污点类型跟踪原型对象及其相关污染数据流的传播过程, 实现轻量级的原型链污染检测方案. 需要指出的是, 本文聚焦于原型链污染漏洞中污染步骤的检测, 即攻击者通过恶意输入对 JavaScript 顶层原型对象 Object.prototype 属性进行污染的行为. 若需进一步分析污染属性对程序

逻辑的后续影响, 可结合前述多段数据流分析方法进行. 以下将详细介绍基于多标签数据流的原型链污染漏洞检测算法的实现.

● 污染标签配置. 原型链污染检测规则使用两种类型的污染标签, 即 $T=\{t_{\text{input}}, t_{\text{proto}}\}$: 输入污染标签 t_{input}, 用于标记用户输入污染数据; 原型对象标签 t_{proto}, 用于标记与原型对象相关的数据.

● 污点源匹配. 原型链污染的漏洞特性, 决定了污点源的匹配方式根据分析目标而异: 对于第三方库, 可采取更保守的匹配方式, 将所有外部可访问方法 (如包含 export 关键字) 的参数标记为输入类型污点; 规则中不配置与原型对象相关的污点源模式, 即原型对象污染变量并非由直接匹配得到, 而是在污染传播追踪过程中进行转化.

● 污点传播追踪. 该阶段中, 算法配置如下的跨标签污染传播谓词:

$$\text{isCrossTag}(\langle v, t_{\text{input}}\rangle, \langle v', t_{\text{proto}}\rangle) \equiv \exists obj \in P,\ v' = \text{Read}(obj.v) \wedge \text{isVariable}(v) \tag{6}$$

即关注代码语句中的动态属性读取操作或等价形式, 当访问的对象属性名称为用户输入类型污点时, 将访问结果标记为原型对象类型污点, 实现跨标签污染传播. 例如图 7 中代码第 5 行的动态属性访问操作 curState[keys[i]], 当分析至该语句时, 污染变量集合中包含 ⟨keys[i], t_{input}⟩ 元素, 攻击者可将 keys[i] 的值设置为__proto__, 进而劫持 curState 的原型对象. 因此, 算法将根据如上的跨类型污点传播谓词, 将 ⟨curState[keys[i]], t_{proto}⟩ 加入污染变量集合中. 此处 curState 同样为用户输入类型污染变量, 但实际上算法对于属性访问的目标对象是否污染并无要求, 仅要求其原型对象为 Object.prototype.

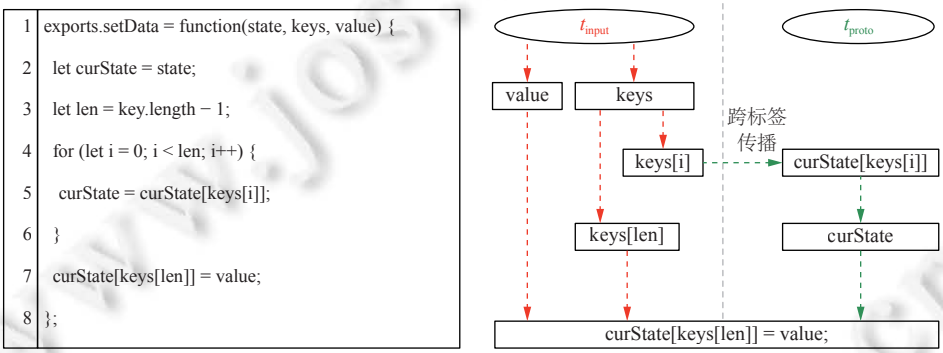


图 7 原型链污染漏洞检测过程

● 敏感操作检测. 原型链污染漏洞的约束条件谓词采取如下定义方式:

$$\text{isVuln}(\mathbb{V}_{\text{taint}}) \equiv \exists \langle obj, t_{\text{proto}}\rangle, \langle v, t_{\text{input}}\rangle \in \mathbb{V}_{\text{taint}}, \text{Write}(obj.v) \vee \text{Delete}(obj.v) \tag{7}$$

具体而言, 算法关注代码语句中的动态属性赋值操作 (如图 7 中第 7 行) 或动态属性删除操作 (如 delete curState[key]), 并基于当前语句处的污染变量集合信息进行判断, 若目标对象包含原型对象污染标签, 且目标属性包含用户输入污染标签, 证明攻击者可对 Object.prototype 产生实际影响, 此时将报告原型链污染漏洞.

3 MultiFlow 分析系统实现

基于第 2 节中介绍的两类多数据流污点分析算法及其应用场景, 本文实现了一个面向 Web 应用安全的静态漏洞检测原型系统 MultiFlow. 该系统采用 Java 和 TypeScript 作为开发语言, 支持对 Java Web/JavaScript Web 应用的安全漏洞静态检测, 并具备扩展至其他编程语言场景的潜力. MultiFlow 的整体工作流程如后文图 8 所示, 分为 3 个主要阶段: 基础分析、多数据流污点分析及漏洞报告生成.

3.1 系统输入

MultiFlow 以待分析程序和漏洞规则为输入, 采取与现有 Web 应用静态分析工具[6,9]类似的漏洞规则配置方式, 支持为目标漏洞指定三元组信息, 以及污点传播条件等其他常见安全相关方法配置. 此外, MultiFlow 在规则中

扩展了与多数据流分析方法相关的配置项, 进一步提升规则可扩展性. 具体而言, 对于多段数据流分析规则, 可配置应用中涉及的共享数据访问操作 (包括存储与读取两类操作模式); 对于多标签数据流分析规则, 可声明污染标签集合, 并为污点源、污点汇和净化函数模式关联特定标签, 或声明跨标签传播、敏感操作判断等自定义条件谓词.

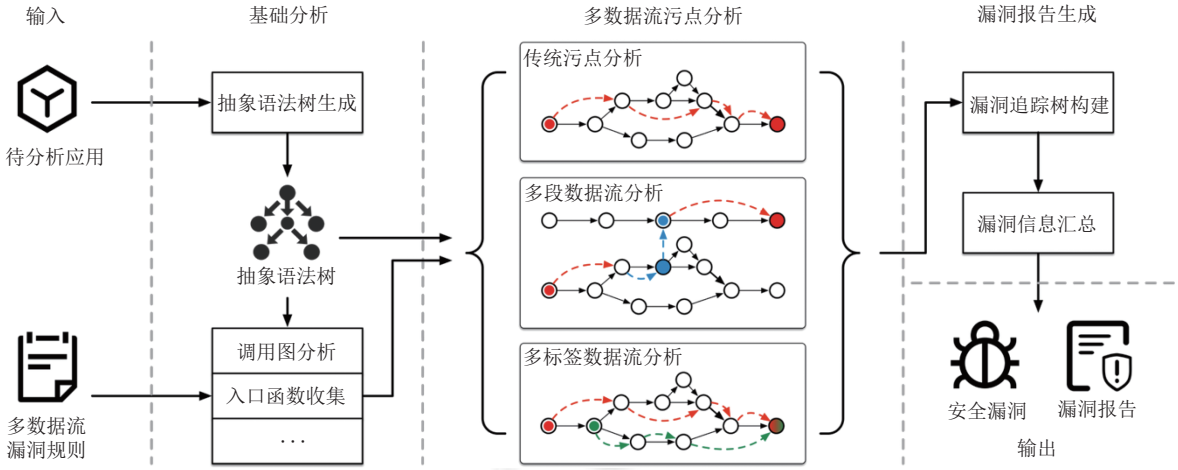


图 8 MultiFlow 工作流程

3.2 基础分析

该阶段主要完成漏洞检测的前期准备工作, MultiFlow 首先为应用程序代码生成抽象语法树, 并基于此构建调用图、初始化全局辅助数据结构, 从而为后续跨函数分析提供支持. 如第 1.2 节中所述, 作为 Web 应用程序分析中的一个重要概念, 入口函数是连接外部请求与应用内部业务逻辑的关键节点. 考虑到攻击者在利用安全漏洞时必然通过入口函数触发漏洞相关代码, MultiFlow 在基础分析阶段将综合考虑待分析程序的编程语言、开发框架等信息, 收集程序中的所有入口函数, 并以此为起点执行后续污点分析, 避免报告外部不可达或无法实际利用的安全漏洞.

3.3 多数据流污点分析

该阶段为 MultiFlow 的核心构成部分, 集成了支持域敏感、流敏感与上下文敏感的基础污点分析算法, 并扩展两种类型的多数据流污点分析能力. 在实际分析场景中, MultiFlow 将根据目标漏洞类型灵活调用相应的分析模块. 在污点分析算法的具体实现上, MultiFlow 针对如第 1.2 节中所述的 Web 应用安全特性, 采用基于抽象语法树的深度优先遍历方法, 并对循环语句和递归调用等代码结构进行单次展开, 忽略控制流中的回边. 该处理方式的核心思想是将程序控制流图转化为有向无环图 (directed acyclic graph, DAG), 从而避免传统迭代数据流分析的高复杂度问题, 可在检测精确度损失较小的前提下, 令分析复杂度与代码规模呈线性关系, 显著提升了整体分析效率. 此外, 考虑到传统静态污点分析不易覆盖现代 Web 应用中的诸多数据传播方式, MultiFlow 针对性建模了多种类型的细粒度污点传播策略, 例如 Java 线程类中 start 与 run 方法的跨过程数据流、各类字符串处理方法, JavaScript 中的原型修改、异步回调函数等. 在分析过程中, MultiFlow 使用污染传播图存储分析得到的中间信息, 其定义为有向图 $G=(N,E)$. 其中, 节点集 $N \subseteq \mathbb{V}_{\text{taint}}$ 用于记录分析到的污染变量, 边集 $E \subseteq \mathbb{V}_{\text{taint}} \times \mathbb{V}_{\text{taint}}$ 用于记录污点在程序中的传播路径. 基于污染传播图, MultiFlow 即可在后续漏洞报告生成阶段中, 构建可追溯的漏洞触发路径信息.

3.4 漏洞报告生成

该阶段中, 对于检测到的所有安全漏洞, MultiFlow 将汇总分析信息, 为漏洞生成详细报告, 包含漏洞代码具体位置、危险参数以及漏洞追踪树等关键信息, 帮助开发者快速定位问题并进行修复. 其中, 漏洞追踪树基于污染传播图进行构建. 具体而言, MultiFlow 在污染传播图中以漏洞触发点为起点, 执行反向深度优先遍历, 直至污点源变

量停止, 从而提取恶意输入的传播路径. 对于多段数据流分析结果, 漏洞追踪树将展示完整的跨控制流污染传播路径, 并标注共享数据写入与读取的关键节点; 对于多标签数据流分析结果, 漏洞追踪树将显示传播路径中各污染变量对应的污染标签, 以及跨标签的污染传播过程.

4 实验分析

为了全面评估 MultiFlow 在 Web 安全漏洞检测任务上的综合表现, 本节将探讨并回答如下 4 个研究问题.

● RQ1: 相较于现有方法, MultiFlow 的 Web 漏洞检测基础性能如何? 此问题将基于漏洞检测准召率指标, 综合对比 MultiFlow 与现有 Web 应用静态分析工具在不同类型 Web 漏洞检测任务上的整体能力表现.

● RQ2: MultiFlow 的多段数据流分析方法表现如何? 该问题旨在验证第 2.1.1 节中多段数据流分析方法的漏洞检测性能, 评估其是否能够有效地扩展传统污点分析的异步追踪能力, 实现对存储型 Web 漏洞的检测支持.

● RQ3: MultiFlow 的多标签数据流分析方法表现如何? 该问题旨在验证第 2.1.2 节中多标签数据流分析方法的漏洞检测性能, 评估其是否能够增强传统污点分析的语义表征能力, 实现对复杂类型 Web 漏洞检测支持.

● RQ4: MultiFlow 的漏洞检测效率表现如何? 该问题旨在验证 MultiFlow 及其核心方法是否能够高效地完成漏洞检测任务, 进而满足工业级规模 Web 应用场景下的分析需求.

4.1 实验设计

● 对比方法选取. 本文综合考虑学术界与工业界各类方法的先进性、分析场景支持能力与编程语言通用性, 选取以下几款具备代表性的 Web 静态应用安全测试工具作为比较对象.

(1) CodeQL[23]. CodeQL 是一款由 GitHub 研制的静态代码分析工具, 其支持多种编程语言, 且拥有较大规模的开源社区规则库, 已具备非常广泛的使用场景. CodeQL 的主要工作原理为将代码转化为可查询的数据库形式, 并使用 Datalog[24]模式的自定义查询语言 (QL) 构建相关规则, 挖掘代码缺陷或安全漏洞.

(2) Coverity[25]. Coverity 是一款基于高级静态分析技术的软件质量与安全检测工具, 其核心功能聚焦于源代码层级的缺陷检测与安全漏洞识别. 该工具通过布尔可满足性验证算法, 对 C、C++、Java 等多种语言代码进行深度分析, 精准定位内存泄漏、空指针解引用、并发竞争条件及输入验证漏洞等复杂问题.

(3) Tai-e[3]. Tai-e 是一款面向 Java 应用的静态程序分析框架, 其系统整合了 Soot、WALA、Doop 和 SpotBugs 等经典框架的最佳设计理念, 具备高效性、易用性和高度可扩展性. Tai-e 的核心设计包含基于创新的中间表示形式、强大的指针分析框架和可扩展的分析插件系统, 支持指针分析、污点分析和控制流/数据流分析等多种静态分析技术, 可有效识别程序中的缺陷和安全漏洞.

● 数据集构建. 为充分比较 MultiFlow 和现有方法在 Web 安全漏洞检测任务上的性能表现, 本文选取了知名度较高的 Java Web 安全漏洞测试用例集合 OWASP benchmark[26]作为基准漏洞数据集, 该数据集中共包含 2 740 个漏洞测试用例 (1 415 个正样本、1 325 个负样本), 涉及 SQL 注入、XSS、远程代码执行等 11 种常见 Web 安全漏洞类型, 可有效地测试各类 Web 应用静态分析工具的漏洞检测能力. 此外, 为评估 MultiFlow 中两种多数据流分析方法的性能表现, 本文在 GitHub[27]、NPM[28]等开源代码仓库, 以及来自大型企业的闭源工业级应用程序中, 选取具备一定规模与流行度的 Web 应用构建真实应用数据集, 共包含 30 个 Java Web 项目、10 个 JavaScript Web 项目和 20 个 JavaScript 第三方组件.

● 实验环境. 下文所有实验评估均在一台配备 Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50 GHz 处理器 (75 核), 256 GB 内存的 Ubuntu 20.04.6 LTS 服务器上运行. 在实验评估中, 设置最大可用线程数为 8, 设置 Java 堆内存上限为 6 GB, 使用的各工具版本分别为 CodeQL cli v2.19.0, Coverity 2024.12.1 和 Tai-e 0.5.1.

● 评价指标. 在 Web 安全漏洞检测任务中, 对于单个检测结果通常可分类为如下 4 种形式之一: (1) 真阳性 (true positive, *TP*), 指检测结果为存在漏洞、实际存在漏洞的样例; (2) 假阳性 (false positive, *FP*), 指检测结果为存在漏洞、实际无漏洞的样例; (3) 假阴性 (false negative, *FN*), 指检测结果为无漏洞、实际存在漏洞的样例; (4) 真阴性 (true negative, TN), 指检测结果为无漏洞、实际无漏洞的样例. 本文在评估 MultiFlow 和其他方法的漏洞检测

能力时, 使用相关工作中常见的几类评估指标: 精确率 (*Precision*)、召回率 (*Recall*) 和 *F*1 分数 (*F*1-*score*). 其中, 精确率表示检测出的漏洞中实际为漏洞的比例, 反映了检测结果的精确性; 召回率表示所有真实漏洞中被正确检测出的比例, 衡量了检测系统的覆盖能力; *F*1 分数是精确率和召回率的调和平均数, 用于综合评估检测性能, 平衡精确率和覆盖能力. 基于检测结果分类的 3 类评估指标计算方式如下所示:

$$Precision=\frac{TP}{TP+FP},\ Recall=\frac{TP}{TP+FN},\ F1\text{-}score=\frac{2\times Precision\times Recall}{Precision+Recall} \tag{8}$$

4.2 RQ1: MultiFlow 基础分析性能评估

为了验证 MultiFlow 对基础 Web 安全漏洞的检测能力, 本节将使用基准漏洞数据集开展实验评估, 比较 MultiFlow 和现有方法针对不同类型 Web 漏洞的检测效果. 评估过程中使用如下测试配置.

● 在评估 MultiFlow 时, 为各个漏洞类型独立编写其对应的基础污点分析规则, 其中包含基准漏洞数据集测试用例涉及的所有污点源、污点汇与净化函数模式; 规则中关闭所有多数据流分析相关配置, 即不包含共享数据存储/读取模式, 且仅为所有污点源指定单一类型的污染标签.

● 在评估 Coverity 与 CodeQL 时, 分别基于其官方规则库[29,30]中对应类型的漏洞规则, 使用默认参数配置运行分析; 在评估 Tai-e 时, 基于其官方代码库中提供的常见污点分析配置[31]构建完整漏洞规则, 并补充基准漏洞数据集涉及的污点源与污点汇模式, 以保障规则覆盖能力与其他方法一致; 在运行参数上, 使用默认的非上下文敏感模式. 各方法使用的具体运行命令与参数配置见表 1.

表 1 对比方法运行命令与参数设置

方法名称	运行命令与参数
Coverity	cov-analyze --dir /path/to/benchmark --all --max-mem 1024 --include-java --fb-max-mem 6144 --android-security -j4 --enable-audit-checkers --webapp-security --enable DC.STRING_BUFFER --enable ENUM_AS_BOOLEAN --enable HARDCODED_CREDENTIALS --enable ORM_LOST_UPDATE --enable UNENCRYPTED_SENSITIVE_DATA --enable FLOATING_POINT_EQUALITY --enable USER_POINTER --enable WEAK_GUARD --enable WEAK_PASSWORD_HASH --enable XML_INJECTION
CodeQL	codeql database analyze /path/to/database ql/java/ql/src/Security/CWE/path/to/rule.ql --format=csv --output=result.csv
Tai-e	java -jar tai-e-all-0.5.1.jar -cp /path/to/benchmark/target/classes -acp/ path/to/benchmark/target/classes -java 8 -ap -apta=taint-config:/path/to/rules.yml;only-app:false;dump:false;cs:ci;timelimit:1200000

基准漏洞数据集的测试结果对比如表 2 所示. 在漏洞检测精确率上, 得益于精细化的污染传播建模, MultiFlow 取得了 90.52% 的出色表现, 相较于 Coverity、CodeQL 和 Tai-e 分别提高了 13.11%、12.15% 和 30.18%; 在漏洞检测召回率上, 除 Tai-e 外, 各工具表现较为接近, MultiFlow 略优于 Coverity 和 CodeQL; 考虑 *F*1 分数时, MultiFlow 相较于其他工具同样表现最佳. 需说明的是, Tai-e 作为一款学术研究领域的程序分析框架, 在污点分析规则库成熟度上与 Coverity、CodeQL 等工具差距较大, 缺少对真实应用场景中常见污点传播模式的覆盖能力, 影响了其整体表现. 综合而言, 上述结果表明, MultiFlow 作为一种 Web 安全漏洞检测方法, 对于传统污点分析算法具备较强的兼容能力, 在检测准召率上表现优秀, 具备较强的实用性.

表 2 基准漏洞数据集测试结果对比

漏洞类型	CWE编号	Coverity			CodeQL			Tai-e			MultiFlow		
		TP	*FP*	*FN*	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*
path traversal	CWE-22	133	75	0	133	66	0	109	92	24	133	22	0
command injection	CWE-78	115	67	11	83	29	43	98	80	28	126	20	0
XSS	CWE-79	246	68	0	246	90	0	194	87	52	246	48	0
SQL injection	CWE-89	272	128	0	272	87	0	193	121	79	272	36	0
LDAP injection	CWE-90	27	15	0	27	13	0	19	22	8	27	4	0
weak cryptography	CWE-327	130	0	0	130	27	0	—	—	—	130	0	0
weak hashing	CWE-328	89	0	40	—	—	—	—	—	—	89	0	40
weak randomness	CWE-339	218	0	0	218	0	0	—	—	—	218	0	0

表 2 基准漏洞数据集测试结果对比 (续)

漏洞类型	CWE编号	Coverity			CodeQL			Tai-e			MultiFlow		
		TP	*FP*	*FN*	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*
trust boundary violation	CWE-501	83	30	0	83	24	0	49	29	34	83	12	0
secure cookie flag	CWE-614	36	0	0	36	0	0	—	—	—	36	0	0
XPATH injection	CWE-643	15	15	0	15	7	0	15	14	0	15	2	0
合计		1 364	398	51	1 243	343	43	860	698	42	1 375	144	40
精确率 (%)/召回率 (%)		77.41/96.40			78.37/96.66			60.34/75.06			90.52/97.17		
*F*1分数 (%)		85.87			86.56			66.90			93.73		

注: “—”表示对比工具 (Coverity、CodeQL、Tai-e) 的官方规则库未支持当前漏洞类型的检测, 故未产生相关检测数据

4.3 RQ2: 多段数据流分析方法评估

本节基于真实应用数据集中的 Java Web 项目, 对 MultiFlow 的多段数据流分析方法的漏洞检测能力开展评估. 在漏洞规则配置上, 为评估算法 1 对传统污点分析产生的增益效果, 本文以如下两种方式配置 MultiFlow 的分析漏洞规则集.

(1) 基础规则集 R_{base}: 与第 4.2 节相似, 配置各类常见 Web 漏洞的传统污点分析规则.

(2) 多段数据流规则集 R_{all}: 在 R_{base} 基础上, 增加与 Mybatis Mapper 框架相关的共享数据操作配置.

在评估漏洞检测结果的准确性时, 遵循如下步骤: 对于 MultiFlow 检测到的所有漏洞, 将由 3 位安全专家对漏洞报告独立进行分析, 标注是否为真实漏洞; 之后, 将对所有专家的漏洞标注结果执行交叉验证, 若存在不一致情况则引入第 4 位安全专家进行额外讨论, 确定该漏洞的最终标注结果.

30 个 Java Web 项目的分析结果如表 3 所示, 在使用基本漏洞规则 R_{base} 时, MultiFlow 共检测到 21 例安全漏洞, 总体检测精确率为 95.24%; 在添加共享数据相关规则配置后, MultiFlow 即可额外检测到 39 例数据库存储型 Web 漏洞, 其具体类型包括 XSS、SSRF、远程代码执行和 URL 重定向 (redirect) 漏洞, 漏洞覆盖能力得到极大提升, 但在检测精确率上略有下降 (约为 87.18%). 这些结果充分证明, 本文提出的多段数据流分析方法可有效地扩展传统污点分析的漏洞覆盖场景, 同时具备较好的泛用性.

表 3 MultiFlow 多段数据流分析测试结果

漏洞类型	MultiFlow × R_{base}				MultiFlow × R_{all}			
	检测总数	*TP*	*FP*	*Precision* (%)	检测总数 (Δ)	*TP* (Δ)	*FP* (Δ)	*Precision* (%)
XSS	9	9	0	100.00	23 (14)	21 (12)	2 (2)	91.30
SSRF	10	9	1	90.00	29 (19)	26 (17)	3 (2)	89.66
RCE	0	0	0	—	1 (1)	1 (1)	0 (0)	100.00
URL redirect	2	2	0	100.00	7 (5)	6 (4)	1 (1)	85.71
合计	21	20	1	95.24	60 (39)	54 (34)	6 (5)	90.00

注: “—”表示在使用基础漏洞规则集时未检测到远程代码执行漏洞, 故相关指标不适用

● 误报分析. 对于 MultiFlow 的 6 例误报, 其主要原因在于域敏感分析实现上的精确度局限性. 如第 2.1.1 节所述, MultiFlow 在获取共享数据的详细存储位置, 以及从共享数据读取操作中恢复污染变量时, 均依赖于细粒度域敏感分析. 若污染变量为 Java 容器类或数组等代码元素, 在后续分析中可能出现动态键值访问 (即键值索引在运行时通过动态表达式生成) 的情况, MultiFlow 难以精确建模此类非确定性键值的传播路径, 采取整体污染的保守策略, 导致错误扩大污染范围产生误报. 未来可基于常量传播、符号执行等技术, 进一步增强对键值的静态推断能力, 提升分析精确率.

4.4 RQ3: 多标签数据流分析方法评估

本节将从越权访问和原型链污染两类漏洞检测任务出发, 评估第 2.1.2 节所述多标签数据流方法在真实应用

中的漏洞检测能力与效率表现.

4.4.1 越权访问漏洞检测

如第 2.2.2 节所述, MultiFlow 越权访问漏洞检测算法中的核心概念为基于可信源与用户输入的鉴权操作识别, 故本节在评估 MultiFlow 所使用多标签数据流分析方法对于越权访问漏洞的检测性能时, 将同时从鉴权操作识别和漏洞检测两个维度出发, 讨论 MultiFlow 的性能表现.

● 鉴权操作识别. 在 30 个 Java Web 应用中, MultiFlow 共识别到 13 457 处鉴权操作, 包含 11 217 处垂直鉴权与 2 240 处水平鉴权. 为评估识别效果, 本文从所有结果中抽样 5% 进行人工分析, 判断其中的真实鉴权逻辑样本, 将真实样本占总样本的比例作为鉴权操作识别精确率指标. 详细分析结果如表 4 所示, MultiFlow 在鉴权操作识别上取得了 85.29% 的总体精确率表现, 且对于水平鉴权的识别精确率高于垂直鉴权. 这表明在越权漏洞检测任务中, 基于可信源的多标签数据流分析算法可有效地识别代码中的鉴权相关操作.

表 4 MultiFlow 鉴权操作识别结果

鉴权操作类型	识别总数	样本数	真实样本数	*Precision* (%)
垂直鉴权I	10 757	538	454	84.39
垂直鉴权II	460	23	19	82.61
水平鉴权I	1 592	79	73	92.41
水平鉴权II	643	32	27	84.38
水平鉴权III	5	1	1	100
合计	13 457	673	574	85.29

● 越权漏洞检测. 表 5 展示了 MultiFlow 的越权漏洞扫描结果. MultiFlow 在 30 个 Java Web 应用中共检测出 31 例垂直越权漏洞和 9 例水平越权漏洞, 总体精确率为 75%. 其中, 10 例误报的详细归因分析如下.

表 5 MultiFlow 越权漏洞检测结果

漏洞类型	检测数量	*TP*	*FP*	*Precision* (%)
垂直越权	31	23	8	74.19
水平越权	9	7	2	77.78
合计	40	30	10	75.00

● 垂直越权误报 (8 例). 均来自对业务允许的非敏感资源或公开数据进行增删改 (如日志数据库的写入/插入等). 这类操作风险较低、无须鉴权, 但“是否需要鉴权”的判断依赖业务逻辑语义与设计文档等外部信息; MultiFlow 目前仅基于程序代码开展分析, 难以准确判断, 因而将其误判为需鉴权的敏感操作. 未来工作中, 可引入基于 LLM 的语义识别, 对操作的敏感性与鉴权需求进行筛选 (结合命名、注释、调用上下文及外部文档), 以提升检测精确率.

● 水平越权误报 (2 例). 由鉴权操作识别遗漏导致. 如第 2.2.2 节所述, 除在业务代码中显式校验外, 鉴权还可能通过中间件、拦截器、注解/AOP、统一网关等机制在非显式控制流中完成. 由于缺乏对相应鉴权模式的充分覆盖, 以及复杂隐式控制流关系的解析能力, MultiFlow 在分析敏感操作时将相关路径误判为“未鉴权”, 从而产生误报. 后续工作中, MultiFlow 将完善鉴权模式库与各类框架/中间件的适配, 强化对跨层调用与集中式鉴权 (如网关、全局拦截器) 的建模与解析, 降低鉴权漏识率并提升精确率.

4.4.2 原型链污染漏洞检测

本节基于真实应用数据集中的 JavaScript Web 项目和开源第三方组件开展实验评估. 考虑到 CodeQL 官方规则库中包含原型链污染漏洞相关规则, 此处将额外对比 MultiFlow 与 CodeQL 的漏洞检测能力. 在 MultiFlow 的漏洞规则配置上, 当分析目标为 Web 项目时, 使用基本的用户输入污点源配置; 分析目标为第三方组件时, 则额外将所有的外部可访问 API 参数配置为用户输入污点源. 在对检测结果进行验证时, 遵循与第 4.3 节相同的分析步骤.

MultiFlow 和 CodeQL 的漏洞检测结果总览如表 6 所示, 在 20 个 JavaScript 第三方组件和 10 个 JavaScript Web 应用中, MultiFlow 共检测到 43 例原型链污染漏洞, 总体检测精确率为 83.72%; CodeQL 共检测到 20 例原型链污染漏洞, 精确率为 85.00%. 可以发现, 在二者的漏洞检测精确率接近的情况下, MultiFlow 在检测召回率上具备显著优势, 相较于 CodeQL 可检测到更多的原型链污染漏洞. 对于数据集中的 20 个开源第三方组件, MultiFlow 检测到的 15 例漏洞内共包含 3 例已知漏洞, 以及 12 例未知 0-day 漏洞. 本文已向 CVE 官方报告了所有未知漏洞, 截至目前共收到 8 个 CVE 编号, 具体信息如表 7 所示. 这一结果有力地证明了 MultiFlow 在真实场景下的 Web 漏洞检测能力.

表 6　原型链污染漏洞检测结果对比

数据集分类	MultiFlow				CodeQL			
	检测总数	*TP*	*FP*	*Precision* (%)	检测总数	*TP*	*FP*	*Precision* (%)
20 NPM packages	21	15	6	71.43	8	6	2	75.00
10 JavaScript Apps	22	21	1	95.45	12	11	1	91.67
合计	43	36	7	83.72	20	17	3	85.00

表 7　20 个开源第三方组件中 MultiFlow 获得的 CVE 编号

组件名称	组件版本	漏洞CVE编号	组件名称	组件版本	漏洞CVE编号
apidoc-core	0.15.0	CVE-2025-57**7	midway	1.0.0	CVE-2025-57**2
dref	0.1.2	CVE-2025-26**8	rollbar	2.26.4	CVE-2025-57**5
fast-redact	3.5.0	CVE-2025-57**9	sassdoc-extras	2.5.1	CVE-2025-57**6
json-schema-editor-visual	1.1.1	CVE-2025-57**0	spmrc	1.2.0	CVE-2025-57**7

● 误报分析. MultiFlow 的 7 例误报主要产生原因在于, 其污点传播追踪阶段存在对间接语义约束的分析缺失. 例如图 9(a) 展示的代码片段中, 变量 groupId 来源于用户输入参数, 但字符串拼接逻辑决定其无法被控制为 __proto__或 constructor.prototype 等危险属性, 不可用于劫持原型. 而 MultiFlow 难以判断该复杂约束, 将 style[groupId] 错误标记为 t_{proto} 类型污染变量, 进而在第 6 行赋值语句处产生误报; CodeQL 基于其声明式查询设计可实现更精细的污染净化逻辑, 在此类案例上表现优于 MultiFlow.

```
1  const GROUP_PREFIX = "group_";
   ...
2  app.post("/style", (req, res) => {
3  const style = ... // javascript object
4  const {id, name, key, value} = req.body;
5  const groupId = GROUP_PREFIX + name + "_" + id;
6  style[groupId][key] = value;
```

(a)

```
1  app.get("/", (req, res) => {
2  let {taint, key, val} = req.query.data;
   ...
3  let object = {};
4  object[taint].foo = "bar";
5  object[taint][key] = value;
```

(b)

图 9　简化后原型链污染漏洞误报示例

● 交集分析. 本文对 MultiFlow 与 CodeQL 检出的漏洞样本进行了详细对比分析, 漏洞检测结果的具体分布关系如图 10 所示, 二者共同识别出的漏洞数量为 9 例, 其中包含 1 例误报. 此外, 对于前述的 15 例开源第三方组件漏洞, CodeQL 仅能检出 4 例, 包含 2 例已知漏洞和 2 例未知漏洞. 考虑 CodeQL 检出而 MultiFlow 未检出的 9 例漏洞, 分析发现其产生原因为二者对漏洞品质要求的差异性: CodeQL 的检测规则中, 仅考虑原型对象 Object.prototype 是否被污染, 而未关注攻击者是否能够任意控制污染属性名, 例如图 9(b) 中, 第 4 行的赋值语句仅能固定影响原型对象的 foo 属性, 但 CodeQL 仍将其视为漏洞进行报告. 此类操作实际上利用条件较为苛刻、漏洞品质低, 在 MultiFlow 的检测规则中被视为非漏洞. 这进一步表明了 MultiFlow 相较于 CodeQL 拥有更强的原型链污染漏洞检测能力. 综上所述, 可认为 MultiFlow 的多标签数据流分析方法在原型链污染漏洞检测任务中具备良好的有效性, 检测性能表现优于 CodeQL 对应漏洞规则实现.

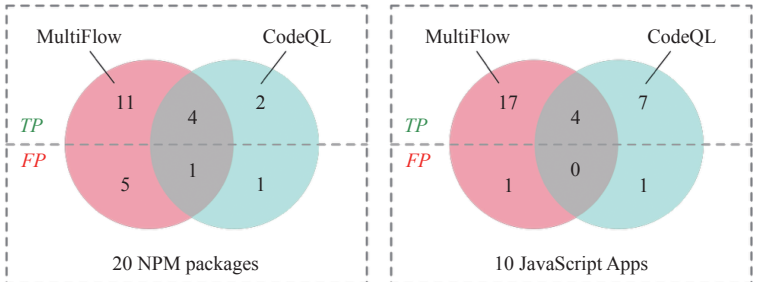


图 10 MultiFlow 与 CodeQL 检测漏洞交集关系

4.5 RQ4: MultiFlow 分析效率评估

本节评估 MultiFlow 在不同类型 Web 漏洞检测场景下的分析效率表现.

4.5.1 基础分析效率

本节将比较 MultiFlow 与 3 种现有方法在 Web 漏洞检测任务中的基础分析效率. 具体步骤为基于第 4.2 节所述各方法的漏洞规则集与运行参数不变, 对扫描流程重复执行 5 次统计总运行时间, 取平均值后得到各工具的平均分析时间开销. 在运行时间统计标准上, 对于 MultiFlow、Coverity 与 Tai-e, 统计其端到端扫描过程的总用时; 考虑到 CodeQL 的分析对象为基于程序代码预先构建的数据库, 故分别统计其数据库构建和漏洞规则扫描两部分用时, 合并作为总时间开销.

各个方法的分析总用时统计数据如表 8 所示. 得益于采用抽象语法树遍历的数据流分析实现方式, MultiFlow 在分析效率上均存在显著优势, 相较于 CodeQL 和 Tai-e 的提速比分别为 2.56 和 3.53 倍, 相较于 Coverity 的提速比则高达 19.98 倍. Coverity 在扫描前需要首先完成代码编译操作, 同时在工具设计上更偏向于通用化的分析模式, 这些因素共同决定了其在 Web 应用场景下高昂的分析开销; 对于 CodeQL 与 Tai-e, 影响其分析开销的核心因素在于, 其分析所需数据的全量预构建步骤 (如 CodeQL 的代码数据库构建、Tai-e 的中间表示 IR 生成与全局指针分析) 均需要较长时间完成, 对于大型项目而言尤其如此; MultiFlow 选择基于抽象语法树的分析策略, 预构建步骤开销较小. 综合而言, 相较于现有漏洞检测方法, MultiFlow 可更加高效地完成漏洞检测分析任务.

表 8 OWASP benchmark 分析效率对比

代码行总数	分析总用时 (s)			
	Coverity	CodeQL	Tai-e	MultiFlow
191 612	1 680.01	215.17	296.74	84.10

考虑 MultiFlow 的分析时间与程序代码规模的关系, 计算可知, 在基准漏洞数据集上分析用时平均值为 0.439 ms/LOC (每代码行 0.439 ms); 后续实验中将继续使用该指标, 评估 MultiFlow 使用的两类多数据流分析方法在分析效率上的表现.

4.5.2 多数据流方法分析效率

本节评估多标签数据流分析方法在两种应用场景下的分析效率表现, 评估时仍遵循如第 4.3、4.4 节所述实验步骤不变, 运行时间统计标准与第 4.5.1 节相同. 在评估原型链污染漏洞检测场景时, 考虑到 MultiFlow 针对 JavaScript 项目与第三方组件的漏洞规则存在一定的差异, 运行时间将基于不同应用分组进行统计; 此外, 将同时评估 CodeQL 的分析效率进行对比.

各个扫描任务的分析时间开销详细数据如表 9 所示. 在数据库存储型漏洞检测任务中, MultiFlow 使用的多段数据流分析方法平均每代码行分析用时为 0.466 ms, 相较基础污点分析存在 6.08% 的额外开销, 这部分额外开销主要来源于算法 1 中与共享数据相关的迭代分析过程, 以及二者的分析目标差异 (前者为基准用例, 后者为工业级真实 Web 应用). 这证明了 MultiFlow 的多段数据流分析方法能够在不产生过多额外开销的前提下, 将分析能力扩展至异步过程场景, 满足工业界真实场景下的轻量级 Web 漏洞扫描需求.

表 9 MultiFlow 多数据流分析效率统计

漏洞类型	应用分组	代码行总数	分析总用时 (s)		每代码行用时 (ms)	
			MultiFlow	CodeQL	MultiFlow	CodeQL
数据库存储型	30 Java Apps	2 253 984	1 049.48	—	0.466	—
越权访问	30 Java Apps	2 253 984	1 029.30	—	0.457	—
原型链污染	20 NPM packages	121 576	62.47	154.10	0.514	1.268
	10 JavaScript Apps	1 314 020	533.362	606.87	0.406	0.462

注: “—”表示在数据库存储型与越权访问漏洞检测任务的实验评估中未使用CodeQL工具进行对比实验, 故无相关分析用时数据

考虑多标签数据流分析方法, MultiFlow 在分析效率上同样取得了优异表现: 在越权访问检测任务中, MultiFlow 分析耗时为 0.457 ms/LOC, 相较于基础污点分析的额外开销为 4.04%, 主要与污点传播追踪过程中对可信源的处理相关; 在原型链污染检测任务中, MultiFlow 对于 JavaScript 项目和第三方组件的分析效率存在一定差异, 用时分别为 0.514 ms/LOC 和 0.406 ms/LOC, 这主要来源于二者的漏洞规则配置差异, 即 MultiFlow 针对第三方组件的分析规则中配置了更多的用户输入污点源模式, 分析过程中相应地涉及更多的污染传播步骤; 相较于 CodeQL, MultiFlow 实现了更高的分析效率, 同一应用分组下效率可提升 1.13–2.46 倍. 此外可以发现, 虽然 Java 与 JavaScript 的语言特性存在较大差异, 但得益于 MultiFlow 采用了基于抽象语法树遍历的分析方式, 分析效率上较为接近. 综合上述结果, 可认为 MultiFlow 的多标签数据流分析方法能够以较小的额外开销, 实现对 Web 应用中不同语义数据的细粒度表征, 有效提升工业级场景下 Web 漏洞检测能力.

5 讨 论

5.1 有效性威胁

本节主要讨论方法有效性的几个可能威胁.

● 评估数据集构成. 使用不同的评估数据集, 可能产生不同的检测结果. 本文在构建实验评估数据集时, 选取了若干具备代表性的工业界真实 Web 应用与第三方组件, 可认为基于这些应用的相关评估结论具备普遍性与泛用性.

● 分析效率评估结果. 考虑到服务器性能波动、漏洞扫描规则差异等因素, MultiFlow 的分析效率评估结果可能存在一定偏差; 本文在评估分析效率时, 采用多次运行扫描流程并取用时平均值的方式, 在一定程度上可客观反映不同方法的实际效率表现.

● 漏洞规则有效性. 对于多数据流污点分析方法, 其漏洞检测能力将受到规则中配置的准确性影响, 如存储型 Web 漏洞检测算法依赖于对应用中数据库框架的准确建模, 或是越权漏洞检测算法的效果依赖于规则中与可信源相关的污点源模式配置是否准确. 在实际使用过程中, 可通过框架建模、专家经验、LLM 辅助等方式, 识别相关的程序代码模式, 辅助提高多数据流分析算法的精确率.

5.2 算法复杂度分析

本节讨论两类多数据流分析算法以及 MultiFlow 的分析时间复杂度表现. 讨论时统一使用如下符号约定: 给定待分析程序 P 和漏洞规则, E 代表程序的控制流图规模, 具体而言可表示控制流图中的基本块数量或控制流边数; $S_{\text{source}} = |Source|$ 与 $S_{\text{sink}} = |Sink|$ 分别代表漏洞规则中污点源模式、污点汇模式的数量; V 代表程序控制流图中的总节点数, 可近似为程序中的语句数量, 满足 $V \sim E$ (一个控制流基本块对应单个或多个语句); V_t 代表分析结束后的污染节点 (语句) 数量, 在程序中总体呈稀疏分布, 满足 $V_t \ll V$.

5.2.1 传统污点分析

考虑图 2 中传统污点分析算法各个阶段的时间开销, 设每个污点源/污点汇模式的匹配开销为 O(1).

● 污点源匹配阶段. 需对所有程序语句节点判断其是否包含任一污点源, 匹配次数上界为 $S_{\text{source}}V$.

● 污点传播追踪阶段. 其总体时间开销与 E 和 V 相关 (下文使用 $\tau(E,V)$ 表示), 具体取值则由算法中采用的数据流

分析方法决定. 举例而言, 对于 IFDS 分析框架[32], 有 $\tau(E,V)\sim O(ED^3)$, 其中 D 代表数据流值域集合大小, 传统污点分析场景下满足 $D=2^V$; 对基于有限格 (lattice) 的数据流不动点迭代分析算法[33], 则有 $\tau(E,V)\sim O(Eh^2)\sim O(EV^2)$, 其中 h 为格的高度. 如第 3.3 节中所述, MultiFlow 采用基于抽象语法树遍历的分析策略, 忽略循环、递归等代码结构中的控制流回边, 将代码控制流处理为有向无环图结构, 在分析复杂度上可实现 $\tau(E,V)\sim O(E+V)$. 前述实验结果表明, 在 Web 应用安全领域中使用该分析策略, 可在不损失过多检测精确度的条件下显著提高分析效率, 适用于工业级大规模应用场景.

● 敏感操作检测阶段. 对所有污染节点执行污点汇模式匹配判断是否为敏感操作, 匹配次数上界为 $S_{\text{sink}}V_t$.

当规则规模相对固定时, S_{source} 和 S_{sink} 可视为常数, 且 $V_t\ll V$, 由此得到传统污点分析复杂度上界表示为:

$$O(S_{\text{source}}V+\tau(E,V)+S_{\text{sink}}V_t)=O(V+\tau(E,V)) \tag{9}$$

对于 MultiFlow, 代入得到其传统污点分析模式的时间复杂度上界为 $O(E+V)$.

5.2.2 多段数据流分析

对于多段数据流分析算法, 其主要的额外分析开销来源于针对共享数据存储/读取操作的迭代分析过程. 而迭代过程遵循增量分析思想, 不会重复分析先前已有结果, 故最终可在有限程序内实现收敛.

设算法迭代总轮次数为 n, 漏洞规则中配置的共享数据存储模式数量为 S_{store}, 以下详细推导其时间复杂度.

在第 i 轮迭代中, 使用的污点源模式数量为 s_i, 识别到污染可达的共享数据存储操作数量为 w_i. 考虑到共享数据存储操作在程序中的实际分布, 有 $w_{i-1}\ll V_t$; 每个轮次中的共享数据存储操作将被映射至共享数据读取模式, 并在去重后作为下一迭代轮次的污点源, 因此有:

$$\begin{cases} s_1=S_{\text{source}} \\ s_i\leqslant w_{i-1},\ 1<i\leqslant n \end{cases} \tag{10}$$

由此可以得到污点源匹配总次数上界满足:

$$\sum_{i=1}^{n}s_iV\leqslant\left(S_{\text{source}}+\sum_{i=1}^{n-1}w_i\right)V\ll(S_{\text{source}}+nV_t)V \tag{11}$$

对于污点传播追踪, 单个迭代轮次的时间开销仍为 $\tau(E,V)$. 此外, 在每个轮次迭代中, 将逐一验证该轮次的污染语句集合 V_{t_i} 是否为敏感操作或符合共享数据存储模式, 由 $V_t=\sum_{i=1}^{n}V_{t_i}$, 可知总匹配次数满足:

$$\sum_{i=1}^{n}(S_{\text{sink}}+S_{\text{store}})V_{t_i}<n(S_{\text{sink}}+S_{\text{store}})V_t \tag{12}$$

综合上述结果, 可得到 MultiFlow 多段数据流分析的时间复杂度表示:

$$O((S_{\text{source}}+nV_t)V+n\tau(E,V)+n(S_{\text{sink}}+S_{\text{store}})V_t)=O(V_t\cdot V+nE) \tag{13}$$

即算法开销与迭代轮次数 n 直接相关. 在实际分析中, n 通常为较小值, 即多段数据流分析算法可在常数轮次内实现快速收敛.

5.2.3 多标签数据流分析

对于多标签数据流分析算法, 其额外分析开销来源于将单一类型污点扩展到多类型污点后的污染变量集合维护, 以及增强的传播条件与漏洞条件谓词分析步骤.

设多标签数据流分析漏洞规则中配置的污染标签数量为 k, 其各个阶段时间复杂度推导过程如下.

● 污点源匹配阶段. 由公式 (1) 可知 $|\mathbb{S}_{\text{source}}|\leqslant kS_{\text{source}}$, 同第 5.2.1 节的分析, 得多标签污点源匹配次数上界为 $kS_{\text{source}}V$; 在实际分析场景中, 绝大多数污点源模式仅关联单个污染标签, 即 $|\mathbb{S}_{\text{source}}|\sim S_{\text{source}}$.

● 污点传播追踪阶段. 该阶段中, 若考虑对每条语句与不同污染标签均进行组合的极端情况, 则等价于为每个污染标签独立执行一次污点传播追踪, 总开销为 $k\tau(E,V)$; 实际分析场景中, 除跨类型污染传播谓词外, 不同污染标签均执行独立传播, 且程序中的多数污染语句或变量仅关联单个污染标签, 可认为污染传播步骤的分析开销仍为 $\tau(E,V)$.

● 敏感操作检测阶段. 由公式 (4) 显然有 $|\mathbb{S}_{\text{sink}}| = kS_{\text{sink}}$; 此外, 考虑污染传播行为映射满足分配律性质, 可将公式 (4) 基于多标签污点的敏感操作检测进行分解, 分别判断污染语句是否拥有漏洞谓词条件所需的各个污染标签. 极端情况下, 每个污点汇模式的谓词条件中均涉及所有 k 种类型的污染标签, 此时匹配次数上界为 $kS_{\text{sink}}V_t$.

综上所述, 多标签数据流分析的时间复杂度上界可表示为:

$$\mathrm{O}(kS_{\text{source}}V + k\tau(E,V) + kS_{\text{sink}}V_t) = \mathrm{O}(kV + k\tau(E,V)) \tag{14}$$

一般地, 实际分析场景中多标签数据流分析算法的复杂度为:

$$\mathrm{O}(S_{\text{source}}V + \tau(E,V) + kS_{\text{sink}}V_t) = \mathrm{O}(E+V) \tag{15}$$

即多标签数据流分析算法与传统污点分析算法开销相近.

6 相关工作

6.1 Web 静态应用安全测试

作为一种高效的漏洞检测方法, Web 应用静态安全测试技术始终是研究领域备受关注的热点课题. 早期技术主要依赖于对代码文本的简单模式匹配, 虽能快速发现浅层代码缺陷, 但存在极高误报率问题. 随着 Web 应用复杂度的提升, 众多研究者[34−40]开始引入污点分析这一关键技术, 将程序代码转化为抽象语法树、代码属性图[41]或静态单赋值 (static single assignment, SSA) 等中间表示形式后, 基于数据流分析追踪不可信输入的传播过程, 进而识别 Web 安全漏洞.

近年来, 基于静态污点分析的 Web 应用漏洞检测研究范式呈现出两个主要演进方向. 其一, 聚焦于整体分析精度与效率提升, 主要思路为在污点分析具体实现上融入各类形式化方法与工具, 从流敏感、域敏感、上下文敏感等分析能力维度增强静态分析技术, 进而提升漏洞检测精确率、召回率与整体分析效率. 例如, Tan 等人[3] 提出的 Tai-e 框架实现了针对 Java Web 应用的创新性高效指针分析算法, 可在降低复杂度的同时实现更准确的分析结果; Li 等人[4]提出了基于对象依赖图 (ODG) 和图查询的 Node.js 应用漏洞检测方法, 其检测过程结合了流敏感、上下文敏感的静态分析与混合分支敏感技术; 在此基础上, Kang 等人[5]针对 JavaScript 语言动态特性提出一种扩展抽象解释方法, 通过自底向上与自顶向下的两阶段处理, 平衡静态分析的准确性与可扩展性问题; Luo 等人[6]的 TChecker 框架针对 PHP 语言弱类型问题, 建模面向对象相关特性并执行对象的过程间数据流分析来细化类型, 实现 PHP 动态特性分析支持. 其二, 致力于特异性分析模型构建, 即针对 Web 应用开发过程中常见的框架、开发架构或特殊环境进行细粒度语义建模, 补全缺失的程序控制流、数据流信息, 提升静态分析技术在不同场景下的安全漏洞识别能力. 例如, Chen 等人[7]提出了一种针对 Spring 框架核心技术的静态分析方法, 通过优化调用图构建过程增强其完整性, 来提高对 Spring Web 应用中依赖注入、面向切面编程等行为的分析能力; Guo 等人[8]提出了组件图概念, 用于建模 JavaScript React 框架数据流以及跨组件数据依赖, 可实现对 React 应用漏洞的有效检测; Wang 等人[42]提出了 ANTaint, 通过扩展调用图并按需对库应用污点传播, 来提高方法在工业级场景下面对大量库、原生方法和企业级框架的可扩展性与准确性; Liu 等人[43]针对微服务架构 Web 应用中的漏洞检测问题, 提出了一种名为 MScan 的新型安全分析方法, 通过基于 LLM 的入口识别、服务依赖关系建模、距离引导的选择性污点分析策略等多项技术, 解决传统方法难以处理跨服务通信与调用关系的问题; Park 等人[44]提出的 SAFEWapp 框架可针对不同浏览器执行环境构建静态分析模型, 并支持调整 DOM 树抽象层次来平衡分析的性能与精确率.

上述工作虽然在实现层面有所差异, 但采取的核心污点分析算法均相似. 而本文提出的多数据流分析方法着重于在 Web 应用场景下改进污点分析算法的底层框架设计, 与这些工作呈正交关系, 且方法本身与 Web 应用程序的编程语言或开发框架实现无关, 具备较强的泛用性. 在实践中, 可将多数据流分析方法与上述各类分析框架相互结合, 共同提升 Web 应用静态安全测试技术的分析能力.

此外, 部分研究工作在关注特定场景下的静态分析问题时, 采用了定制化的数据流抽象层次与近似分析策略, 相较于通用分析方法可在准召率与性能间取得更好平衡, 如 Zhong 等人[45]针对工业微服务应用场景下的敏感数据泄露问题, 提出了可扩展组合静态污点分析工具 CFTaint, 其核心思路为采取对象非敏感 (object-insensitive) 的

字段级 (field-based) 分析策略, 通过合并同一类型的所有对象实例, 在不依赖完整调用图的前提下实现对跨过程敏感数据传播的精确追踪, 有效解决了工业微服务中敏感数据治理的效率与精确度矛盾; 本文则针对 Web 应用的诸多独有特性, 采用基于抽象语法树遍历的分析策略, 并忽略循环、递归等结构中的控制流回边. 评估结果表明, 该策略可在不影响 Web 漏洞检测精确度的前提下, 显著提升整体分析效率.

6.2 异步过程中的 Web 漏洞检测

如第 1.2 节中所述, 异步执行过程是 Web 应用的重要特性之一, 但目前的 Web 漏洞静态检测工作对于该特性的考虑较少; 部分研究者采用建模思想连接部分污染数据流断边, 处理存储型漏洞的检测问题. 例如, Dahse 等人[11]提出了针对 PHP Web 应用的持久数据存储建模方法; Su 等人[10]提出的 Splendor 框架采用启发式的令牌模糊匹配方法, 定位数据库与源代码之间的污染数据流, 进而检测 PHP Web 应用中的存储型 XSS 漏洞; Balzarotti 等人[12]则关注 Web 应用中跨模块的漏洞检测问题, 核心思想为通过建模数据库操作, 分析模块间的交互关系. 上述研究并未建立异步过程 Web 漏洞检测的普适性方法论, 且使用的建模机制通常依赖于特定语言场景或框架, 迁移成本高, 难以实现简单推广.

在混合检测技术领域, 基于动静态混合分析方法的异步漏洞检测研究也取得一定进展. 如 Olsson 等人[46]提出了一种基于数据库的灰盒扫描方法 Spider-Scents, 其思路为在数据库中注入攻击载荷, 并在运行 Web 应用时自动查找未受保护的输出, 检测存储型 XSS 漏洞; Alhuzal 等人[47]提出的 Navex 框架则采用基于导航图 (navigation graph) 引导的符号执行技术, 在分析过程中对存在链接跳转关系的跨请求异步控制流实现连接. 这些方法同样面临特定语言环境约束、分析复杂度高等技术瓶颈, 难以有效适配 Web 静态应用安全测试场景的实际需求.

本文所提出的多段数据流分析方法, 其核心创新为基于共享数据概念的多轮次迭代分析思想, 分析能力并不依赖于特定的持久化介质类型, 能够以较小的分析开销, 系统性地实现对跨越多段异步执行过程的污染数据流追踪, 相较于现有工作具备更广泛的适用场景.

6.3 多标签污点分析技术

在二进制安全领域中, 已有诸多研究者[13−18]提出使用多标签污点分析 (multi-tag taint analysis) 技术来增强动态污点分析、符号执行等漏洞检测方法. 其实现形式通常为针对指定内存单元或寄存器设置污染标签位, 在运行时动态记录污染数据来源组合, 进而实现对多源信息流的实时追踪, 或优化符号约束计算. 考虑到 Web 应用安全更多关注应用层语义、无须精确追踪变量或内存具体取值的分析特性, 直接迁移上述方法将产生较大的分析开销, 无法满足工业级 Web 静态分析场景的实际需求.

在 Web 应用安全领域的静态分析相关工作中, 极少有研究者提及类似概念, 仅 TChecker[6]将污染标签用于表征不同漏洞类型, 实现对多种漏洞的并行扫描, 其污染标签设计本质上服务于漏洞分类标识, 而非反映输入源语义特征. 本文提出的多标签数据流分析算法目标场景为针对 Web 应用的安全漏洞静态检测, 将污染标签概念用于表示 Web 应用程序中不同来源的输入数据, 设计上考虑了 Web 应用的诸多特性, 同时通过跨标签污染传播、基于多标签的敏感操作检测等方式, 进一步提高了分析的灵活性, 同时可在分析性能开销上取得优异表现.

7 总 结

针对现有 SAST 技术受 Web 应用场景下传统污点分析算法局限性影响的问题, 本文提出一种基于面向 Web 安全漏洞检测的多数据流静态分析方法, 并基于该方法实现了适用于 Java Web 与 JavaScript Web 应用的漏洞检测原型系统 MultiFlow. 该系统通过多段数据流与多标签数据流两种类型的全新分析范式, 从多个维度扩展传统污点分析算法的漏洞检测能力与适用场景, 并针对 Web 应用独有特性进行多项优化. 实验结果表明, 在包含 60 个真实 Web 应用与第三方组件的数据集上, MultiFlow 具备良好的有效性, 可准确检测数据库存储型漏洞、越权访问、原型链污染等多种复杂类型 Web 安全漏洞; 在基准数据集上的评估结果表明, MultiFlow 的漏洞检测基础性能与分析效率表现均优于两类现有方法, 验证了其实用价值.

未来研究工作中, 计划将 MultiFlow 扩展至更多语言与漏洞类型, 进一步提升其泛用性; 此外, 还将关注基于

两类多数据流分析方法相结合的 Web 漏洞检测技术, 可在原型链污染后续利用等场景中实现应用; 目前的分析系统将多数据流漏洞规则作为输入, 而规则本身的编写仍需要较多的专家知识, 后续可进一步探索基于大语言模型的分析规则自动化生成技术, 以持续提升漏洞检测全流程的自动化能力.

References

[1] OWASP TOP 10: 2021. 2025. https://owasp.org/Top10/

[2] Static application security testing (SAST) software market size and forecast. 2023. https://www.verifiedmarketresearch.com/product/static-application-security-testing-sast-software-market/

[3] Tan T, Li Y. Tai-e: A developer-friendly static analysis framework for Java by harnessing the good designs of classics. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 1093–1105. [doi: 10.1145/3597926.3598120]

[4] Li S, Kang MQ, Hou JW, Cao YZ. Mining node.js vulnerabilities via object dependence graph and query. In: Proc. of the 31st USENIX Security Symp. Boston: USENIX, 2022. 143–160.

[5] Kang MQ, Xu YC, Li S, Gjomemo R, Hou JW, Venkatakrishnan VN, Cao YZ. Scaling JavaScript abstract interpretation to detect and exploit node.js taint-style vulnerability. In: Proc. of the 2023 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2023. 1059–1076. [doi: 10.1109/SP46215.2023.10179352]

[6] Luo CH, Li PH, Meng W. TChecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in PHP applications. In: Proc. of the 2022 ACM SIGSAC Conf. on Computer and Communications Security. Los Angeles: ACM, 2022. 2175–2188. [doi: 10.1145/3548606.3559391]

[7] Chen M, Tu TF, Zhang H, Wen QY, Wang WH. Jasmine: A static analysis framework for spring core technologies. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2022. 60. [doi: 10.1145/3551349.3556910]

[8] Guo ZY, Kang MQ, Venkatakrishnan VN, Gjomemo R, Cao YZ. ReactAppScan: Mining react application vulnerabilities via component graph. In: Proc. of the 2024 ACM SIGSAC Conf. on Computer and Communications Security. Salt Lake City: ACM, 2024. 585–599. [doi: 10.1145/3658644.3670331]

[9] Backes M, Rieck K, Skoruppa M, Stock B, Yamaguchi F. Efficient and flexible discovery of PHP application vulnerabilities. In: Proc. of the 2017 IEEE European Symp. on Security and Privacy. Paris: IEEE, 2017. 334–349. [doi: 10.1109/EuroSP.2017.14]

[10] Su H, Li F, Xu LL, Hu WB, Sun YJ, Sun Q, Chao HN, Huo W. Splendor: Static detection of stored XSS in modern Web applications. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 1043–1054. [doi: 10.1145/3597926.3598116]

[11] Dahse J, Holz T. Static detection of second-order vulnerabilities in Web applications. In: Proc. of the 23rd USENIX Conf. on Security Symp. San Diego: USENIX Association, 2014. 989–1003. [doi: 10.5555/2671225.2671288]

[12] Balzarotti D, Cova M, Felmetsger VV, Vigna G. Multi-module vulnerability analysis of Web-based applications. In: Proc. of the 14th ACM Conf. on Computer and Communications Security. Alexandria: ACM, 2007. 25–35. [doi: 10.1145/1315245.1315250]

[13] She DD, Chen YZ, Shah A, Ray B, Jana S. Neutaint: Efficient dynamic taint analysis with neural networks. In: Proc. of the 2020 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2020. 1527–1543. [doi: 10.1109/SP40000.2020.00022]

[14] Cui BJ, Wang FW, Guo T, Dong GW, Zhao B. FlowWalker: A fast and precise off-line taint analysis framework. In: Proc. of the 4th Int'l Conf. on Emerging Intelligent Data and Web Technologies. Xi'an: IEEE, 2013. 583–588. [doi: 10.1109/EIDWT.2013.105]

[15] Ming J, Wu DH, Xiao GY, Wang J, Liu P. TaintPipe: Pipelined symbolic taint analysis. In: Proc. of the 24th USENIX Conf. on Security Symp. Washington: USENIX Association, 2015. 65–80. [doi: 10.5555/2831143.2831148]

[16] Cui BJ, Wang FW, Guo T, Dong GW. A practical off-line taint analysis framework and its application in reverse engineering of file format. Computers & Security, 2015, 51: 1–15. [doi: 10.1016/j.cose.2015.02.006]

[17] Redini N, Machiry A, Das D, Fratantonio Y, Bianchi A, Gustafson E, Shoshitaishvili Y, Kruegel C, Vigna G. BootStomp: On the security of bootloaders in mobile devices. In: Proc. of the 26th USENIX Conf. on Security Symp. Vancouver: USENIX Association, 2017. 781–798. [doi: 10.5555/3241189.3241251]

[18] Zhang H, Chen WT, Hao Y, Li GR, Zhai YZ, Zou XC, Qian ZY. Statically discovering high-order taint style vulnerabilities in OS kernels. In: Proc. of the 2021 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2021. 811–824. [doi: 10.1145/3460120.3484798

[19] Wikipedia. Program analysis. 2025. https://en.wikipedia.org/wiki/Program_analysis

[20] Liskov B. A history of CLU. In: History of Programming Languages II. New York: ACM, 1996. 471–510. [doi: 10.1145/234286.1057826]

[21] Dan Book. perlsec—Perl security. 2025. https://perldoc.perl.org/perlsec
[22] MyBatis 3. 2025. https://mybatis.org/mybatis-3/
[23] CodeQL. 2025. https://codeql.github.com/
[24] Wikipedia. Datalog. 2025. https://en.wikipedia.org/wiki/Datalog
[25] BlackDuck. Coverity static analysis. 2025. https://www.blackduck.com/static-analysis-tools-sast/coverity.html
[26] OWASP Foundation. OWASP benchmark project. 2025. https://owasp.org/www-project-benchmark/
[27] GitHub. 2025. https://github.com/
[28] NPM. 2025. https://www.npmjs.com/
[29] BlackDuck. Coverity static analysis coverage for common weakness enumeration (CWE). 2025. https://www.blackduck.com/static-analysis-tools-sast/cwe.html
[30] GitHub. The CWEs supported by CodeQL. 2025. https://github.com/github/codeql/tree/main/java/ql/src/Security/CWE
[31] GitHub. Commonly used taint-config of Tai-e. 2025. https://github.com/pascal-lab/Tai-e/tree/master/src/main/resources/commonly-used-taint-config
[32] Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In: Proc. of the 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. San Francisco: ACM, 1995. 49–61. [doi: 10.1145/199448.199462]
[33] Aho AV, Lam MS, Sethi R, Ullman JD. Compilers: Principles, Techniques, & Tools. 2nd ed., Boston: Pearson, 2007.
[34] Liu JY, Han JX, Huang C. Vulnerability detection in source code using statice analysis. Journal of Cyber Security, 2022, 7(4): 100–113 (in Chinese with English abstract). [doi: 10.19363/J.cnki.cn10-1380/tn.2022.07.08]
[35] Wang L, Li F, Li L, Feng XB. Principle and practice of taint analysis. Ruan Jian Xue Bao/Journal of Software, 2017, 28(4): 860–882 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5190.htm [doi: 10.13328/j.cnki.jos.005190]
[36] Zhang J, Zhang C, Xuan JF, Xiong YF, Wang QX, Liang B, Li L, Dou WS, Chen ZB, Chen LQ, Cai Y. Recent progress in program analysis. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 80–109 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]
[37] Jovanovic N, Kruegel C, Kirda E. Static analysis for detecting taint-style vulnerabilities in Web applications. Journal of Computer Security, 2010, 18(5): 861–907. [doi: 10.5555/1841962.1841968]
[38] Jovanovic N, Kruegel C, Kirda E. Pixy: A static analysis tool for detecting Web application vulnerabilities. In: Proc. of the 2006 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2006. 258–263. [doi: 10.1109/SP.2006.29]
[39] Livshits VB, Lam MS. Finding security vulnerabilities in Java applications with static analysis. In: Proc. of the 14th Conf. on USENIX Security Symp. Baltimore: USENIX Association, 2005. 18. [doi: 10.5555/1251398.1251416]
[40] Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android Apps. ACM Sigplan Notices, 2014, 49(6): 259–269. [doi: 10.1145/2666356.2594299]
[41] Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. In: Proc. of the 2014 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2014. 590–604. [doi: 10.1109/SP.2014.44]
[42] Wang J, Wu YG, Zhou G, Yu YM, Guo ZY, Xiong YF. Scaling static taint analysis to industrial SOA applications: A case study at alibaba. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 1477–1486. [doi: 10.1145/3368089.3417059]
[43] Liu FY, Zhang Y, Chen T, Shi YK, Yang GL, Lin ZH, Yang M, He JY, Li Q. Detecting taint-style vulnerabilities in microservice-structured Web applications. In: Proc. of the 2025 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2025. 972–990. [doi: 10.1109/SP61157.2025.00137]
[44] Park C, Won S, Jin J, Ryu S. Static analysis of JavaScript Web applications in the wild via practical DOM modeling. In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering. Lincoln: IEEE, 2015. 552–562. [doi: 10.1109/ASE.2015.27]
[45] Zhong ZX, Liu JC, Wu DY, Di P, Sui Y, Liu AX, Lui JCS. Scalable compositional static taint analysis for sensitive data tracing on industrial micro-services. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice. Melbourne: IEEE, 2023. 110–121. [doi: 10.1109/ICSE-SEIP58684.2023.00015]
[46] Olsson E, Eriksson B, Doupé A, Sabelfeld A. Spider-scents: Grey-box database-aware Web scanning for stored XSS. In: Proc. of the 33rd USENIX Conf. on Security Symp. Philadelphia: USENIX Association, 2024. 377. [doi: 10.5555/3698900.3699277]
[47] Alhuzali A, Gjomemo R, Eshete B, Venkatakrishnan VN. NAVEX: Precise and scalable exploit generation for dynamic Web applications. In: Proc. of the 27th USENIX Conf. on Security Symp. Baltimore: USENIX Association, 2018. 377–392. [doi: 10.5555/3277203.3277232]

附中文参考文献

[34] 刘嘉勇, 韩家璇, 黄诚. 源代码漏洞静态分析技术. 信息安全学报, 2022, 7(4): 100–113. [doi: 10.19363/J.cnki.cn10-1380/tn.2022.07.08]

[35] 王蕾, 李丰, 李炼, 冯晓兵. 污点分析技术的原理和实践应用. 软件学报, 2017, 28(4): 860–882. http://www.jos.org.cn/1000-9825/5190.htm [doi: 10.13328/j.cnki.jos.005190]

[36] 张健, 张超, 玄跻峰, 熊英飞, 王千祥, 梁彬, 李炼, 窦文生, 陈振邦, 陈立前, 蔡彦. 程序分析研究进展. 软件学报, 2019, 30(1): 80–109. http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

作者简介

毛祥煜, 工程师, 主要研究领域为程序分析, 系统软件与安全, Web 应用安全.

肖庆, 博士, 主要研究领域为程序分析, 软件测试及分析, Web 应用安全.

戴嘉润, 博士, 主要研究领域为漏洞挖掘, AI 系统安全.

何君尧, 工程师, 主要研究领域为程序分析, 软件测试及分析, Web 应用安全.

谭杰, 硕士, 主要研究领域为生产网安全, 研发安全.