软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(9):3407−3434 [doi: 10.13328/j.cnki.jos.007600] [CSTR: 32375.14.jos.007600]

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

AWTaint: 面向 Web 应用漏洞检测的增量静态分析框架*

罗天涵[1], 肖 庆[1], 戴嘉润[2], 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 计算机科学技术学院, 上海 200437)

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

摘　要: 在 DevOps 持续集成实践中, Web 应用的高频代码迭代对传统静态分析工具提出了严峻挑战: 全量扫描模式导致计算资源浪费与分析延迟, 而现有增量分析技术因缺失对多样化漏洞检测能力, 且精度、效率与一致性间存在矛盾, 难以满足实际需求. 对此, 提出一种面向 Web 应用漏洞检测的增量静态分析框架 AWTaint. 该框架具备域敏感、上下文敏感与流敏感能力, 其利用函数摘要表征输入输出变量之间的映射关系, 产出与各类检测规则相关的污点传播信息. 该框架采用一种细粒度的增量计算方法, 首先利用调用图估算保守的增量变化范围, 其次利用函数摘要变化感知具体影响范围. 从而有效满足工业级 Web 应用漏洞检测对分析精度、计算效率与结果一致性的 3 项核心要求. 实验表明, 在包含 10 个真实 Java Web 应用的测试集上, AWTaint 可以支持多种 Web 应用漏洞检测需求, 其增量分析模式相较全量分析模式平均加速 3.63 倍, 内存峰值控制在 8 GB 以内, 且漏洞检测具有完全一致性. 该框架为安全左移实践提供了工程化解决方案, 在保障检测精度的前提下, 显著优化了资源利用率与开发迭代效率.

关键词: 静态分析; 污点分析; 增量分析; 函数摘要; Web 应用安全

中图法分类号: TP311

中文引用格式: 罗天涵, 肖庆, 戴嘉润, 谭杰. AWTaint: 面向Web应用漏洞检测的增量静态分析框架. 软件学报, 2026, 37(9): 3407–3434. http://www.jos.org.cn/1000-9825/7600.htm

英文引用格式: Luo TH, Xiao Q, Dai JR, Tan J. AWTaint: Incremental Static Analysis Framework for Vulnerability Detection in Web Applications. Ruan Jian Xue Bao/Journal of Software, 2026, 37(9): 3407–3434 (in Chinese). http://www.jos.org.cn/1000-9825/7600.htm

AWTaint: Incremental Static Analysis Framework for Vulnerability Detection in Web Applications

LUO Tian-Han[1], XIAO Qing[1], DAI Jia-Run[2], TAN Jie[1]

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

[2](School of Computer Science and Technology, Fudan University, Shanghai 200437, China)

Abstract: In DevOps continuous integration practices, the high-frequency code iteration of Web applications poses significant challenges to traditional static analysis tools: full analysis techniques cause computational resource waste and delays, while existing incremental analysis techniques struggle to meet practical requirements due to limitations in detecting diverse vulnerabilities and inherent trade-offs between accuracy, efficiency, and consistency. To address these challenges, this study proposes AWTaint, an incremental static analysis framework for Web application vulnerability detection. The framework features field-, context-, and flow-sensitivity, leveraging function summaries to characterize relationships between input and output variables, generating taint propagation information associated with various detection rules. A fine-grained incremental computation approach is adopted in this framework: first, a conservative incremental change scope is estimated through call graph analysis, and then function summary changes are utilized to determine impact ranges. This effectively satisfies the three core requirements of industrial Web application vulnerability detection: precision, efficiency, and consistency. Experimental results on a dataset containing 10 real-world Java Web applications demonstrate that AWTaint supports multiple Web

* 本文由“形式化方法与应用”专题特约编辑安杰副研究员、顾斌研究员、李建文教授推荐.
收稿时间: 2025-08-29; 修改时间: 2025-10-28; 采用时间: 2025-12-08; jos 在线出版时间: 2025-12-24
CNKI 网络首发时间: 2026-05-14

application vulnerability detection requirements. Compared to full analysis, its incremental analysis achieves an average speedup of 3.63×, with memory peak kept within 8 GB, while maintaining complete consistency in vulnerability detection results. This framework provides an engineering solution for shift-left security practices, significantly optimizing resource utilization and development iteration efficiency without compromising detection accuracy.

Key words: static analysis; taint analysis; incremental analysis; function summary; Web application security

互联网的迅猛发展在为人类社会带来数字化便利的同时, 也催生了日益严峻的网络安全威胁. CrowdStrike 在《2024 全球安全威胁报告》[1]中指出: 攻击者正在不断缩短从初始入侵、横向移动到数据泄露之间的时间, 网络攻击的速度和破坏性正持续加快. 在 OWASP Top 10 漏洞榜单中, 注入攻击、服务端请求伪造等 Web 应用漏洞持续存在, 凸显出应用层安全防护的迫切需求[2]. 历史上的重大安全事件已多次印证传统安全模式的局限性: 2013 年, 雅虎数据泄露事件因未及时修复已知漏洞, 导致 10 亿用户数据遭窃[3]; 2017 年, Equifax 数据泄露事件暴露其漏洞响应滞后, 最终影响 1.48 亿用户并引发大规模隐私泄露[4]; 而 2025 年, Bybit 交易所被盗事件因应用层安全缺陷, 造成超 14.6 亿美元资产损失[5]. 这些案例的共同特征在于漏洞均在系统上线后才被发现, 不仅导致高昂的修复成本, 更造成难以挽回的声誉损害.

在此背景下, 将安全检测环节前置到软件开发生命周期 (software development life cycle, SDLC) 的早期阶段, 已成为降低安全风险的核心策略. 这种“安全左移”理念要求在代码编写阶段即进行漏洞检测, 而静态应用安全测试 (static application security testing, SAST) 工具通过直接分析源代码, 可在不执行程序的情况下检测潜在安全风险, 这使其成为实现安全左移的关键技术手段. 通过在开发早期集成 SAST 工具, 开发团队能够在编码阶段发现并修复诸如注入攻击, 服务端请求伪造等常见漏洞, 从而显著降低后期修复成本与攻击风险. Gartner 研究显示, SAST 工具市场正以 14.2% 的年复合增长率扩张, 2023 年全球 SAST 工具市场规模已达 87 亿美元, 其中 Web 应用安全检测占比超过 62%[6]. 预计到 2028 年将有 40% 的 IT 服务合同强制嵌入自动化安全测试条款[7]. 这种行业趋势推动着开发团队普遍采用静态应用程序安全测试技术, 作为代码发布前的关键安全保障环节.

然而, 随着 DevOps[8]概念的普及, 现代 Web 应用的架构复杂性与持续交付模式对 SAST 工具提出了多重挑战. 一方面需要满足高频次全量扫描带来的算力需求, 另一方面需要应对代码版本间微小差异带来的分析冗余问题. 然而, 在漏洞相关的语义未发生代码变更或不受代码变更影响的情况下, 漏洞检测的结果不会出现波动. 冗余的计算将导致企业投入额外的计算成本, 并增加开发者等待 CI/CD 流程的时间. 相关研究表明, 开发人员对扫描器的低误报率和运行效率有极高的预期, 要求工具在代码变更时仅针对增量部分进行快速分析 (而非扫描整个仓库), 且 75% 开发者无法容忍超过数分钟的运行时长[9,10]. 同时, 开发者期望工具需深度集成至 IDE 或 PR 审查等开发工作流, 并在分支切换前提供即时反馈以降低理解成本[11].

针对上述问题, 增量分析是一个理想的解决方案, 该方法通过高效复用已有扫描结果, 仅分析输入程序中的增量或修改部分, 从而达到减少重复计算, 节省时间的效果. 现有的增量指针分析方法[12−18]与增量图分析方法[19−21]通过感知两次输入程序中差异部分, 计算指针指向集或超图的变化, 从而完成增量分析. 然而, 这两类方法在工业级规模的 Web 应用场景中面临分析粒度与资源消耗的矛盾, 高精度的上下文与域敏感分析常因内存爆炸问题无法应用于百万行级代码库. 函数摘要式增量分析[22−24]是另一种热门的增量分析方法, 该技术以函数为单元进行抽象建模, 该方法能与污点传播等分析目标所要求的程序语义建立等价映射, 从而通过函数摘要的变化精确界定代码变更的影响范围, 仅需局部重分析即可保证全局正确性. 因此, 函数摘要技术成为目前实现工业级规模 Web 应用漏洞检测中平衡“精度”与“效率”的关键解决方案. 但是, 将函数摘要技术用于 Web 应用漏洞检测中还有如下挑战.

● 挑战 1. 支持类型丰富的 Web 应用漏洞: Web 应用包含各种各样的漏洞类型, 如跨站脚本攻击 (XSS)、SQL 注入 (SQL injection, SQLI)、命令注入、跨站请求伪造 (server-side request forgery, SSRF) 等. 不同漏洞模式对应差异化的语义依赖关系 (如 XSS 需追踪字符串拼接路径中的污染标记传播, SQLI 需关联被污染字段与数据库表结构约束, SSRF 需验证请求地址主机是否受用户输入控制). 它要求函数摘要结构具备语义建模能力, 能够根据具体漏洞类型, 表征变量间的污点传播路径. 然而, 现有函数摘要式增量分析技术[22−25]面向单一漏洞类型 (如未初始化变量漏洞) 设计抽象机制, 其函数摘要设计仅聚焦于变量之间的状态信息, 难以兼容多类型漏洞对污点传播路径的

建模需求因此, 如何设计一个可支持多种 Web 应用漏洞的摘要结构以用于增量分析, 是待解决的一大挑战.

● 挑战 2. 精度、效率和一致性的统一: 精度、效率与一致性是衡量增量分析的关键指标. 在分析精度方面, Web 应用漏洞检测需捕捉复杂的变量语义依赖关系, 要求摘要结构具备域敏感性和上下文敏感性[26]: 前者需区分对象属性/数组索引等结构化数据的差异化传播路径, 后者需刻画不同调用上下文对函数行为的影响. 在分析效率方面, 全路径敏感分析和传统迭代式数据流分析均难以满足工业级性能要求, 函数摘要算法应支持在线性时间复杂度条件下生成并适用于所有调用上下文的摘要, 且其所需的存储空间应尽可能小, 同时支持快速导入并还原分析状态. 此外, 增量计算时要尽可能仅考虑受影响部分以提升效率. 最后, 增量分析的结果要与全量分析的结果一致, 否则会给安全运维人员带来困扰. 当前大多数摘要式增量分析方法在存在过于保守或过于激进的问题[22,23,25]: 前者引入大量无关代码重复分析, 降低效率; 后者可能未充分分析代码变更的实际影响, 从而遗漏关键的语义变化与数据流更新, 导致结果与全量分析不一致. 因此, 如何在增量分析过程中, 实现精度、效率和一致性的统一是我们面对的另一大挑战.

对于挑战 1, 我们设计了一种基于输入输出映射的函数摘要结构. 该结构不仅可以表示变量之间的状态关系, 还能够表征输入变量到输出变量的污点传播信息. 此外, 本文的函数摘要与漏洞检测规则绑定, 不仅包含函数输入输出变量关联的完整数据流特征, 还包含规则相关的污点语义封装信息, 从而支持多种 Web 应用漏洞检测. 对于挑战 2, 本文方法将传统增量分析中需要维护的中间状态从完整的程序依赖图缩减为模块化的函数级特征描述. 在摘要计算中引入采用符号化字段, 对函数内的数据流传播进行按需拆分, 从而完成域敏感与流敏感的数据流分析, 并在线性时间内生成可适用于所有调用上下文的函数摘要. 我们还提出了一种细粒度增量计算方法, 与传统方法将代码变更简单地映射为文件级或类级重新分析不同, 该方法通过两个列表 (删除代码行号列表和新增代码行号列表) 来精确表示增量信息, 并将其与调用图相映射以获取一个以函数为单位的保守增量变化范围. 在此基础上, 进一步利用函数摘要的变化感知和计算具体的增量变化影响范围, 从而避免不必要的重复分析. 综上所述, 本文提出的轻量化且高精度的函数摘要生成机制与细粒度增量影响范围分析算法相协同, 能够有效满足工业级 Web 应用漏洞检测对分析精度、效率与结果一致性的 3 项核心要求.

我们实现了一个面向 Web 应用漏洞检测的增量静态分析框架 AWTaint, 该框架基于污点分析与数据流分析, 既可以对 Web 应用进行全量分析产出中间结果与漏洞报告, 又可以对其进行增量分析, 基于已有中间结果产出增量式的漏洞报告. 本文构建了一个包含 5 个开源热门 Java Web 应用以及 5 个来自真实工业场景的闭源 Java Web 应用的数据集, 涵盖 SpringBoot、Struts 等主流框架, 包含 210 次有效代码提交. 本文从多维度评判了函数摘要表述能力, 增量分析的分析效率, 以及分析一致性. 实验结果表明, AWTaint 相较于全量扫描, 平均加速 3.63 倍, 内存峰值控制在 8 GB 内, 且漏洞检测结果完全一致. 在存储成本方面, AWTaint 的万行代码中间结果存储大小仅需 1.52 MB (相比之下, CodeQL 万行代码中间结果达 79.4 MB[14,27]).

总结而言, 本文做出了以下核心贡献.

(1) 提出一种面向 Web 应用漏洞检测的函数摘要表示形式, 该形式不仅可以表示输入输出变量之间的污点传播关系, 还可以根据检测规则产出对应的污染传播信息, 从而适配多类型漏洞检测需求. 此外, 该函数摘要具备域敏感、流敏感以及上下文敏感能力, 可在线性时间复杂度内生成可适用于所有调用上下文的函数摘要.

(2) 提出一种面向 Web 应用漏洞检测的增量计算方法. 该方法可以精确计算增量变化影响范围, 与函数摘要更新协同, 在增量分析过程中, 完成对分析精度、效率和一致性的统一.

(3) 实现一套支持增量分析的 Web 应用漏洞检测框架 AWTaint. 实验结果表明, 该框架能够扫描出真实环境中存在的多种 Web 应用漏洞, 且在域敏感、流敏感、上下文敏感等能力上达到行业领先水平. 框架的增量分析模式可在保障检测精度与结果一致性的前提下, 显著提升分析效率, 同时有效控制存储成本与内存开销, 现已成功应用于工业级规模的 Web 应用场景.

本文第 1 节介绍本工作的相关背景知识. 第 2 节介绍提出的增量分析算法. 第 3 节介绍静态分析器的原型实现. 第 4 节通过实验展示 AWTaint 的性能. 第 5 节讨论本增量分析算法的局限性. 第 6 节回顾相关工作. 第 7 节对工作进行总结, 并提出未来展望.

1 背景知识

1.1 基于污点分析的 Web 应用漏洞检测

污点分析作为检测程序漏洞的重要技术手段, 其核心在于追踪外部输入对程序状态的影响: 当攻击者向程序注入恶意数据且未被适当防护时, 可能导致系统进入不安全状态, 这些受用户或文件输入, 主函数参数等外部来源影响的数据在分析中被标记为“污染”. 该技术通过识别程序中可能被污染的变量并追溯其传播路径, 最终定位未经验证直接使用受污染数据从而引发漏洞的危险语句. 根据实现方式不同, 污点分析可分为静态与动态两类, 前者无需实际执行代码即可实现更高的源代码覆盖率, 但可能因缺乏运行时信息产生误报或漏报; 后者则通过执行程序动态追踪数据流向, 虽能确保结果的真实性, 却难以全面覆盖所有代码路径.

在基于 Web 应用的污点分析中, 大多数研究者鉴于其特有的执行环境与分析目标, 通常会对传统静态分析方法进行针对性调整: 由于 Web 应用的请求处理逻辑多为单次执行且循环路径对污点传播的实际影响有限, 分析器常忽略循环回边和递归结构以避免状态爆炸问题; 同时, 考虑到 Web 程序的路径数量呈指数级增长而实际有效攻击路径相对集中, 多数实现会弱化路径敏感能力, 转而采用路径无关的分析方式[28−32]. 专注于追踪变量的污点状态而非具体值, 这种简化虽可能引入一定误报, 却显著提升了分析效率与可扩展性, 使其能够高效应对大规模 Web 应用的代码扫描需求.

1.2 增量分析算法

传统的静态分析方法通常是全量扫描, 即每次静态分析过程都是独立的, 都会从头分析项目中当前的所有代码生成相应的缺陷. 在这个过程中每次从头分析所有代码无疑需要大的内存开销和时间开销, 这对于基于版本控制的持续集成环境来说并不是优化的解决方案. 版本控制是一种记录一个或若干文件内容变化, 以便将来查阅特定版本修订情况的系统, 是持续集成环境必不可少的功能之一. 在版本控制系统中, 项目的代码变化被组织成一次次的提交 (commit), 每次提交可以理解为一个在原来基础上的“增量变化”.

在基于版本控制的持续集成环境中, 优化的静态安全缺陷检测解决方案应该是增量式的: 应尽量避免对程序中“增量变化”影响不到的部分进行分析, 只分析那些“增量变化”中和受“增量变化”影响到的部分. 显然, 增量分析过程中每次的分析对象不能仅是“增量变化”中的部分, 应还要包括“增量变化”影响到的部分.

从调用图的角度考虑, 增量变化带来的影响可分为两部分: 由增量变化直接造成的影响和由增量变化间接造成的影响. 对于删除代码来说, 直接影响包括原来调用图中直接调用被删函数的函数和直接被被删函数调用的函数 (被删函数的前驱节点和被删函数的后继节点). 对于增加代码来说, 直接影响包括新增函数, 新调用图中直接调用新增函数的函数和直接被新增函数调用的函数 (新增函数的前驱节点和新增函数的后继节点). 删除代码和新增代码的直接影响可分别用后文图 1(a) 和 (b) 表示.

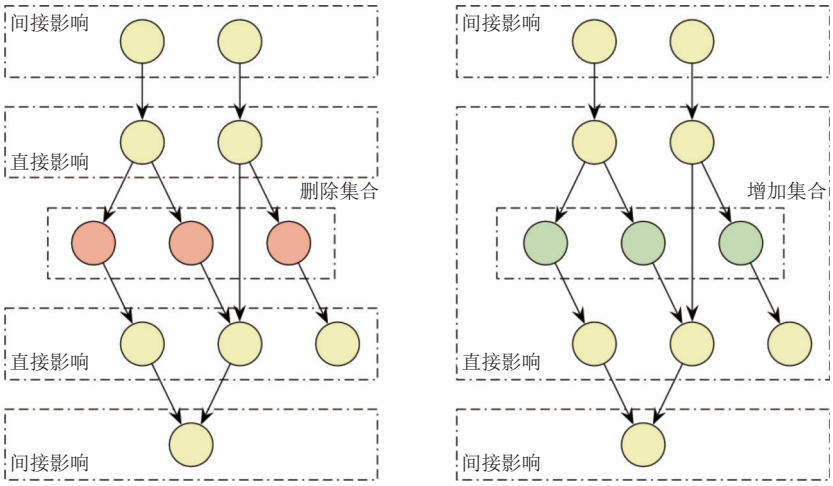
1.3 基于函数摘要的静态分析

上下文敏感 (context-sensitive) 分析是程序分析领域提升精度的核心技术, 其通过区分函数在不同调用场景下的行为差异来避免误报. 以图 2 为例, 当 goo() 在第 3 行被调用时参数 p 处于污染状态, 而在第 5 行调用时为非污染状态. 非上下文敏感分析会因混淆这两种场景导致第 5 行误报. 全展开分析虽然能解决此问题, 但其指数级时间复杂度 ($O(2^n)$) 使其难以应用于大规模程序. 函数摘要技术通过构建函数行为的抽象描述实现了效率与精度的平衡. 该方法在首次分析函数时生成其输入输出关系的抽象表示 (如数据流信息), 后续调用可直接复用该摘要信息, 避免重复分析函数体[33].

函数摘要的设计过程中, 域敏感 (field-sensitive) 是一个需要重点考虑的关键因素. 域敏感性要求函数摘要能够精确刻画函数中的对象属性 (如面向对象语言中的成员变量) 或容器元素 (如数组, 集合等) 的访问和修改行为, 而非仅将对象或容器视为整体. 这种细粒度的分析能力对于避免误报, 提升程序分析的准确性至关重要[33]. 如图 3 展示的案例显示, 若将对象 p 的整体状态作为分析单元, 会错误地将 $p.f$ 的污染传播到 $p.g$ 字段. 当前主流解决方案采用访问路径 (access path) 建模技术[34], 其将变量表示为“基础变量.访问字段”的形式 (如 $p.f.g.h$), 其中字段链

长度通常受超参数 k 限制以控制复杂度[35].

该分析技术已在工业级工具链中广泛应用: 如 FlowDroid 基于域敏感摘要进行 Android 应用污点分析[34], Clang Static Analyzer 则采用混合粒度摘要实现性能优化[36]. 然而, 大多数工作依然采用 k-limited 的方法来限制访问深度, 难以应对复杂的 Web 应用场景.



(a) 删除增量变化影响 (b) 新增增量变化影响

图 1 增量影响范围图例

```
1  void foo(String p){
2          p = source();
3          goo(p);
4          p = sanitizer(p);
5          goo(p);
6  }
7  void goo(String p){
8          sink(p);
9  }
```

图 2 需要上下文敏感分析的代码样例

```
1  void foo(A p){
2      p.f = source();
3      sink(p.g);
4  }
```

图 3 需要域敏感分析的代码样例

2 方 法

2.1 方法框架

本方法的静态分析架构采用分层递进式设计, 基于污点分析与数据流分析技术, 通过抽象语法树 (abstract syntax tree, *AST*)、函数调用图 (call graph, *CG*) 和控制流图 (control flow graph, CFG) 实现多层次代码解析. 如图 4 所示, 本方法有着两种分析模式, 分别对应全量分析以及增量分析两种场景.

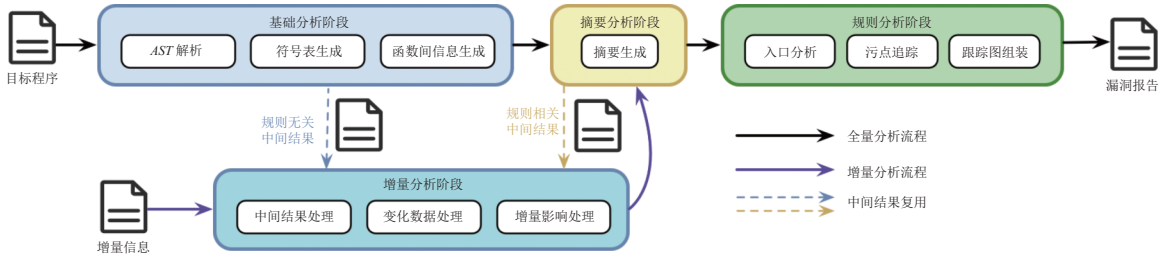


图 4 AWTaint 工作流

● 全量分析模式: 其核心是生成中间数据、函数摘要等信息. 在全量分析中, 系统核心流程包含 3 个递进分析阶段.

(1) 基础分析阶段: 通过词法解析和语法解析构建 *AST*, 同时建立函数级符号表和类型推断系统, 为后续分析奠定语义基础; 基于包导入关系补全类型信息, 生成函数调用图并识别程序入口点, 最终输出包含 *AST* 结构、过程间类型系统和调用图关系的规则无关中间表示.

(2) 摘要分析阶段: 根据给定规则, 分析程序中所有函数的数据传播, 从而生成域敏感、流敏感、上下文敏感的函数摘要. 函数摘要的相关技术细节在第 2.2 节中详细讨论.

(3) 规则分析阶段: 根据分析入口将所有函数摘要组合在一起. 分析器会根据给定的污染源 (source) 和汇聚点 (sink) 组合污点传播路径, 生成漏洞报告.

● 增量分析模式: 其核心是通过计算增量影响范围来完成快速分析. 在增量分析中, 系统核心流程包含 3 个递进分析阶段.

(1) 增量分析阶段: 导入上轮扫描的中间结果, 处理变化数据, 并基于增量计算方法, 对影响范围进行界定. 相关技术细节将在第 2.3 节中详细讨论.

(2) 摘要分析阶段: 根据影响范围, 递归生成函数摘要. 将发生变更的函数摘要转化具体增量影响范围, 用于后续规则分析. 相关技术细节将在第 2.3 节中详细讨论.

(3) 规则分析阶段: 根据具体增量影响范围, 重新分析受影响的分析入口, 整合上一轮扫描的分析结果, 生成漏洞报告.

2.2 面向 Web 漏洞检测的函数摘要

在程序分析领域, 上下文敏感分析的基准方法是调用点展开, 其通过在每个调用点独立展开被调函数体实现上下文区分. 然而该方法具有指数级时间复杂度 $O(2^n)$ (其中 n 为调用链深度), 导致其难以应用于工业级代码库的分析. 相较而言, 函数摘要技术通过构建函数行为的抽象语义描述, 在保证分析精度的同时显著提升分析效率. 其核心思想是将函数语义压缩为输入输出关系的映射, 从而避免重复展开函数.

2.2.1 函数摘要结构与设计

函数摘要的核心设计原则是面对分析问题对函数尽可能抽象. 在 Web 应用漏洞检测场景中, 面向函数调用的污点分析主要聚焦于 3 类核心抽象语义: 其一, 函数调用对“当前污染变量集合”的动态影响. 例如, sanitizer 操作可通过从集合中移除特定元素实现净化效果; 其二, 函数内部是否封装了 source/sink 操作单元, 即是否包含污染源注入点或漏洞触发点; 其三, 函数体内污点变量的传播路径与传递机制. 这 3 个维度的抽象语义共同构成了函数摘要设计的理论基础.

基于上述分析目标, 本文提出一种包含 3 种属性的函数摘要结构. 该结构要求每个函数摘要 s 必须包含以下核心属性: ① 污染传播信息; ② 封装信息; ③ 污点传播图. 其中, 污染传播关系的表达能力直接决定摘要的上下文敏感能力, 是本工作的核心设计重点. 形式化地, 对于 $m \in \mathbb{M}$, 其函数摘要 s_m 由三元组 $(\mathbb{P},\mathbb{E},\mathbb{G})$ 构成, 分别对应上述 3 种信息, 具体定义如下.

● 污染传播信息 $\mathbb{P} \subseteq \mathcal{A}\times\mathcal{A}$: 定义为访问路径间的二元关系集合, 其中, $\mathcal{A}$ 代表访问路径集合, 如 $v.\lambda$, v 为基础变量, λ 为访问字段. $\langle v.\lambda, v'.\mu\rangle \in \mathbb{P}$ 代表当访问路径 $v.\lambda$ 被污染的时候, 其污点可传播至访问路径 $v'.\mu$.

● 封装信息 $\mathbb{E} = \mathbb{E}_{\text{source}} \cup \mathbb{E}_{\text{sink}}$: 定义包括两部分, $\mathbb{E}_{\text{source}} \subseteq LoC_{\text{source}}\times\mathcal{A}$ 代表外部污染源 LoC_{source} (如 http 请求参数) 可以将污染传播到访问路径 $v.\lambda$, $\mathbb{E}_{\text{sink}} \subseteq \mathcal{A}\times LoC_{\text{sink}}$ 代表访问路径 $v.\lambda$ 可以将污染传播到漏洞触发点 $\mathbb{E}_{\text{sink}}$ (如 SQL 查询接口).

● 污染传播图 $\mathbb{G} = (N, E)$: 定义为有向图, 其中节点集 $N \subseteq \mathcal{A}$ 表示带访问路径的污点传播节点, 边集 $E \subseteq \mathbb{P}$ 记录污点传播的详细路径, 该结构仅保留关键中间传播节点, 为漏洞报告生成提供可追溯的路径信息.

针对在函数边界处的传播行为, 设函数 f 的输入变量集合为 $X = \{x_1, x_2, x_3, \ldots, x_n\}$, 其污染传播行为可表示为幂集映射 $\mathcal{F}: \mathcal{P}(X) \to \mathcal{P}(Y)$, Y 为输出污染变量集合. 传统方法要求该映射覆盖所有 2^n 种输入组合, 因此输入空间大

小为 $O(2^n)$, 导致摘要空间复杂度呈指数增长.

基于对 OWASP Top 10 漏洞的模式分析, 本研究发现 Web 漏洞基本都满足单输入的触发特性. 在注入漏洞 (SQL/XSS) 中, 仅需单一污染源即可触发漏洞 (例如通过 URL 参数注入恶意字符串); 在数据泄露类漏洞中, 需要依赖单一敏感字段的非预期传播来判断漏洞 (例如会话令牌的跨域传输). 形式化地, 假设 $VulnTrigger(\cdot)$ 为漏洞触发谓词, 则 Web 漏洞大多数满足析取性约束, 如公式 (1) 所示:

$$VulnTrigger(A \cup B) \equiv VulnTrigger(A) \vee VulnTrigger(B) \tag{1}$$

该析取性约束表明了漏洞触发条件对输入污染源具有逻辑或关系, 即污染传播行为映射满足分配律 $\mathcal{F}(A \cup B) = \mathcal{F}(A) \cup \mathcal{F}(B)$ 时, 该分析框架即构成分配框架, 其具有最优精度保证: 此时最大不动点 (maximum fixed point, MFP) 解与全路径汇聚 (meet overall path, MOP) 解相等, 即 MOP=MFP[34,37]. 因此, 基于分配框架的污染分析可在多项式时间内获得最精确解, 即可将原始映射 $\mathcal{F}$ 分解为 n 个单变量子映射的并集 (如公式 (2)). 该分解将摘要空间复杂度从 $O(2^n)$ 降低至 $O(n)$, 同时保留漏洞检测所需的上下文敏感性.

$$\mathcal{F} = \bigcup_{i=1}^{n} \mathcal{F}_i,\ \mathcal{F}_i : \mathcal{P}(\{x_i\}) \to \mathcal{P}(Y) \tag{2}$$

对于代码 foo(pl, p2){*return* p1+p2;}, 采用输入污染变量集合到输出污染变量集合的映射, 在所示 *foo* 函数示例中, 污染传播行为可通过单变量分解形式精确建模. 设输入变量集合 X={pl, p2}, 输出变量集合 Y={*ret*}, 其污染传播映射可以及分解为两个单变量的子映射, 分别对应输入变量 $p1$ 和 $p2$ 的污染传播路径. 完整的污染传播映射 $\mathcal{F}$ 是这两个子映射的并集, 即 $\mathcal{F} = \mathcal{F}_{p1} \cup \mathcal{F}_{p2}$. 无论输入集合为 ∅、{$p$1}、{$p$2}或者是{$p$1, p2}, 输入污染状态始终遵循“任一输入污染即导致输出污染”的规则, 详见公式 (3):

$$\mathcal{F} = \bigcup_{i=1}^{n} \mathcal{F}_i,\ \begin{cases} \mathcal{F}(\varnothing) = \varnothing \\ \mathcal{F}(\{p1\}) = \{ret\} \\ \mathcal{F}(\{p2\}) = \{ret\} \\ \mathcal{F}(\{p1, p2\}) = \mathcal{F}(\{p1\}) \cup \mathcal{F}(\{p2\}) = \{ret\} \end{cases} \tag{3}$$

对于污染传播信息 $\mathbb{P}$ 而言, 多个输入污染变量的污染传播效果可以通过分别计算单个输入污染变量的污染传播效果, 然后求集合并集得到. 对于函数摘要中的封装信息 $\mathbb{E}$ 而言, $\mathbb{E}_{source}$ 封装信息主要记录 source 点程序位置和新增的输出污染变量集合, $\mathbb{E}_{sink}$ 封装信息主要记录 sink 点程序位置和触发 sink 点的输入污染变量集合, 其本质上是较为特殊的污点传播信息. 而函数摘要中的污染传播图 $\mathbb{G}$ 通过有向图结构刻画污点流动的轨迹, 该结构通过保留关键中间传播节点 (如函数参数、返回值、全局变量、赋值信息), 为漏洞触发路径的可追溯性分析提供了细粒度的中间表示. 通过精简非关键节点, 在保证传播路径完整性的前提下提升了分析效率.

2.2.2 域敏感处理

在静态程序分析领域, 域敏感方法通过引入访问路径来精确描述复杂数据结构的字段级传播关系. 访问路径定义为从基础变量出发, 通过连续字段访问形成的操作序列, 其形式化表示为“基础变量.(访问字段)*”. 例如, 表达式 $p.f.g.h$ 对应的访问路径长度为 3, 其中 p 为基础变量, $f.g.h$ 为字段访问序列. 这种表示法能够精确捕捉对象嵌套结构的语义特征, 但同时也带来了理论上的挑战: 高级语言的递归嵌套特性导致访问路径长度可能无限增长. 考虑函数“*bar*(p1): *return* p1;”, 其语义特征决定了输入变量 p 与返回值之间存在完全的数据依赖关系. 当采用传统访问路径分析时, 该函数的输入输出映射将扩展为无限集合, 如公式 (4) 所示:

$$\begin{cases} \{p\} \to \{return\} \\ \{p.f\} \to \{return.f\} \\ \{p.f.g\} \to \{return.f.g\} \end{cases} \tag{4}$$

针对上述问题, 本文提出基于符号字段的抽象表示法. 通过引入特殊标记符“$”, 将访问路径扩展为“基础变量.(访问字段)*[$]”的混合表示形式. 其中符号字段“$”作为通配符, 可匹配任意长度的实际字段序列. 对于示例函数 *bar*, 其输入输出映射可抽象为有限形式, 如公式 (5) 所示. 这种表示本质上建立了输入/输出空间的同构划分, 将无限的访问路径集合投影到有限的符号表示空间, 当且仅当两个访问路径共享相同的符号字段前缀时, 它们属于

同一划分等价类.

$$\{p.\$\} \to \{return.\$\} \tag{5}$$

引入符号字段“$”后, 函数摘要的输入和输出变量空间将转化为有限集: 该有限集的元素由当前函数代码对应语义决定, 有限集中元素的个数为当前函数输入或输出空间对于污点传播效果来说的同构划分个数. 输入空间有限集元素的确定可采用“按需拆分”的策略: 在函数分析的开始, 对输入变量不进行字段拆分, 而是在分析过程中“根据需要”拆分出对应的实际字段. 本方法提出的符号字段抽象表示方法具有类型泛化特性, 可有效覆盖基础类型变量与复合数据结构的域敏感需求. 具体而言, 针对类实例变量及 Array/Map 等复合类型结构, 系统将执行轻量级的函数内数据流回溯分析, 以推导索引表达式的确定性取值. 当检测到索引项为原子常量时 (包含数值型常量, 字符串字面量及类级常量引用), 分析器将采用该确定值作为符号域标识符, 实现精确的域敏感分析. 对于非确定性索引访问场景, 如 Map 类型字段的赋值模式 (形如 *map.put*(*varl*, *taint*)), 我们会建立一个模糊字段 K 来表征不可解析的键值维度, 在这类情况中, 变量 *map* 的符号字段表示为 *map*.@UK.$.

为了实现对符号化字段的应用以及按需拆分的策略, 本文引入映射 $\mathcal{O}$ 表示污染来源, 该映射与每个程序位置点上的污染变量集合 $\mathcal{T}$ 一同更新, 以计算出最终的污点传播信息 $\mathbb{P}$. 形式化表示为, 其中, v_{cur} 表示当前的污染变量 (一般为被分析的变量), v_{orig} 表示初始的污染来源变量 (例如函数的输入参数、*this* 变量等). $v \in \mathcal{A}$, v 中的访问字段以符号化字段的形式表示, 可以通过当前污点分析的污染变量集合进行运算.

在对任意程序位置点对污染变量集合 $\mathcal{T}$ 进行更新后, 需要动态维护 $\mathcal{O}$, 从而体现出带访问字段的变量与污染来源变量之间的关系. 具体而言, 在对每个函数进行分析的时候, 我们会假设所有输入变量都是污染的, 因此所有输入变量均处在污染变量集合 $\mathcal{T}$ 中. 在分析过程中, 对于每个能够关联到输入变量某个字段的访问, 我们会将其加入 $\mathcal{O}$, 并更新污染传播图 $\mathbb{G}$. 特别的, 如果该字段将该输入空间划出一个更小的子输入空间, 我们会将其按需分裂. 我们将该过程记录为 updateOrigin, 该过程的输入信息包括映射 $\mathcal{O}$, 污染传播关系 $\langle v.\lambda, v'.\mu\rangle$ 以及污染传播图 $\mathbb{G}$.

本文将以表 1 中所示的代码为例, 阐述具体的 $\mathcal{O}$ 更新逻辑. 首先, 在函数入口处, 我们假设输入变量 p 污染, 将 $p.\$$ 加入污染变量集合 $\mathcal{T}$, 并向映射 $\mathcal{O}$ 中加入 $p.\$ \mapsto \{p.\$\}$, 代表输入污染变量 $p.\$$ 污染来自本身. 此时映射 $\mathcal{O}$ 如表 1 第 1 行所示. 之后, 分析语句 *T q=p*, 由于 $p.\$$ 将污染信息传递给了 $q.\$$, 因此将 $q.\$$ 加入污染变量集合 $\mathcal{T}$, 与此同时, 我们向 $\mathcal{O}$ 中加入 $q.\$ \mapsto \{p.\$\}$, 代表输入污染变量 $q.\$$ 污染来自 $\{p.\$\}$, 此时映射 $\mathcal{O}$ 如表 1 中第 2 行所示. 最后, 处理语句 *return q.f*, 我们将其转换为类似 *return q.f* 的语句进行处理. 首先处理赋值右边, 发现该字段访问的是 $q.\$$ 的更细划分, 在 $\mathcal{O}$ 中并不包含该划分, 因此需要对 $\mathcal{O}$ 中的 $q.\$$ 及其污染来源 $\{p.\$\}$ 进行分裂, 向 $\mathcal{O}$ 中加入 $q.f.\$ \mapsto \{p.f.\$\}$, 接着处理对 *return* 的赋值, 得到 $return.\$ \mapsto \{p.f.\$\}$. 此时映射 $\mathcal{O}$ 如表 1 第 3 行所示.

表 1 对象关系分析结果

Statement	Result of $\mathcal{O}$
Object zoo(*T p*){	$\{p.\$ \mapsto \{p.\$\}\}$
T q = p;	$\begin{cases} p.\$ \mapsto \{p.\$\} \\ q.\$ \mapsto \{p.\$\} \end{cases}$
return q.f; }	$\begin{cases} p.\$ \mapsto \{p.\$\} \\ q.\$ \mapsto \{p.\$\} \\ q.f.\$ \mapsto \{p.f.\$\} \\ return.\$ \mapsto \{p.f.\$\} \end{cases}$

最后, 当前函数分析结束, 需要计算污点封装信息 $\mathbb{P}$, 其结果组成来自能够传播到 *return*.$ 变量的起始变量. 因此, 该函数的封装信息 $\mathbb{P}$ 为 $\{p.f.\$ \mapsto \{return.\$\}\}$. 在面向对象的程序语言中, 需要额外记录类变量 *this*.$为起始变量. 在计算污点封装信息 $\mathbb{E}$ 时, 其结果组成还要包括能够传递到 *this*.$任意划分的起始变量. 总结而言, 在 *zoo* 函数分析开始时, 对于输入变量 p 的所有输入空间用 $p.\$$ 表示, 当分析到第 3 行的 $q.f$ 时, 这时触发 $q.\$$ 的起始污染变量 $p.\$$ 的拆分, 即此时从 p 的所有输入空间用 $p.\$$ 拆分出子划分 $p.f.\$$. 因此, “按需拆分”是一种“懒惰初始化”策略.

2.2.3 函数摘要生成算法 *BuildSummary*

与基于 IFDS 的经典分析工具 (如 FlowDriod[34]) 不同, 我们采用自底向上的分析模式, 为每一个函数生成包含 $(\mathbb{P},\mathbb{E},\mathbb{G})$ 三元组的函数摘要. 对于一组调用关系来说, 首先生成被调函数的函数摘要. 分析过程中循环仅展开一次, 忽略递归调用, 以避免迭代计算, 同时保留流敏感分析的能力. 随后, 对每个函数 $f \in \mathbb{F}$ ($\mathbb{F}$ 为所有函数的集合), 我们将调用 *BuildSummary*(*f*) 构建其摘要. 每个函数仅需生成一次摘要, 生成后可复用于所有调用上下文中. 算法 1 展示了函数摘要的具体生成过程. 对给定函数 *f*, 首先将 $\mathbb{P}$, $\mathbb{E}$, $\mathbb{G}$ 初始化为空集, 然后在函数的每个程序位置上计算当前污染变量集合 $\mathcal{T}$ 和污染来源映射 $\mathcal{O}$, 并返回 $(\mathbb{P},\mathbb{E},\mathbb{G})$ 作为 *f* 的函数摘要 *s*.

算法 1. 函数摘要生成算法 *BuildSummary*.

输入: f;
输出: $(\mathbb{P},\mathbb{E},\mathbb{G})$ as s.

1. $\mathbb{P},\mathbb{E}_{\text{sink}},\mathbb{E}_{\text{source}},\mathbb{G} \leftarrow \varnothing,\mathcal{T} \leftarrow \varnothing,\mathcal{O} \leftarrow \varnothing$
2. **for** $statement \in f$ **do**
3. **if** $statement$: $v'.\mu \leftarrow v.\lambda$ **then**
4. **if** $v.\lambda \in \mathcal{T}$ **then**
5. $\mathcal{T} \leftarrow \mathcal{T} \cup \{v'.\mu\}$
6. $p.\phi \leftarrow find(v.\lambda,\mathcal{O})$
7. $updateOrigin(v'.\mu \leftarrow p.\phi,\mathcal{O},\mathbb{G})$
8. **end**
9. **end**
10. **else if** $statement$: $v'.\mu \leftarrow source()$ **then**
11. $\mathcal{T} \leftarrow \mathcal{T} \cup \{v'.\mu\}$
12. $updateOrigin(v'.\mu \leftarrow LoC_{\text{source}},\mathcal{O},\mathbb{G})$
13. **end**
14. **else if** $statement$: $sink(v.\lambda)$ **then**
15. $p.\phi \leftarrow find(v.\lambda,\mathcal{O})$
16. $\mathbb{E}_{\text{sink}} \leftarrow \mathbb{E}_{\text{sink}} \cup \{\langle p.\phi, LoC_{\text{sink}}\rangle\}$
17. **end**
18. **else if** $statement$: $return\ v.\lambda$ **then**
19. $p.\phi \leftarrow find(return,\mathcal{O})$
20. **if** $p.\phi$ is source **then**
21. $\mathbb{E}_{\text{source}} \leftarrow \mathbb{E}_{\text{source}} \cup \{\langle LoC_{\text{source}}, return.\phi\rangle\}$
22. **end**
23. **else**
24. $\mathbb{P} \leftarrow \mathbb{P} \cup \{\langle return.\mu \leftarrow p.\phi\rangle\}$
25. **end**
26. **end**
27. **else if** $statement$: $res = func(v.\lambda)$ **then**
28. $func' \leftarrow resolve(func)$
29. $s \leftarrow BuildSummary(func')$
30. **for** $\langle v.\lambda, v'.\mu\rangle \in \mathbb{P}_s$ **do**
31. $\mathcal{T} \leftarrow \mathcal{T} \cup \{res\}$

32. $\mathcal{O} \leftarrow updateOrigin(res \leftarrow v'.\mu, \mathcal{O}, \mathbb{G})$
33. **end**
34. **for** $\langle LoC_{\text{source}}, v'.\mu \rangle \in \mathbb{E}_{\text{source}}$ **do**
35. $\mathcal{T} \leftarrow \mathcal{T} \cup \{res\}$
36. $\mathcal{O} \leftarrow updateOrigin(res \leftarrow LoC_{\text{source}}, \mathcal{O}, \mathbb{G})$
37. **end**
38. **for** $\langle v.\lambda, LoC_{\text{sink}} \rangle \in \mathbb{E}_{\text{sink}}$ **do**
39. $\mathbb{E}_{\text{sink}} \leftarrow \mathbb{E}_{\text{sink}} \cup \{\langle res, LoC_{\text{sink}} \rangle\}$
40. **end**
41. **end**
42. **end**

● 过程内分析: 我们按照程序控制流顺序对当前函数进行分析. 当进入控制流分支时, 当前分析状态会被复制, 并分别沿各分支继续执行; 在控制流交汇点, 采用 MAY 合并策略[34]对多个状态进行聚合. 我们重点关注赋值语句、源点/汇点语句、函数调用以及函数返回语句的处理, 其余部分遵循常见的污点分析流程.

● 赋值语句处理: 对于形如 $v'.\mu \leftarrow v.\lambda$ 的赋值语句. 其中 $\_.\mu$、$\_.\lambda$ 表示对变量的访问路径, 长度可为 0 (即无域字段), 1 或多个字段. 如果 $v.\lambda$ 属于污染变量集合 $\mathcal{T}$, 则将 $v'.\mu$ 加入 $\mathcal{T}$, 并通过 $find(v.\lambda, \mathcal{O})$ 获取其与输入污染源的关系, 得到传播映射 $v'.\mu \leftarrow p.\phi$. 我们利用 *updateOrigin* 处理该映射关系, 更新污染来源映射表 $\mathcal{O}$ 与图 $\mathbb{G}$.

● 源点语句处理: 对于形如 $source(v.\lambda)$ 的语句, 若该语句命中规则的 source 配置, 则将 $v.\lambda$ 加入 $\mathcal{T}$, 并加入污染来源映射 $\mathcal{O}$. 在后续污染传播中, 对其进行跟踪与分裂.

● 汇点语句处理: 对于形如 $sink(v.\lambda)$ 的语句, 若该语句命中规则中的汇点配置且 $v.\lambda \in \mathcal{T}$, 则通过 $find(v.\lambda, \mathcal{O})$ 获取其与输入污染变量的关系, 并将结果整合为 LoC_{sink}, 加入 $\mathbb{E}_{\text{sink}}$. 这表明当前函数内存在某个输入变量的特定划分能够触发汇点.

● 函数调用及返回语句处理: 对于函数调用, 除了应用函数摘要外, 实参和形参之间存在隐含的赋值操作, 需按照赋值语句进行处理. 对于形如 $return v.\lambda$ 的返回语句, 也将其转化为赋值语句处理. 处理完成后, 通过 $find(return, \mathcal{O})$ 获取当前返回变量与输入污染变量的关系, 并推导出 $return.\mu \leftarrow p.\phi$ 中, 将该记录更新至 $\mathbb{P}$. 若该信息来源于规则中的源点配置 (而非输入污染变量集合), 则将结果整合为 LoC_{source}, 并加入 $\mathbb{E}_{\text{source}}$, 表示该函数返回值的某个划分引入了一个源点. 随后, 从当前返回语句向后分析, 重新收集所有赋值语句和函数返回语句, 将这些信息转化为污点跟踪图并加入 $\mathbb{G}$.

● 过程间分析: 过程间分析的核心在于将被调函数的函数摘要应用于当前函数分析上下文, 当遇到调用语 $res = func(v.\lambda)$ 时, 首先通过类继承关系分析 (class hierarchy analysis, CHA) 和过程内类型分析定位目标函数 *func*. 随后, 从摘要集合 $\mathcal{S}$ 中读取 *func* 的摘要 s (若存在); 否则递归调用 $BuildSummary(func)$ 生成. 获取 *func* 的摘要 s 后, 需要在当前函数上下文中应用该摘要. 具体分为如下 3 种情况.

(1) 污染传播 $\mathbb{P}$ 应用: 若存在 $\langle v.\lambda, v'.\mu \rangle \in \mathbb{P}$, 则将 $v'.\mu$ 的污染传递给 *res*, 并更新当前函数的 $\mathcal{O}$.

(2) 污染源封装信息 $\mathbb{E}_{\text{source}}$ 应用: 若存在 $\langle LoC_{\text{source}}, v'.\mu \rangle \in \mathbb{E}_{\text{source}}$, 则将 $\mu'.p$ 的污染传递给 *res*, 并将其标记为输入污染变量. 在后续处理 $\mathbb{E}_{\text{source}}$ 时, 对该情况进行特殊处理.

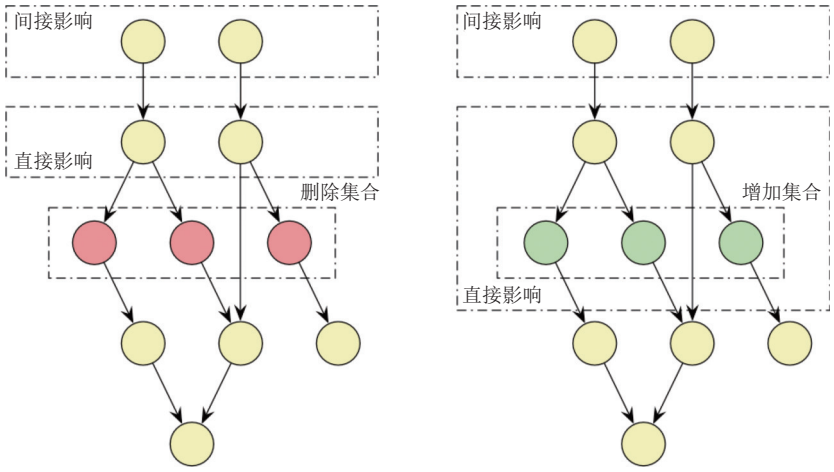
(3) 汇聚点封装信息 $\mathbb{E}_{\text{sink}}$ 应用: 若 $s_{m'}$ 中包含 $\mathbb{E}_{\text{sink}}$, 且其依赖的输入变量被污染, 则将相应结果添加至当前函数摘要的 $\mathbb{E}_{\text{sink}}$.

2.3 面向 Web 漏洞检测的函数摘要

2.3.1 函数摘要完整性分析

本方法提出的函数摘要生成算法, 能够生成对所有上下文适用的通用摘要, 在任意调用场景中均可保持分析

有效性. 通过符号化建模实现了变量污染状态的等价类划分, 将具有相同数据传播特性的变量集合划分为等价类, 并基于此类划分生成规范化摘要信息. 这种结构保证了函数摘要在不同调用上下文中的语义一致性, 使得单次分析结果能够被全局复用. 基于此, 由于一次展开被调函数分析生成了适用所有上下文的函数摘要, 因此对于完整函数摘要方法, 在考虑增量变化影响时, 只需要考虑沿着调用关系的反方向传播 (向上传播), 如图 5 所示.



(a) 删除增量变化影响　　(b) 新增增量变化影响

图 5　本方法的增量影响范围图例

2.3.2　函数摘要增量生成过程

在版本控制系统中, 项目源码的每次变化都代码的增加、删除和修改这 3 种基本情况组成. 为了能更好地进行统一处理, 我们将其描述为两个列表: 删除代码列表和增加代码列表. 代码的修改可以拆分为先删除文件和再增加文件两个动作. 处理增量变化时, 先处理删除代码列表再处理增加代码列表. 算法 2 展示了增量计算的过程, 其输入包括上一轮全部的函数摘要集合 $\mathcal{S}$, 删除代码列表 $\mathbb{L}_{\text{del}}$、新增代码列表 $\mathbb{L}_{\text{add}}$ 以及上一轮分析的中间结果, 核心内容包含抽象语法树、类型信息以及调用图.

算法 2. 增量摘要更新算法 *updateSummary*.

输入: $\mathcal{S}$, $\mathbb{L}_{\text{del}}$, $\mathbb{L}_{\text{add}}$, CG, AST;
输出: $\mathcal{S}'$.

1. $F_{\text{del}} \leftarrow GetFunc(\mathbb{L}_{\text{del}})$
2. $F'_{\text{del}} \leftarrow ForwardReach(F_{\text{del}}, CG)$
3. $RemoveSymbols(F_{\text{del}})$
4. $AST_{\text{add}} \leftarrow BuildAST(\mathbb{L}_{\text{add}})$
5. $F_{\text{add}} \leftarrow ExtractFunctions(AST_{\text{add}})$
6. $CG' \leftarrow CG$
7. $F'_{\text{add}} \leftarrow ForwardReach(F_{\text{add}}, CG')$
8. $F'_{\text{affect}} \leftarrow F'_{\text{del}} \cup F'_{\text{add}}$
9. $\tau_F \leftarrow ReverseTopOrder(F'_{\text{affect}}, CG')$
10. **for** $f \in \tau_F$ **do**
11. **if** $NeedVisit(f)$ **then**
12. $\mathcal{S}'(f) \leftarrow BuildSummary(f)$

13. **if** $\mathcal{S}'(f) \neq \mathcal{S}(f)$ **then**
14. $MarkVisit(Callers(f, CG'))$
15. **end**
16. **end**
17. **end**
18. **return** $\mathcal{S}'$

增量分析算法的第 1 步是对删除代码进行分析. 我们将每个行号 $line \in \mathbb{L}_{del}$ 解析为对应的函数集合 F_{del}. 在此基础上, 为捕捉删除操作对程序语义的全局影响, 本文基于原始调用图执行反向依赖传播: 通过深度优先遍历从 F_{del} 的节点出发, 识别所有直接或间接调用其的函数节点, 构成扩展受影响集合 F'_{del}. 随后, 从全局符号表中移除 F_{add} 对应的抽象语法树 (*AST*) 及类型信息, 确保后续分析不被残留状态干扰. 对于在新增文件列表中的每个函数 $f \in F_{add}$, 我们将以函数为单位, 重新生成 *AST* 树与过程内类型信息, 从而将其解析为新增函数集合 F'_{add}. 随后, 我们将根据当前信息, 增量生成调用图, 考虑重新连接的调用点包括变化文件中的所有类函数, 以及其在旧调用图中的父亲节点. 特别的, 如果修改的类为基类或接口, 我们还会将其所有子类或实现类对应的函数考虑在内. 在生成调用图的过程中,函数调用图中不包含反射调用, 并将对所有多态调用进行连边. 根据新的调用图, 计算所有直接调用或间接调用 F_{add} 函数集合 F'_{add}, 合并为组织 F'_{add} 与 F'_{del}, 记为 F'_{affect}. 为组织 F'_{affect} 的复杂依赖关系, 我们对 F'_{affect} 进行逆拓扑序的排序, 从而形成逆拓扑序列 τ_F. 该序列隐含调用链的层次关系结构: 若 f_i 调用 f_j, 则在 τ_F 中 f_j 一定在 f_i 前方. 对于每个 $f \in \tau_F$, 我们将调用 *buildSummary*() 构建函数摘要, 如果其函数摘要 s 发生了改变, 则需要将其调用者的函数摘要也需要重新计算中, 如果其函数摘要 s 并没有发生改变, 则不需要考虑重新计算, 即增量影响仅停止于当前函数. 我们将迭代计算, 直到遍历所有 τ_F 中的函数. 最终产出当前版本的全部函数摘要集合 $\mathcal{S}'$.

3 实　现

3.1 分析工具实现

基于上述方法, 本文实现了一个面向 Java Web 应用漏洞检测的增量静态分析框架 AWTaint. 该框架基于 JavaCC 库[38]实现 Java 语法解析内核, 该工具支持通过形式化语法生成确定性有限自动机 (deterministic finite automaton, DFA) 解析器, 为后续的符号表构建与类型推导奠定可靠基础. 通过定制 Java 的语法规则文件, AWTaint 可精准捕获方法体边界, 变量作用域及类型声明等关键信息, 为后续过程间分析提供结构化输入. 针对过程间分析需求, 本框架扩展了跨函数调用链信息收集机制, 建立包含包导入关系, 类继承结构和方法重写关系的全局符号表.

为提升中间结果持久化的效率与可靠性, 本系统采用 Kryo 序列化框架[39]实现中间结果的高效导入导出. 该框架通过 ASM 字节码操作技术动态生成序列化逻辑, 在处理复杂对象图时性能优势更为突出. 其内置的循环引用检测机制可自动维护对象间的关系网络, 确保包含多层嵌套结构的符号表与调用图能够完整重建, 且能够减少冗余存储.

3.2 分析工具实现

在 Web 应用安全检测领域, 普遍做法是采用基于启发式策略的污点分析方法来定义规则[29,30,32,40,41], 将漏洞检测模型抽象为三元组形式: (source, sink, sanitizer), 每一个三元组将面向一种特定漏洞. 该建模方法通过形式化定义信息流传播路径中的关键节点, 实现对攻击者可控参数在程序控制流的传递进行系统性追踪. 具体建模策略如下.

(1) 污染源 (source) 建模: 在 Web 应用安全场景中, 污点源特指能够引入不可信数据或敏感信息的程序入口点. 根据 OWASP Top 10 安全规范, 本研究将用户可控输入 (如 HTTP 请求参数、Cookie 字段等) 作为主要污点源. 通过静态分析框架对主流 Web 开发框架 (包括 Spring、Struts 等) 的方法签名进行模式匹配, 建立覆盖 MVC 架构控制器层的污点源规则库. 例如, 我们将针对 Servlet API 的 Http ServletRequest.getParameter() 系列方法, Spring 框架的@RequestBody 注解方法等典型数据接收接口标记为污点源.

(2) 净化器 (sanitizer) 建模: 对于净化函数, 通过模式匹配识别潜在数据净化操作 (如输入验证正则表达式、数据加密函数等), 随后结合数据流传播路径的上下文敏感分析验证净化操作的有效性. 例如, 针对 XSS 漏洞, 我们对 Spring 框架中的 HtmlUtils.htmlEscape 方法进行标记, 从而清除该函数返回参数的污点状态.

(3) 汇聚点 (sink) 建模: 对于汇聚点, 通过模式匹配识别潜在的风险函数操作 (如 SQL 注入函数、命令注入函数等), 并指定在特定位置参数被污染的情况下, 才能够上报漏洞.

3.3 自定义污点传播

在 Java Web 分析中, 存在诸多较难用传统静态污点分析覆盖的逻辑. 针对这些问题, 我们在 AWTaint 中加入了部分自定义的污点传播策略, 在一定程度上缓解静态分析方法在 Java Web 中的断边问题. AWTaint 中采用的自定义污点传播逻辑可以概括为如下 3 个方面.

(1) 跨线程污点传播建模: 本框架将显式建模 Thread 类及其子类的生命周期方法调用链, 强制建立 start() 方法与 run() 执行体之间的跨方法数据流映射. 从而确保在 ExecutorService 等线程池场景下仍能保持污点传播.

(2) 数据库污点传播建模: 本框架按照 JDBC 规范的核心接口构造特定的污点传播逻辑. 如对于 PreparedStatement() 的参数绑定方法, 建立参数位置与污染标签的关联映射, 精确追踪各参数值在动态 SQL 中的传播路径, 对于利用 ORM 框架的 SQL 查询场景, 本框架将建模查询语句 *AST* 节点与参数对象的污染传播映射与类型映射. 如对 MyBatis 的 XML 映射文件, 将建立语句结构与参数占位符的关联关系, 并识别形如${param}的占位符因直接拼接导致的注入风险.

(3) 依赖注入传播建模: 针对 Spring 框架依赖注入导致的污点断边问题, 本框架通过建模 Bean 间依赖关系 (包括注解/XML 注入、AOP 代理调用、配置类工厂方法), 建立跨 Bean 的污点传播链, 确保污染源经 setter/构造器注入后仍能沿业务方法传播, 例如当 User Controller 通过@Autowired 接收用户输入参数并注入 UserService 时, 系统能识别污染数据经 Spring 容器从 controllerParam→userService.userDao→SQL 查询的完整传播路径, 而非在注入点断开.

3.4 额外增量影响范围考虑

对于修改在函数外的情况, 本框架会有相对保守的增量影响范围估计方案. 对于类字段的修改, 本框架会额外考虑该类字段在应用中所有的出现语句, 并重新计算该语句所在的函数摘要. 对于类名与包名的修改 (如 class A→class B), 我们会将所有对该类的引用 (包括继承、实例化、类型声明等) 标记为潜在影响范围. 对于类字段的修改 (如 private String a→private String b), 我们会将所有对该字段的引用 (包括当前类方法内的使用, 子类方法内的使用, setter 以及 getter 等) 标记为潜在影响范围. 针对这些潜在的影响范围, 我们都会通过符号表更新与跨类依赖分析, 动态重构调用图与类继承关系树.

除此之外, 我们还对部分配置文件进行了建模, 支持除了 Java 文件修改之外, 针对主流配置文件的修改. 如对 MyBatis 的 XML 进行建模, 通过 SQL 语句与 Java 方法的绑定关系解析, 可以将针对 XML 的修改 (如新增字段映射或调整查询条件) 反映到具体的 SQL 执行点上, 从而扩展额外的增量影响范围. 又如对 Springboot Bean 配置文件进行了建模, 当配置文件发生修改时 (如新增 Bean 定义、调整注入参数等), 框架会触发额外的增量分析逻辑, 确保配置驱动的逻辑不被遗漏.

3.5 行号偏移校正策略

在增量分析过程中, 需要将上一次扫描的摘要等中间结果进行导入. 为了让中间结果在两次分析之间都可以使用, 除了工作路径等元数据信息的修改之外, 更加需要考虑的是行号相关的问题. 即便源代码仅进行结构修改 (如头部空白行插入), 此类微小变更也将触发函数作用域内所有行号偏移, 进而导致基于行号标记的函数摘要产生版本间差异. 这种差异虽不影响函数内部控制流分析的语义一致性, 但在组装缺陷报告与完整跟踪图的过程中, 会导致与源文件的映射产生偏差. 本框架将基于稳定语法锚点的行号偏移进行. 具体而言, 本框架通过识别源代码中较稳定性的语法结构 (如函数声明符、类定义标识等) 来处理行号的偏移, 并在缺陷跟踪路径组装过程中实现源码位置映射的精确还原. 类似技术方案在工业级静态分析工具 Coverity 的增量分析模块中亦有应用验证[15].

4 实验评估

实验评估主要回答以下 3 个研究问题.

(1) RQ1: AWTaint 的基础分析能力如何? 在污点分析能力上是否优于现有工具?

(2) RQ2: AWTaint 的分析效率如何? 增量分析模式相较于全量分析模式有哪些方面的效率提升? 是否会引入额外开销?

(3) RQ3: AWTaint 在全量分析模式与增量分析模式下的检测结果是否一致?

4.1 实验设置

● 实验环境: 所有实验在一台配备 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.

● 基线工具: 本研究综合考虑业界各类方法的分析场景支持能力, 选取以下 3 款具有代表性的 Web 静态应用安全测试工具作为比较对象.

(1) CodeQL[27]: 作为当前最主流的静态分析工具, 利用快速指针分析来实现精确的污点分析. 其研究团队曾提出基于差分 Datalog 的增量分析框架[14], 可以为 CodeQL 支持增量分析. 然而, 由于其实现细节未公开且缺乏官方文档支持, 我们未能成功复现该增量分析模式, 我们选用的 CodeQL 版本为 v2.19.0.

(2) Coverity[42]: 作为当前较为流行的商业工具已实现增量分析架构[15], 但其增量分析功能并不支持 Web 应用的扫描[43]. 鉴于本实验对象为 Java Web 应用项目, 因此无法利用其增量分析模式对比数据, 我们选用的 Coverity 版本为 2024.12.1.

(3) Tai-e[44]: 作为目前学术界较为热门的 Java 静态分析工具, 采用极具创新性的指针分析技术提升准确性和效率. 其系统整合了 Soot、WALA 等经典框架, 具备高效性、易用性和高度可扩展性. 在实验中, 采用 2-obj 的指针分析设置, 即堆对象的上下文采用两层对象敏感策略, 我们选用的版本为 0.5.1.

此外, 诸多学术界的工具也实现了增量分析功能, 旨在提升静态分析的效率与可扩展性. 然而, 在实际应用中, 这些工具难以有效迁移到 Web 应用漏洞检测领域. 例如, INCRELUX[24]的研究以及其他相关工作主要聚焦于单一类型的 C/C++语言漏洞检测, 其函数摘要机制的设计高度依赖于底层内存模型, 难以适配 Web 应用特有的分析模式. 类似地, Facebook 开发的 Infer[45]虽声称支持 Java 应用的增量分析, 其函数摘要面向内存类型漏洞设计, 包含大量约束谓词信息. 通常来说函数摘要越复杂, 那么判断两个函数摘要是否等价也会越困难. 如果直接使用文本比较那么微小的变化就会引起函数摘要的大规模连锁失效. 此外其增量分析功能在多个独立评估中被指出存在显著的实现缺陷, 其会错误地对部分函数进行重复分析, 还会跳过本应重新分析的部分函数, 导致分析结果不可靠且难以复现, 从而无法作为严谨的对比基准[46]. 因此, 当前学术界与工业界缺乏针对 Web 应用漏洞检测场景的成熟增量分析工具, 其现有成果或局限于特定语言与漏洞类型, 或在工程实现上存在实质性缺陷. 这导致我们难以找到功能完备, 可复现的基准系统, 与 AWTaint 在增量分析性能, 精度及适用性方面进行系统化对比评估.

● 实验基准集: 主要包括 3 个数据集, 包括 Java Web 应用数据集、xAST[47]和 OWASP-BenchMark[48].

(1) Java Web 应用数据集: 本研究所构建的 Java Web 应用数据集中除了考虑传统开源社区的代码仓库外, 还融合了工业级生产环境的真实案例. 该数据集由两个维度构成: 其一为来自 GitHub 平台的 5 个高活跃度开源 Web 应用项目, 其二为来自大型企业的 5 个闭源工业级应用程序. 项目筛选采用多维度标准: 通过“java”“web”“springboot”等技术栈关键词进行初始检索, 继而依据代码提交频率 (大于 1 次/周) 和 Star 收藏量 (大于 1 500) 等活跃度指标进行二次筛选, 最终形成兼具技术代表性和行业覆盖度的实验数据集 (详见表 2). Java Web 应用数据集涵盖了各类代码量级的 Java Web 应用, 规模范围从 23k 到 923k 代码行, 有效覆盖了实际开发中的项目量级. 所有实验均基于 Git 主分支开展增量式代码分析, 时间窗口设定为从表中记录的起始提交点开始的连续 21 次 commit 历史 (将合并不包含 Java 文件变更的 commit), 共计 210 个 commit. 在本数据集中, 实际开发中的单次提交代码变更呈现显著的局部性特征 (平均变更量 255 行, 中位数 215 行), 其中 50% 的变更量低于 28 行. 统计检验表明, 单次变更量与项目总规模未发现相关性 (Pearson r=0.03, p=0.66), 该数据分布验证了变更代码的数量与项目规

模没有关联性, 因此仅针对变更代码的分析可以有效减少分析复杂度. 在实际应用中, 可能针对的一次 PR 或一次发布, 这种情况下可以将若干个 commit 进行合并, 从而完成增量分析.

表 2 Java Web 应用数据集

Project	init	Freq	*LoC* (k)	Δ*LoC* (avg/max)	Δ*file* (avg/max)	Δ*func* (avg/max)
Stirling-PDF	5b0ea	1 d 1 h	36	376/4 568	21/241	65/965
eladmin	75df4	1 h 16 h	23	66/363	21/254	66/896
xwiki-platform	e7d08	11 h	923	36/217	2/6	6/34
JeecgBoot	ec9f2	1 d 19 h	95	198/1 101	6/31	10/57
RuoYi	9a833	5 d 17 h	37	20/136	3/11	14/64
Application-1	5827a	14 h	272	357/1 467	6/21	39/209
Application-2	08248	4 d 21 h	687	343/2 949	5/12	48/239
Application-3	bb8b0	9 h 1 h	320	882/2 155	55/179	292/1 266
Application-4	a041b	6 h 1 h	420	231/578	18/29	69/134
Application-5	25e84	6 d 15 h	49	44/380	3/12	4/13

注: 行号、函数、文件数变化仅标注Java 文件

(2) xAST[47]: 本研究利用 xAST 数据集评估本方法的基础分析能力. 不同于以漏洞类型为评价视角的传统漏洞样本集, xAST 在同一套漏洞样本集上进行测试, 并从引擎精度能力维度与引擎实现完整度对白盒分析引擎进行测试. 在引擎能力维度上, xAST 包含流敏感、对象敏感、路径敏感和上下文敏感这 4 个子维度. 在引擎实现完整度上, xAST 包含异步能力支持, Java 表达式语句支持, 以及 Java 复合类型支持 3 个子维度. 我们选取了 xAST 在 2024 年 11 月的版本进行测试, 仅考虑其白盒分析引擎的测试样例 (即 sast-java 目录下的测试样例), 共包含 332 个漏洞测试样例, 包括 228 个正样本和 104 个负样本.

(3) OWASP-BenchMark[48]: 本研究利用知名度较高的 Java Web 安全漏洞测试用例集合 OWASP-BenchMark 评估本方法对各种漏洞类型的覆盖率. 我们选取了其 v1.2 版本进行测试, 在测试的过程中, 去除了代码质量相关的类目, 如明文密码 (CWE-327)、弱哈希算法 (CWE-327) 等, 仅保留 Web 应用漏洞相关的 7 个测试类目, 共包含 1 740 个漏洞测试样例, 包括 902 个正样本和 838 个负样本.

● 规则设置: 在本实验中, 共构建了 13 项针对典型污点分析风格漏洞的检测规则, 如表 3 所示. 这些规则覆盖了包括 SQL 注入 (CWE-89), 命令注入 (CWE-78), 跨站脚本攻击 (XSS, CWE-79), 路径遍历 (CWE-22), XPath 注入 (CWE-643) 等多种安全威胁, 涵盖了 OWASP Top 10[2]中的主要注入类, 数据泄露类和集成风险类问题. 所有的规则采用相同的污点源 (Sources) 建模 Web 应用入口分析策略, 部分规则包含特定净化器 (Sanitizers) 建模与自定义污点传播.

表 3 13 类典型污点分析风格漏洞以及对应的汇点 (Sinks) 样例

ID	Vulnerablity name	CWE	Sinks example
1	Path traversal	CWE-22	java.nio.file.Files.read()
2	Command injection	CWE-78	java.lang.Runtime.exec()
3	Cross-site scripting (XSS)	CWE-79	javax.servlet.http.HttpServletResponse.getWriter.write()
4	SQL injection (SQLI)	CWE-89	java.sql.Statement.executeQuery()
5	LDAP injection	CWE-90	javax.naming.Context.lookup()
6	Code injection	CWE-94	javax.script.ScriptEngine.eval()
7	Trust boundary violation	CWE-501	javax.servlet.http.HttpSession.setAttribute()
8	Deserialization of untrusted data	CWE-502	java.io.ObjectInputStream.readObject()
9	Open redirect	CWE-601	javax.servlet.http.HttpServletResponse.sendRedirect()
10	XML external entity (XXE)	CWE-611	javax.xml.parsers.DocumentBuilder.parse()
11	XPATH injection	CWE-643	javax.servlet.http.HttpSession.setAttribute()
12	Groovy code injection	CWE-693	groovy.lang.GroovyShell.evaluate()
13	Server-side request forgery (SSRF)	CWE-918	java.net.http.HttpClient.send()

4.2 RQ1: 静态分析能力评估

我们首先从引擎的基础分析能力与对不同 Web 漏洞类型的支持能力两个维度系统评估 AWTaint 的静态分析能力. 在基础分析能力验证中, 采用 xAST 基准集进行测试: 针对 AWTaint 启用全量分析模式并应用表 3 中定义的命令注入漏洞检测规则 (对应第 5 列规则集), 同时为基线工具配置其官方提供的命令注入规则, 并严格遵循 xAST 文档要求扩充 sink 与 source 配置以确保测试环境一致性. 在漏洞类型支持能力验证中, 则利用 OWASP-BenchMark 基准集全面考察工具对主流 Web 漏洞的覆盖度, 其中 AWTaint 在全量分析模式下, 集成表 3 预定义的 7 条规则以覆盖基准集全部漏洞类型, 而 CodeQL 与 Coverity 直接采用官方 CWE 对应规则且未作修改, Tai-e 则因官方规则库缺失目标 CWE, 需将基准集明确定义的 source 与 sink 注入其规则系统以实现完整覆盖. 针对 4 种工具对这两个基准集的检测结果, 通过人工抽样 10% 的检测结果进行二次校验, 以确保检测结果真实反映引擎的静态分析能力而非规则实现差异.

4.2.1 引擎基础能力对比

如表 4 所示, 其中最优性能用加粗表示. 在 xAST 基准集上对 AWTaint 与 CodeQL、Coverity、Tai-e 等静态分析工具的系统性对比表明, AWTaint 在整体漏洞检测性能上展现出显著优势, 尤其在召回率与精确率的平衡方面表现突出. 具体而言, AWTaint 以 66.98% 的召回率位居首位, 不仅显著优于 CodeQL (53.51%) 和 Tai-e (62.72%), 并略微超越 Coverity (66.23%), 同时, 其精确率 (88.57%) 虽略低于 Coverity (90.96%), 但明显高于 CodeQL (87.77%) 和 Tai-e (85.12%), 表明 AWTaint 在维持高检测广度的同时有效控制了误报率. 从分析时长来看, AWTaint 以 51 s 的平均分析时间显著优于其他工具, 展现出卓越的分析效率. 相较于 CodeQL (64 s) 和 Coverity (145 s), AWTaint 不仅在性能上更具优势, 且远优于 Tai-e 所需的 1 643 s.

表 4 xAST 基准集上, AWTaint 与基线工具的对比

Metrics	AWTaint			CodeQL			Coverity			Tai-e		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
flow-senstitive	6	3	1	7	4	0	7	2	0	4	5	3
object-sensitive	2	0	0	2	0	0	2	0	0	0	0	2
path-sensitive	9	5	3	6	3	6	7	7	5	5	4	7
field-sensitive	23	10	18	13	8	28	29	2	12	5	5	36
context-sensitive	5	2	0	5	2	0	4	2	1	5	5	0
async-feature-support	5	0	0	0	0	5	1	0	4	0	0	5
statement-support	49	0	29	35	0	43	40	2	38	57	3	21
component-type-support	56	0	22	54	0	24	61	0	17	67	3	11
Total	155	20	73	122	17	106	151	15	77	143	25	85
Precision (%)	88.57			87.77			**90.96**			85.12		
Recall (%)	**67.98**			53.51			66.23			62.72		
Analysis time (s)	**51**			64			145			1 643		

● 流敏感能力: 我们发现 CodeQL 与 Tai-e 仅部分支持流敏感, 分析过程中未考虑 break 和 continue 控制流中断语句; 而 AWTaint 支持流敏感且考虑控制流中断. 但在流敏感的样例中, 包含需要对变量具体值进行追踪来决定流执行分支的情况, 如 for 循环或 while 循环条件为假时提前退出. AWTaint 的分析策略是循环体仅展开一次, 无法处理此类复杂情况. 相比之下, CodeQL 和 Coverity 则可以处理这类情况.

● 路径敏感能力: 所有工具均无法支持根据变量的情况判断分支, 因此在这类路径敏感的样例中表现不佳. 相比之下, AWTaint 对数组, 结构体等对象存在较好的域敏感建模, 因此可以更准确地捕捉分支内的污点传播情况. 而其余基线工具无法做到这一点. 因此在路径敏感的测试样例中 AWTaint 有着最高的表现.

● 域敏感能力: 在容器类的处理上, CodeQL 未区分 map 容器中不同键值的对象 (统一处理为 ⟨map.value⟩), 而 AWTaint 与 Coverity, 一样可以对常量的 map 容器对象进行处理, 可以从一定程度上缓解该问题. 此外, CodeQL 与 Coverity 对数组长度有一定的检测手段, 会避免将存在数组越界的变量设为污染, 而 AWTaint 并不存在这样的

处理. 在复杂对象属性的处理上, 所有基线工具均表现不佳, 主要原因在于几乎所有用例中都包含了不同形式的别名关系, 在未使用指针分析算法的情况下, 准确处理污点传播的难度较大. 比如将二叉树类型的变量进行重赋值后, 再进行访问.

● 上下文敏感能力: 我们发现 AWTaint 与其他工具发生误报的情况相同. 主要是因为上下文信息的局限性所致, 诸如污点对象为数组的情况下部分污染, 而在函数内, 该结果取决于非污染变量具体取值. 在分析中, 我们并不会考虑变量的具体取值, 只会考虑变量的污点分析状态.

● 引擎实现完整度: 我们发现 AWTaint 能够支持所有异步特征, 此外, AWTaint 对 Java 表达式语句支持最全面, 而其余基线工具在这方面的支持较差. 然而, AWTaint 对 Java 复合类型的支持没有 Coverity 完善, 从而进一步导致在总体精准度上略低于 Coverity, 经过人工分析, 这些漏报主要是因为 AWTaint 并没有对复合类型操作建模不全面且不精确, 而 Coverity 存在更加完善的内置 Java 函数污点传播逻辑, 且对数组下标, 复合变量元素有更精确的识别, 因此在该维度优于 AWTaint.

4.2.2 规则支持能力对比

如表 5 所示, 在 OWASP-BenchMark 基准集上对 AWTaint 与主流静态分析工具的系统性评估表明, AWTaint 在 Web 应用漏洞类型支持方面有显著优势, 尤其在关键安全指标上实现了理想平衡. 具体而言, AWTaint 以 100% 的召回率 (Recall) 位居首位, 显著优于 Tai-e (75.06%), 还超越了 Coverity (98.78%) 和 CodeQL (95.23%); 同时, 其精确率 (Precision) 达到 86.23%, 在基线工具中同样领先 (CodeQL 为 73.11%, Coverity 为 69.12%, Tai-e 为 60.34%). 从分析时长来看, AWTaint 仅需 73 s 即可完成整个 OWASP-BenchMark 基准集的分析, 显著优于其他对比工具, 展现出极高的分析效率. 相比之下, CodeQL 耗时 208 s, 约为 AWTaint 的 2.85 倍; Coverity 高达 1 680 s (28 min), 是 AWTaint 的 23 倍; 而 Tai-e 的分析时间长达 8 705 s (145 min), 性能瓶颈明显.

表 5 在 OWASP-BenchMark 基准集上, AWTaint 与基线工具的对比

Metrics	AWTaint			CodeQL			Coverity			Tai-e		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
CWE-22	133	22	0	133	66	0	133	75	0	109	92	24
CWE-78	126	20	0	83	29	43	115	67	11	98	80	28
CWE-79	246	48	0	246	90	0	246	68	0	194	87	52
CWE-89	272	36	0	272	87	0	272	128	0	193	121	79
CWE-90	27	4	0	27	13	0	27	15	0	19	22	8
CWE-501	83	12	0	83	24	0	83	30	0	49	29	34
CWE-643	15	2	0	15	7	0	15	15	0	15	14	0
Total	902	144	0	859	316	43	891	398	11	677	445	225
Precision (%)		**86.23**			73.11			69.12			60.34	
Recall (%)		**100**			95.2			98.78			75.06	
Analysis time (s)		**73**			208			1 680			8 705	

这种性能优势源于 AWTaint 针对 Web 应用漏洞特性的深度优化设计. 例如, 在 CWE-89 (SQL 注入) 的测试类目中, AWTaint 比 CodeQL 减少了 58.6% 的误报, 比 Coverity 减少了 71.9% 的误报. 误报显著降低的原因在于 AWTaint 实现了更加精细的数据库查询语句污染建模机制, 不仅判断 SQL 语句整体是否被污染, 还精确追踪污染参数在查询语句中的具体位置 (如 WHERE 子句或 VALUES 子句), 从而有效过滤了非漏洞场景的误报. 此外, 在 CWE-79 (XSS) 等关键测试类目中, AWTaint 同样展现出系统性优势, 其误报率显著低于 CodeQL (90 例)、Coverity (68 例) 和 Tai-e (87 例), 这主要归功于其对 Web 应用中常见字符串处理操作 (如 HTML 编码、正则表达式替换等) 的细粒度建模, 使字符串类型相关的污点传递路径分析更加精确, 避免了因粗粒度字符串操作处理导致的误报累积, 进一步强化了在复杂 Web 环境中的检测可靠性.

4.2.3 实验小结

实验表明, 在 xAST 上, AWTaint 的召回率为 67.98%, 领先基线 1.75 个百分点; 在 OWASP-BenchMark 上 (针

对其覆盖的漏洞类型), 召回率达到 100%. 在这两个测试集上 AWTaint 的扫描时间也是最短的. AWTaint 能在兼顾检测准确性与召回率的同时显著提升分析速度, 关键在于本文提出的面向 Web 应用安全的函数摘要机制. 该机制使每个函数只需分析一次, 即可生成适用于所有调用上下文的域敏感函数摘要. 与此同时, 该种函数摘要可与抽象语义建立等价映射关系, 从而能够帮助精确计算代码变更的影响范围, 因此也能应用于增量扫描场景中.

4.3 RQ2: 分析效率

为对比 AWTaint 在增量分析模式以及全量分析模式下的分析效率. 首先, 我们将利用全量分析框架对表 2 中的初始版本进行分析, 并将中间结果进行导出. 该步骤为准备步骤, 不列入统计范围. 随后, 针对后续连续 20 个 commit, 我们将会进行全量分析与增量分析, 并统计扫描在每个阶段的结果. 在本节, 我们将讨论增量分析的效率以及影响增量分析效率的因素.

4.3.1 增量分析效率

表 6 展示了针对 10 个 Web 应用, 在连续 20 个 commit 提交中, 增量分析与全量分析各个阶段的对比. 为了进行直观对比, 我们省略在扫描结束后对中间结果进行导出的阶段, 中间结果导出为一个可选的异步任务, 其导出时机位于扫描结束后. 在分析过程中, 本框架内存峰值不超过 8 GB. 实验结果表明, 在引入增量分析后, 加速比平均可达 3.63 倍. 增量分析的加速效果本质上取决于变更代码占项目总规模的比例. 本数据集中, xwiki-platform (923k *LoC*) 虽为最大项目, 但其单次变更比例最低 (0.039 行/k *LoC*), 因此获得最高加速比 (5.12 倍); 而 RuoYi (37k *LoC*) 为相对较小项目, 其变更比例较高 (0.54 行/k *LoC*), 加速比降至 2.26 倍.

表 6 不同分析阶段的时间

Target	Full scan average time (ms)				Increment scan average time (ms)			
	T_{all}	T_{basi}	$T_{summary}$	T_{rule}	$T_{all}\times S_p$	$T_{inc.}$	$T_{summary}$	T_{rule}
Stirling-PDF	25 696	6 171	13 904	5 621	7778×3.3	5 371	1 595	812
eladmin	11 041	2 443	5 394	3 204	4509×2.45	2 737	1 078	694
xwiki-platform	379 237	59 060	165 837	154 340	74119×5.12	35 424	2 375	36 319
JeecgBoot	34 648	5 968	18 261	10 419	10079×3.44	7 973	686	1 419
RuoYi	11 559	2 435	5 766	3 358	5106×2.26	4 039	607	460
Application-1	78 360	15 218	51 958	11 184	19035×4.12	13 029	2 538	3 468
Application-2	152 971	31 382	74 321	47 269	42421×3.61	33 809	2 086	6 526
Application-3	86 224	16 903	46 304	23 017	23687×3.64	15 874	4 523	3 290
Application-4	131 079	22 612	72 239	36 229	26682×4.91	18 572	4 268	3 842
Application-5	15 507	3 367	7 338	4 803	4444×3.49	3 389	582	474

为进一步量化单次变更比例与增量分析加速比的关系, 我们对 eladmin 的样本 (N=20) 进行了线性回归分析, 检验变更代码行数对增量分析加速比的预测作用. 结果表明, 变更代码行数对加速比有较显著负向预测作用, 变更代码行数每多 1 行, 增量分析加速比平均下降 0.002 7 (β=−0.0027, SE=0.0008, p<0.005). 该关系呈中等程度负相关 (Pearson r=−0.629, R=0.396, p<0.001).

此外, 我们对变更行数在 10–30 行的样本 (N=97) 进行了线性回归分析, 检验项目规模 (k *LoC*) 对增量分析加速比的预测作用. 结果表明, 项目规模对加速比有显著正向预测作用, 项目规模每增加 1k *LoC*, 增量分析加速比平均提高 0.002 3 (β=0.0023, SE=3.0185, p<0.001), 该关系呈强正相关 (Pearson r=0.650, R^2=0.422, p<0.001). 上述统计分析表明, 在变更量相同的情况下, 项目规模越大, 变更所占比例越小, 增量分析的加速比越高, 优化效果越为明显. 该结果验证了增量分析在大规模系统中面对局部化变更时具有更显著的性能收益.

4.3.2 增量分析效率

尽管增量分析对低变更密度应用 (如 xwiki-platform, 0.039 行/k *LoC*) 展现出显著加速效果 (5.12 倍), 但其实际加速比受多重因素制约, 其最终的加速效果取决于最各个阶段的效率提升. 为系统地量化影响因素, 我们从 4 个关键维度展开分析: (1) 中间数据导入开销; (2) 摘要增量生成效率; (3) 扫描入口重分析比例; (4) 漏洞报告合并成本. 具体数据如表 7 所示.

表 7　对增量分析效率产生影响的因素

Project	Data import		Summary regen.		Entrance recalc.		Vuln.
	Basic-data *t* (ms)/*s* (MB)	Rule-data *t* (ms)/*s* (MB)	#affect (Est.)	#affect (Changed)	#entrance (All)	#entrance (Inc.)	#num
Stirling-PDF	2 410/5.78	726/1.83	110.90	65.45	319.45	67.15	3.90
eladmin	1 112/3.1	494/1.48	123.80	9.55	277.15	41.70	28.00
xwiki-platform	9 525/123.77	10 844/73.64	780.20	2.15	6 428.45	221.10	40.00
JeecgBoot	2 353/14.19	1 140/8.17	36.40	10.90	818.10	137.05	12.35
RuoYi	1 186/4.63	576/2.55	71.30	1.10	278.45	50.45	1.00
Application-1	3 257/47.11	2 595/18.73	148.35	17.00	1 755.30	17.00	24.00
Application-2	5 836/97.48	12 856/61.07	124.65	27.65	2 253.10	134.15	7.00
Application-3	3 794/48.36	3 662/26.58	576.45	128.10	2 466.65	123.15	4.00
Application-4	5 398/79.3	4 625/41.83	1 269.70	68.35	4 845.05	446.00	8.00
Application-5	1 361/7.74	675/3.25	14.30	1.10	482.15	21.85	1.00

增量分析阶段替代基础分析阶段, 对中间结果数据导入, 变化数据处理以及增量影响范围处理. 增量分析阶段对比全量分析的基础分析阶段, 该提升并不明显, 如 eladmin、JeecgBoot 等个别代码量较小的应用中出现了负提升. 主要是因为该步骤需要从文件存储中导入中间结果数据, 并将其反序列化到内存中. 尽管我们选择了 Kyro 这一反序列化效率较高的三方库, 但其依旧需要花费比较多的时长. 在表 7 中, 统计了中间结果导入所需的时长, 其中基础数据 (Basic-data) 列代指 *AST*, 符号表等与规则无关数据, 函数摘要数据 (Rule-data) 列代函数摘要, 漏洞追踪图等与规则相关数据. 结果显示, 中间结果的导入时间平均占总体增量分析阶段 54.07%, 是增量分析阶段的主要额外开销. 我们对所有样本 (N=200) 进行了线性回归分析, 检验项目代码量对中间结果导入时长的预测作用. 结果表明, 项目代码量每提高 1 000 行, 中间结果导入时长平均上升 0.022 s (β=0.0220, SE=1.1452, p<0.001). 该关系呈现强正相关 (Pearson r=−0.965, R^2=0.9329, p<0.001). 因此, 项目代码量对中间结果导入时长有显著的正向预测作用另外, 我们以 String-PDF 为例, 依次对其 20 次代码提交应用 1–13 条检测规则. 我们对检测结果 (N=260) 进行了线性回归分析, 检验规则数量对中间结果导入时长的预测作用. 结果表明, 检测规则数量每多 1 条, 中间结果导入平均时长平均上升 0.073 1 s (β=0.0731, SE=1.0779, p<0.001). 该关系呈现强正相关 (Pearson r=0.4671, R^2=0.4671, p<0.001). 因此, 规则数量对中间结果导入时长也有显著的正向预测作用.

从存储开销角度考虑, 对于万行代码而言, AWTaint 中间结果的基础分析部分数据, 约为 1.52 MB. 中间结果中的函数摘要数据, 对于单条规则而言, 约为 0.06 MB. 因此, 在本实验设置中, 万行代码的中间结果大小为 2.27 MB. 虽然我们未能复现 CodeQL 的增量分析能力, 但可以通过 CodeQL 的中间结果估算其存储开销, 针对上述 Web 应用, CodeQL 的万行代码平均中间结果大小为 79.4 MB, 为 AWTaint 的 35 倍. 得益于较低的存储成本, 中间结果大小是可以适配真实工业化场景的.

增量分析的摘要分析阶段利用算法 1 生成部分函数摘要. 我们首先会根据调用图关系, 估算出一个较大的函数修改范围 Affect: 即表 7 中的#affect (Est.) 列, 它代表了图 6 中的所有节点. 表 7 中的#affect (Changed) 列展示了实际发生变化的函数摘要, 它代表了图 6 中的右侧部分. 在该列数据中, Application-4 存在着一个极端值, 我们观察到 Application-4 虽然变化的平均函数仅为 69 个, 但是根据调用图与类继承图估算出的影响范围平均约为 1 269.70 个, 这是因为 Application-4 中存在 8 个 commit 直接修改了函数名或类字段, 在这种情况下, 将会触发我们的保守计算逻辑. 比如 Application-4 的某个 commit 中, 由于将接口类的名称进行重命名, 我们会将该接口类所有实现类的方法均纳入影响范围中, 共计影响到了超过 1.5k 的函数摘要. 然而, 这种类型的修改是比较少见的, 我们评估了 Application-2 中包含的 210 次代码变更, 共计发现了 19 次 (9.04%) 代码变更包含对类名的修改. 对于 Application-4 而言, 实际情况下, 只有平均 68.35 个函数的函数摘要发生变化. 结果显示, 摘要分析阶段相较于全量分析阶段, 平均加速比为 22 倍. 提速最高的应用为 eladmin, 主要是因为该应用的体量较大, 共包含 38k 个函数, 而我们仅为平均 780.2 个函数重新计算了函数摘要 (2.05%), 因此可达到 70 倍的加速比.

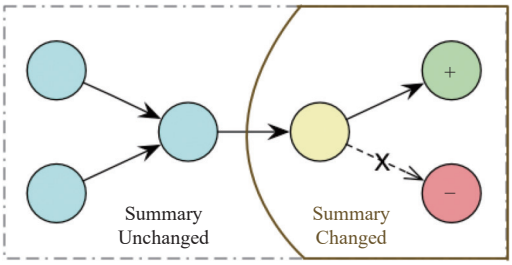


图 6　增量影响范围划分

增量分析的规则分析阶段不再以全部的入口作为分析起点, 而是从变化的函数摘要向上搜索, 查找变化的入口点. 在表 7 中, 我们统计了导致入口点的实际分析情况. 列#entrance (All) 代表了全量分析的入口, 而列#entrance (Inc.) 代表了从变化的函数摘要出发, 影响到的分析入口. 这是规则扫描阶段, 增量分析显著比全量分析快的原因. 从这两列数据的对比中, 我们发现某些情况下, 变化的函数摘要与需要重新分析的入口之间并不存在线性关系. 如 RuoYi 应用虽然仅平均变化了 1.1 个函数摘要, 却需要为平均 50.45 个入口进行扫描, 经过人工审查分析, 我们发现在若干个相近的 commit 中, 开发者对一个通用的工具类函数进行了多次修改, 且该函数被普遍应用于多个数据处理类中, 从而引起了较大的范围的重新分析. 而 Application-1 应用的函数摘要变化数量与规则入口变化数量几乎相同, 人工审查发现开发者在多个 commit 中对 Request 处理接口有频繁的修改, 该接口也被引擎处理为了分析入口.

除了增量入口之外, 漏洞数量的变化可能也需要纳入考虑的因素. 如果入口发生变化, 我们会删除从该入口到 sink 的污点跟踪图, 并重新生成这部分路径. 如果漏洞对应的 sink 点被删除, 这我们会将该漏洞完全删除, 并根据规则扫描的结果重新生成该漏洞对应的污点跟踪图. 然而, 漏洞相对于整体应用而言相对稀疏. 在增量分析阶段, 然而我们仅发现 Stirling-PDF 与 JeecgBoot 在这 20 个 commit 区间内, 发生了漏洞数量或污点跟踪图的变化. 以初始版本为例, AWTaint 共为这 10 个应用检出合计 128 个漏洞, 其中 SQL 注入 (CWE-89) 占比最高, 达 81 个 (63.28%), 其余检出漏洞包含命令注入 (CWE-78) 8 个, SSRF (CWE-918) 漏洞 15 个, 开放重定向 (CWE-601) 漏洞 16 个, 任意文件读 (CWE-22) 漏洞 2 个, 以及 XML 外部实体注入 (CWE-611) 漏洞 6 个. 在后续版本的扫描中, Stirling-PDF 的漏洞数量从 3 个变更到了 5 个. JeecgBoot 的漏洞数量从 12 个变更到了 13 个. 尽管这两个应用在漏洞引入发生前后, 增量分析时长并无明显差别, 但在实际情况下, 我们依然认为这部分时间是可能会影响增量分析的效率, 尤其是当污点追踪图深度较深, 或大范围的修改引起大量漏洞重新生成的情况. 我们将会在第 4.4 节中进一步讨论漏洞一致性问题.

4.3.3　实验小结

实验表明, AWTaint 在真实 Web 应用增量分析中, 平均加速比达 3.63 倍 (最高 5.12 倍), 万行代码中间结果存储开销仅 1.52 MB (基础数据) 与 0.06 MB (单规则摘要), 仅为 CodeQL 的 1/50, 其高效且稳定的表现与较低存储成本为工业级 CI/CD 安全扫描提供了高效可行的解决方案.

4.4　RQ3: 漏洞结果一致性评估

在第 2 节中, 我们强调了本方法提出的函数摘要具有完整性, 且考虑增量分析影响范围的时候, 采取了保守的手法. 通过对静态分析能力的评估, 从实验数据上证明了本方法在实现上不存在问题. 在本实验中, 我们将重点讨论全量分析与增量分析的结果一致性. 结果的一致性并不仅表现在漏洞数量上, 还需要表现在漏洞跟踪图的一致性上, 即增量分析的漏洞跟踪图与全量分析完全一致.

对于表 2 中的数据, 选取了 Stirling-PDF 与 JeecgBoot 的所有版本审查了漏洞结果, 因为这两个应用存在漏洞数量以及跟踪图的变化, 除此之外, 对于剩余的 8 个应用, 我们抽样检查了 10% 次分析的漏洞以及跟踪图. 所有结果均一致. 然而, 在第 4.3 节中, 我们发现漏洞变化相对于整体代码是相对稀疏的, 对此, 还对 xAST 进行了增量,

共选取了 10 个版本 (2014 年 9 月–2025 年 3 月版本) 进行测试, 测试结果如表 8 所示. 通过人工审查分析结果, 我们发现增量分析与全量分析的结果一致. 因此, 暂未发现由增量扫描引入的额外误报率与漏报率.

表 8　xAST 的 10 次 commit 中, AWTaint 增量分析与全量分析的结果

Commit	Δ*file*	Δ*LoC*	Δ*func*	#Vuln (Full/Inc.)	#Affect (Est./Ch.)	#Entrance (All/Inc.)
e03e	—	—	—	168/—	—/—	448/—
4749	335	5692	352	183/183	658/658	553/286
e43c	15	175	29	183/183	29/15	549/429
8fc5	3	650	4	188/188	10/1	565/429
62ec	16	16	18	188/188	18/2	565/445
25bc	12	60	12	188/188	12/0	565/445
4668	4	11	4	188/188	8/1	565/445
75ac	4	9	4	188/188	6/4	565/445
e3ba	486	4037	822	153/153	1025/335	517/377
611c	3	2	3	153/153	5/4	521/381

5 讨　论

5.1 通用性

本研究所提出的框架原型虽基于 Java 语言实现, 但其核心设计范式具有跨语言适配性. 在函数级摘要生成层面, 该方法通过抽象语法树的遍历实现程序语义表征, 此机制可推广至任意具备标准化 AST 解析能力的编程语言. 以基于 Node.js 语言的 Web 应用为例, 可通过 V8 引擎内置 AST 解析接口可以获取结构化语法信息, 进而函数摘要生成与增量分析. 为确保中间数据处理效率, 系统采用基于 Kryo 框架[39]的二进制序列化方案, 然而, 其他语言可能在中间结果存储时遇到 I/O 瓶颈问题. 在增量影响范围分析模块, 调用图的完整性直接影响增量分析的准确性: 对于因间接调用解析失效或跨语言边界导致的调用边缺失, 可能引发影响范围的误判.

此外, 本文以 Web 应用漏洞检测为典型应用场景, 规则建模围绕恶意载荷在单次程序控制流中的直接传递而进行设计, 函数摘要结构与增量分析策略均围绕该场景特征进行特化设计. 在 Web 应用程序的请求-响应架构中, 恶意载荷还可借助 HTTP 会话, 数据库持久化, 线程共享数据等中间介质在不同执行过程间传播. 针对此类问题, 多段数据流分析是一种常见解决方案, 其核心思想在于识别并处理 Web 应用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”), 从而追踪共享数据上的污染读写操作, 并通过关联与迭代算法捕捉不同执行路径之间的污点数据流关联. 本文所设计的函数摘要机制也能够自然地扩展以支持共享数据的污染读写行为: 待迭代算法收敛后将共享数据上的污染读取视为额外 source 封装信息 (Esource), 将共享数据上的污染写入视为额外的输出污染变量或者特殊的 sink 封装信息 (Esink). 此外, 本框架在迁移到内存漏洞检测领域时, 由于未建立显式的堆内存模型, 因此本方案无法检测如空指针引用, 未初始化变量访问, 内存泄露等内存安全漏洞. 此类漏洞检测需要引入堆抽象机制, 并增强路径敏感分析能力.

5.2 有效性威胁

本研究的有效性存在 3 个主要威胁: (1) 尽管我们的测试集已经涵盖 GitHub 热门项目以及工业界频繁更新项目, 但实验基于有限数量的 Java Web 项目 (共 10 个) 和代码提交记录 (210 次), 可能影响结论的普适性. (2) 中间结果存储与读取, 采用文件存储结构, 可能会导致效率问题. 实验数据显示, 增量分析阶段 54.07% 的时间消耗在中间结果导入, 成为主要性能瓶颈. 尽管 Kryo 库已经在文件存储中足够高效, 但依旧可能会限制大规模应用的扩展性. 后续工作可以探讨以更高效的存储结构 (如增量更新索引) 来解决这一性能瓶颈. (3) 增量影响范围分析采用保守策略, 可能导致非必要代码的重复分析. 实验数据显示, 在 Application-4 项目中, 接口类重命名触发超过 1.5k 函数摘要重计算 (实际变更仅 68.35 个函数); 工具类函数的多次修改 (如通用数据处理工具) 引发平均 50.45 个入口点重新扫描. 这种保守策略旨在绝对保证“结果一致性” (即增量分析结果与全量分析完全一致), 但代价是重新分

析了部分未受影响的代码. 尽管我们通过“函数摘要是否改变来决定入口点重新扫描”的机制规避了部分额外开销, 但当前方法仍未达到理论最精简的增量计算状态. 后续工作可引入轻量级变更传播分析技术 (如依赖切片), 以进一步减少此类非必要分析带来的额外时间开销, 或根据使用场景采用更加激进的策略, 从而达到理论最精简的增量计算状态.

6 相关工作

在本节中, 主要通过 3 个方面来介绍与讨论增量分析框架的相关工作: (1) 增量分析算法; (2) 函数摘要算法; (3) Web 应用漏洞检测.

6.1 增量分析算法

由于能够响应代码变更快速提供最新分析结果的优势, 近年来增量分析在静态分析领域受到了极大的关注. 针对各种静态分析技术, 研究人员分别提出了相应的增量化方案, 这些方案均具有非常广泛的应用场景, 包括终止分析[49]、漏洞检测[50]、代码克隆检测[51]、程序验证等[52].

大多数基于指针的增量分析算法[12,13]都演化自 DRed (deletion-rederivation) 算法[53]. 然而, 这类方法在动态方法调用变化的精确处理上存在不足. 针对 DRed 算法存在的不足, Liu 等人[13]提出了 IPA 算法, 该算法通过观察 Java 程序指针分配图 (pointer allocation graph, PAG), 利用其变化局部性来改进 DRed 算法. SHARP[12]是对 IPA 算法的进一步改进, 通过引入上下文敏感机制来扩展 IPA 算法, 从而减少在删除方法调用时引入的冗余性. 然而, 这类分析算法冗余计算依然较为严重, 缺乏扩展性. 相比之下, 本文方法冗余计算的情况可以通过精确的函数摘要变化范围进行缓解. 除了 DRed 算法外, 也有基于 CFL 可达性的增量指针分析算法. 如 Shang 等人[16]提出了 EMU 系统, 使开发者能够在集成开发环境 (integrated development environment, IDE) 中对持续变化的程序执行指针相关查询. 然而, EMU 假设程序已具备预构建的调用图, 这一前提在代码变更可能修改调用图本身的场景中并不成立. CodeQL[27]是一个基于指针分析的静态代码扫描工具, 近期通过更换后端 Datalog 引擎[14], 实现了一个支持增量分析的原型框架, 然而其算法层面仅能提供基础指针分析能力, 因此对上下文敏感分析的精度缺乏. DDoop[17,18]一定程度上克服了 CodeQL 增量框架的问题, 实现了增量输入事实生成技术与增量分析规则自动化重写技术, 将多版本程序增量分析问题表达为差分 Datalog 评估问题. 然而其依旧在分析效率和内存占用上存在明显短板, 使其难以扩展到真实工业场景. 此外, 还有一些高效的增量指针分析算法[54,55], 然而, 此类算法仅能够处理代码添加, 无法处理代码删除, 使其难以应用到真实的 DevOps 场景. 相比之下, 我们的方法能够同时处理增删操作, 并在真实代码提交场景中验证了有效性.

除了面向指针的增量分析算法之外, 还有诸多研究者设计了面向图的增量分析算法. IncA[20]提出基于图模式匹配的领域特定语言 (domain specific language, DSL) 实现程序分析结果的增量更新. 在其基础上, IncA$_L$[19]增加了数据合成功能, 实现了区间分析的增量执行. Reviser[21]在 IFDS/IDE 框架中, 采用“重置-重新计算 (reset-recompute)”策略: 先收集变更节点及其影响节点, 再通过“清除-传播 (clear-and-propagate)”策略重置并重新计算这些节点的 IDE 结果. Zhao 等人[56]提出基于 CHA 的增量调用图构建算法. 其依赖已有的类层次结构, 对复杂更新处理的效率较低, 该框架仅涉及调用图重计算, 并不涉及漏洞检测. 基于图计算的增量分析算法在面对真实场景中的大型代码项目时, 不可避免地会遇到复杂度爆炸问题. 相比之下, 我们的方法即便是针对真实工业场景的项目, 均在可控的复杂度内.

针对增量指针分析与增量图分析的性能与效率短板, 有研究者提出了基于函数摘要的增量分析算法[22−25,45], 该类算法的核心是通过更新函数摘要和调用图实现分析结果增量. Krishnan 等人[25]提出了一种基于前后向摘要的增量分析方法, 然而他们的方法仅分析变更的代码, 未考虑变更代码的间接影响, 可能会导致漏洞漏报, 导致增量分析结果与全量分析产生不一致. Jana 等人[22,23]的方法优先收集编辑过的函数集合, 直接重新计算其摘要, 若新摘要与旧版本不同, 则递归重新计算调用者的摘要. 该方法假设调用链不变, 对于会改变调用图的变更工具会触发彻底分析而不是增量分析. 可能会因为某些修改情况导致增量分析效率严重下降. 相比之下, 本文方法可以处理调用

图的变更情况, 也会考虑变更代码的间接影响, 且通过函数摘要的变化范围精确标明了需要重新分析的入口, 是一种兼顾效率与一致性的折中策略.

Calcagno 等人[45]设计了一种支持组合式的形状分析的增量静态分析工具 Infer, 该工具目前在 Facebook 广泛使用. 该工具的函数摘要是一个三元组集合, 每个三元组包含前置约束 (函数执行前堆的形状约束), 函数名与参数, 以及后置约束 (函数执行后堆的变化结果). 相比之下, 本文提出的函数摘要和程序污染传播抽象语义能够建立一对一的等价映射, 而 Infer 的函数摘要包含很多和污染传播无关的信息 (约束谓词) 和程序污染传播抽象语义是多对一的关系. 函数摘要发生了变化, 但是程序污染传播抽象语义不一定发生改变. 在实际中, Infer 会因为许多和污点分析无关的极小修改而引起大范围的重新扫描. 而本文提出的函数摘要结构则不会导致该问题. 此外, Zhai 等人[24]设计了面向 Linux 内核 UBI 缺陷检测的增量静态分析工具 INCRELUX. 该工具的函数摘要设计是一种面向 Linux 内核未初始化变量 (use-before-Initialization, UBI) 漏洞的特化, 其函数摘要记录了函数安全调用所需的输入状态以及输出状态变化, 其重点关注内存对象的初始化状态和使用情况. 相比之下, 对比本文提出的函数摘要结构聚焦在入口变量与返回变量之间的直接关系, 并需要根据规则配置的不同, 记录污点封装信息, 因此可以多类的 Web 应用安全漏洞进行检测. 由于聚焦的问题不同, 函数摘要记录的信息不同, 因此, 文献 [24,45] 方法难以迁移到本文的方法中.

6.2 函数摘要

函数摘要是一种通用的程序分析技术, 通过提取函数的高层行为特征 (如输入输出关系、副作用、调用约束等), 为代码分析提供轻量级, 可重用的抽象表示[33]. 它在安全分析领域具有重要价值, 广泛应用于代码克隆检测[57], 漏洞分析[58,59]等领域.

有一类的函数摘要是为了在相同分析过程中对组合分析的结果进行缓存而设计的. 这类摘要只有函数在相同分析中被多次调用时才能使用, 无法通过简单的方式来进行持久化存储, 也不支持在不同轮次的分析中被重复使用. 如 Dillig 等人[60]设计了针对堆操作程序进行验证的精确且紧凑的模块化过程摘要方法, 其摘要用于加速调用点处对堆位置的更新. Babic 等人[61]提出了 Calysto, 该工具通过路径敏感, 上下文敏感和位准确的数据操作建模, 实现了与昂贵形式化分析相当的覆盖率和精确度. Calysto 的函数摘要共享公共子表达式, 从而减少符号执行的复杂度. Dahse 等人[62]提出了 RIPS. 该工具首先建立控制流图, 然后通过模拟每个基本块的数据流来创建函数摘要, 从而进行精确的污点分析. RIPS 使用了一种符号执行来建立函数摘要, 因此其效率堪忧, 较难迁移到大型项目中. Wang 等人[59]提出了 ANTaint, 该工具的函数摘要面向数据泄露问题, 通过解决隐式依赖问题来提升工业级微服务应用的静态污点分析可扩展性, 其依赖于精确且完整的调用图以及自定义的污点传播模型. 针对 ANTaint 存在的问题, Zhong 等人[58]提出了 CFTaint, 该工具使用的是一种非流敏感, 非对象敏感, 基于字段的分析方法, 其函数摘要是专门针对特定敏感数据泄露问题而设计的, 如果用于处理注入类的场景, 则其分析精度远远不能满足需求. 相比之下本文方法是一种流敏感、域敏感、对象敏感和上下文敏感的函数摘要表示, 可用于处理包括多种场景多种 Web 应用分析场景.

另一类函数是为了在不同分析过程中重用分析结果而设计, 除了用于增量扫描之外, 还常用于跨模块与跨组件的分析中[63−66]. StubDroid[63]通过自动生成库代码的精确污点传播摘要, 提高 Android 应用污点分析的性能和精度, 其生成函数摘要需要较大的时间与内存开销. Tang 等人[64]提出了一种基于树连接语言 (tree-structured assembly language, TAL) 可达性的库代码摘要方法, 以加速带回调的客户代码的上下文敏感数据依赖性分析, 同样, 其算法在处理大规模程序时可能遇到的性能瓶颈和复杂度问题. Li 等人[66]提出了 SPATA, 该工具通过构建函数的别名摘要, 进行跨函数的类型状态跟踪和约束检查, 提高了大型操作系统代码的静态漏洞检测准确性和可扩展性. 其函数摘要建模了函数的类型信息与路径约束信息, 因此可以用于检测空指针引用, 未初始化变量访问, 内存泄露等内存安全漏洞. 其函数摘要本质上是一种有限状态机 (finite state machine, FSM) 的建模, 并不支持面向 Web 的污点分析. Wu 等人[65]提出了 LibAlchemy, 该工具通过该系统通过基于程序依赖图以及数据流路径的双层数据摘要设计有效应对第三方库的误用问题 (如空指针引用, 使用已释放内存等问题). 其函数摘要建模了函数内

的控制/数据流依赖关系以及关键数据流路径, 是面向三方库误用问题的抽象方案. 由于本文提出的函数摘要是对单个函数的完整语义分析, 因此本文的函数摘要也可以应用于跨模块与跨组件的分析中, 但需要借助软件成分分析 (software component analysis, SCA) 来解析当前应用所导入的组件与模块, 从而完成函数调用点到三方模块/组件的调用连边. 另外, 上述函数摘要的相关工作与 AWTaint 均表明, “合理且有效的”函数摘要设计需要充分利用分析目标本身的特点.

6.3 Web 应用漏洞检测

10 几年来, 针对 Web 应用漏洞检测一直是程序分析领域的热门研究方向. 部分工作聚焦于特定漏洞的分析, 如聚焦在 XSS 漏洞[67−69]、SQL 注入漏洞[70−72]、文件上传漏洞[73,74]、反序列化漏洞[75,76]和越权漏洞[77]等. Pixy[67]利用格算法与有界模型检查 PHP 应用程序中的跨站脚本漏洞 (XSS). 也有研究者采用符号执行[68]或模糊测试[69]的方法来检测 XSS 漏洞. SQLCIV[70]可以用于检测 SQL 注入漏洞, 而 Dahse 等人[71]与乐德广等人[72]的工作可以用于检测二次 SQL 注入漏洞. UChecker[73]采用静态分析与符号执行结合的方法进行检测对于文件上传漏洞, 而 FUSE[74]采用静态分析的方法模拟文件上传请求, 并利用模糊测试的方法来绕过净化器. 通常这类工作针对特定漏洞类型有具体的分析设计, 难以扩展到多种漏洞类型的检测上, 也难以进行函数摘要收集与增量化改造.

也有一些工作与本文一样, 致力于提供一个面向多种漏洞的 Web 应用漏洞静态分析框架. Tan 等人[44]提出的 Tai-e 框架实现了针对 Java 应用的高效指针分析算法, 可在降低复杂度的同时实现更准确的分析结果. Tai-e 可用于在 Java Web 应用中进行漏洞检测. 然而, 作为指针分析框架, Tai-e 同样面临分析粒度与资源消耗的矛盾, 难以应用于本文的场景. 基于代码属性图 (code property graph, CPG) 的 Web 应用漏洞分析框架[28−32]通过将程序的语法, 控制流, 数据流等语义信息统一建模为属性图结构, 为多类型漏洞检测提供通用分析基础. 但是 CPG 的构建依赖对程序的完整分析而非按需分析, 导致真实工业场景下的计算开销难以控制, 此外, 对 CPG 进行存储与导入导出也需要较高成本.

7 总结与展望

本文提出了一种面向 Web 应用漏洞检测的增量静态分析框架 AWTaint. 旨在对多种类型的 Web 应用漏洞进行增量分析, 并兼顾精度, 效率与一致性. 本文通过轻量级函数摘要机制与细粒度增量计算方法, 在 Web 应用漏洞检测中实现了全量分析与增量分析的统一范式. 实验表明, 在包含 10 个真实 Java Web 应用 (5 个开源+5 个工业闭源项目, 共 210 次提交) 的测试集上, AWTaint 相较全量分析平均加速 3.63 倍, 内存峰值控制在 8 GB 以内, 能够检测多种类型的漏洞, 且漏洞检测结果完全一致, 其中在超 90 万行代码的项目中加速比达 5.12 倍, 验证了其工业级适用性. 研究成果为安全左移策略落地提供了高效可行的技术路径, 不仅减少了安全检测的计算资源消耗, 还显著加速了开发环节的安全响应速度.

本文的方法和框架依然存在一些不足, 这也是我们未来的研究方向: 针对内存安全漏洞检测, 当前框架缺乏对堆内存抽象模型和路径敏感状态追踪的支持, 未来计划显式建模内存对象的生命周期与访问模式, 结合路径敏感的地址状态约束求解, 实现对内存安全漏洞检测的增量分析; 此外, 我们为了确保结果的一致性, 仍旧存在部分冗余计算问题, 计划融合程序切片算法与字段级依赖分析进一步减少增量计算范围. 在跨语言适配方面, 我们计划将 AWTaint 扩展到解释型语言中, 以提高其泛用性; 另外, 针对跨应用跨组件漏洞的问题, 计划结合软件成分分析 (SCA) 技术实现第三方依赖的漏洞传播路径建模与增量分析支持. 这些研究方向将系统性提升框架在工业复杂场景下的实用性与泛化能力.

References

[1] CrowdStrike. CrowdStrike 2024 global threat report. 2024. https://www.crowdstrike.com/en-us/resources/reports/crowdstrike-2024-global-threat-report/

[2] OWASP Foundation. OWASP Top 10—2023 Release. 2023. https://owasp.org/Top10/en/

[3] Yahoo! Inc. Announcement on the large-scale user data breach incident. 2016 (in Chinese). https://www.yahoo.com/news/yahoo-

announces-data-security-incident-192515010.html

[4] U.S. House of Representatives Committee on Oversight and Government Reform. Equifax data breach: Exposure of 148 million consumers' sensitive information. 2018. https://oversight.house.gov/hearing-record/115th-congress-2017-2018/equifax-data-breach-exposure-of-148-million-consumers-sensitive

[5] SlowMist Security Team. In-depth analysis report on the Bybit hack incident involving nearly $1.5 billion. 2025 (in Chinese). https://www.freebuf.com/articles/blockchain-articles/bybit-hack-analysis

[6] Gartner. Gartner forecasts global information security spending to grow 15% in 2025. 2024. https://www.gartner.com/en/newsroom/press-releases/2024-08-28-gartner-forecasts-global-information-security-spending-to-grow-15-percent-in-2025

[7] Gartner. Gartner identifies the top cybersecurity trends for 2025. 2025. https://www.gartner.com/en/newsroom/press-releases/2025-03-03-gartner-identifiesthe-top-cybersecurity-trends-for-2025

[8] Liu BH, Zhang H, Dong LM. Survey on state of DevOps in China. Ruan Jian Xue Bao/Journal of Software, 2019, 30(10): 3206–3226 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5796.htm [doi: 10.13328/j.cnki.jos.005796]

[9] Christakis M, Bird C. What developers want and need from program analysis: An empirical study. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: ACM, 2016. 332–343. [doi: 10.1145/2970276.2970347]

[10] Ami AS, Moran K, Poshyvanyk D, Nadkarni A. "False negative - that one is going to kill you": Understanding industry perspectives of static analysis based security testing. In: Proc. of the 2024 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2024. 3979–3997. [doi: 10.1109/SP54263.2024.00019]

[11] Sadowski C, Van Gogh J, Jaspan C, Soderberg E, Winter C. Tricorder: Building a program analysis ecosystem. In: Proc. of the 37th IEEE/ACM IEEE Int'l Conf. on Software Engineering. Florence: IEEE, 2015. 598–608. [doi: 10.1109/ICSE.2015.76]

[12] Liu BZ, Huang J. SHARP: Fast incremental context-sensitive pointer analysis for Java. Proc. of the ACM on Programming Languages, 2022, 6(OOPSLA1): 88. [doi: 10.1145/3527332]

[13] Liu BZ, Huang J, Rauchwerger L. Rethinking incremental and parallel pointer analysis. ACM Trans. on Programming Languages and Systems, 2019, 41(1): 6. [doi: 10.1145/3293606]

[14] Szabó T. Incrementalizing production CodeQL analyses. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 1716–1726. [doi: 10.1145/3611643.3613860]

[15] McPeak S, Gros CH, Ramanathan MK. Scalable and incremental software bug detection. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering. Saint Petersburg: ACM, 2013. 554–564. [doi: 10.1145/2491411.2501854]

[16] Shang L, Lu Y, Xue JL. Fast and precise points-to analysis with incremental CFL-reachability summarisation: Preliminary experience. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering. Essen: IEEE, 2012. 270–273. [doi: 10.1145/2351676.2351720]

[17] Wang XZ, Shen TQ, Bin XR, Bu L. LLM-powered datalog code translation and incremental program analysis framework. Ruan Jian Xue Bao/Journal of Software, 2025, 36(6): 2515–2534 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7330.htm [doi: 10.13328/j.cnki.jos.007330]

[18] Shen TQ, Wang XZ, Bin XR, Bu L. DDoop: Incremental pointer analysis framework based on differential datalog evaluation. Ruan Jian Xue Bao/Journal of Software, 2024, 35(6): 2608–2630 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7100.htm [doi: 10.13328/j.cnki.jos.007100]

[19] Szabó T, Völter M, Erdweg S. IncA_L: A DSL for incremental program analysis with lattices. In: Proc. of the 31st European Conf. on Object-oriented Programming (ECOOP). ACM, 2017.

[20] Szabó T, Erdweg S, Voelter M. IncA: A DSL for the definition of incremental program analyses. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: ACM, 2016. 320–331. [doi: 10.1145/2970276.2970298]

[21] Arzt S, Bodden E. Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In: Proc. of the 36th Int'l Conf. on Software Engineering. Hyderabad: ACM, 2014. 288–298. [doi: 10.1145/2568225.2568243]

[22] Jana A, Khadsare A, Chimdyalwar B, Kumar S, Ghime V, Venkatesh R. Fast change-based alarm reporting for evolving software systems. In: Proc. of the 32nd IEEE Int'l Symp. on Software Reliability Engineering (ISSRE). Wuhan: IEEE, 2021. 546–556. [doi: 10.1109/ISSRE52982.2021.00062]

[23] Jana A, Chimdyalwar B, Kumar S, Venkatesh R. Fast analysis of evolving software systems. In: Proc. of the 2022 IEEE Int'l Symp. on Software Reliability Engineering Workshops (ISSREW). Charlotte: IEEE, 2022. 49–54. [doi: 10.1109/ISSREW55968.2022.00038]

[24] Zhai YZ, Hao Y, Zhang Z, Chen WT, Li GR, Qian ZY, Song CY, Sridharan M, Krishnamurthy SV, Jaeger T, Yu P. Progressive scrutiny: Incremental detection of UBI bugs in the Linux kernel. In: Proc. of the 2022 Network and Distributed Systems Security Symp. San Diego, 2022. [doi: 10.14722/ndss.2022.24380]

[25] Krishnan P, O'Donoghue R, Allen N, Lu Y. Commit-time incremental analysis. In: Proc. of the 8th ACM SIGPLAN Int'l Workshop on State of the Art in Program Analysis. Phoenix: ACM, 2019. 26–31. [doi: 10.1145/3315568.3329968]

[26] Li KX, Chen S, Fan LL, Feng RT, Liu H, Liu CW, Liu Y, Chen YX. Comparison and evaluation on static application security testing (SAST) tools for Java. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 921–933. [doi: 10.1145/3611643.3616262]

[27] GitHub Security Lab. CodeQL. 2023. https://github.com/github/codeql

[28] 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]

[29] 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 (EuroS&P). Paris: IEEE, 2017. 334–349. [doi: 10.1109/EuroSP.2017.14]

[30] Alhuzali A, Gjomemo R, Eshete B, Venkatakrishnan VN. NAVEX: Precise and scalable exploit generation for dynamic Web applications. In: Proc. of the 27th USENIX Security Symp. Baltimore: USENIX Association, 2018. 377–392.

[31] Chen XC, Wang BZ, Zhang MJ, Cao YQ, Liu QX. VulKiller: Java Web vulnerability detection with code property graph and large language models. In: Proc. of the 2025 IEEE Int'l Conf. on Acoustics, Speech and Signal Processing (ICASSP). Hyderabad: IEEE, 2025. 1–5. [doi: 10.1109/ICASSP49660.2025.10890652]

[32] Chen XC, Wang BZ, Jin Z, Feng Y, Li XL, Feng XC, Liu QX. Tabby: Automated gadget chain detection for Java deserialization vulnerabilities. In: Proc. of the 53rd Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN). Porto: IEEE, 2023. 179–192. [doi: 10.1109/DSN58367.2023.00028]

[33] Aho AV, Lam MS, Sethi R, Ullman JD. Compilers: Principles, Techniques, and Tools. 2nd ed., Boston: Addison-Wesley Longman Publishing Co. Inc., 2006.

[34] 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. In: Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Edinburgh: ACM, 2014. 259–269. [doi: 10.1145/2594291.2594299]

[35] Lerch J, Späth J, Bodden E, Mezini M. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (T). In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Lincoln: IEEE, 2015. 619–629. [doi: 10.1109/ASE.2015.9]

[36] Arroyo M, Chiotta F, Bavera F. An user configurable clang static analyzer taint checker. In: Proc. of the 35th Int'l Conf. of the Chilean Computer Science Society (SCCC). Valparaiso: IEEE, 2016. 1–12. [doi: 10.1109/SCCC.2016.7835996]

[37] Bodden E. The secret sauce in efficient and precise static analysis: The beauty of distributive, summary-based static analyses (and how to master them). In: Proc. of the 2018 ISSTA/ECOOP Workshops. Amsterdam: ACM, 2018. 85–93. [doi: 10.1145/3236454.3236500]

[38] Contributors J. JavaCC: Java compiler compiler. 2023. https://javacc.github.io/javacc/

[39] EsotericSoftware. Kryo: Fast, efficient Java serialization. 2023. https://github.com/EsotericSoftware/kryo

[40] Papagiannis I, Migliavacca M, Pietzuch P. PHP Aspis: Using partial taint tracking to protect against injection attacks. In: Proc. of the 2nd USENIX Conf. on Web Application Development (WebApps). Portland: USENIX, 2011.

[41] 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]

[42] Synopsys Inc. Coverity static analysis. 2023. https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast/coverity.html

[43] Synopsys, Inc. Coverity analysis: Incremental analysis. 2024. https://documentation.blackduck.com/bundle/coverity-docs/page/coverity-analysis/topics/incremental_analysis.html

[44] 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]

[45] Calcagno C, Distefano D, O'Hearn P, Yang H. Compositional shape analysis by means of bi-abduction. In: Proc. of the 36th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Savannah: ACM, 2009. 289–300. [doi: 10.1145/1480881.1480917]

[46] Beránek T. Practical application of Facebook infer on systems code [BS. Thesis]. Brno: Brno University of Technology, 2021.

[47] Ant Group Security Team, Zhejiang University Network Security College. Ant application security testing benchmark (xAST). 2024. https://github.com/alipay/ant-application-security-testing-benchmark

[48] The OWASP® Foundation. OWASP Benchmark v1.2. 2025. https://owasp.org/www-project-benchmark/

[49] He F, Han JT. Termination analysis for evolving programs: An incremental approach by reusing certified modules. Proc. of the ACM on Programming Languages, 2020, 4(OOPSLA): 199. [doi: 10.1145/3428267]

[50] Dura A, Reichenbach C, Söderberg E. JavaDL: Automatically incrementalizing Java bug pattern detection. Proc. of the ACM on Programming Languages, 2021, 5(OOPSLA): 165. [doi: 10.1145/3485542]

[51] Nguyen TT, Nguyen HA, Al-Kofahi JM, Pham NH, Nguyen TN. Scalable and incremental clone detection for evolving software. In: Proc. of the 2009 IEEE Int'l Conf. on Software Maintenance. Edmonton: IEEE, 2009. 491–494. [doi: 10.1109/ICSM.2009.5306283]

[52] Yu QS, He F, Wang BY. Incremental predicate analysis for regression verification. Proc. of the ACM on Programming Languages, 2020, 4(OOPSLA): 184. [doi: 10.1145/3428252]

[53] Gupta A, Mumick IS, Subrahmanian VS. Maintaining views incrementally. ACM SIGMOD Record, 1993, 22(2): 157–166. [doi: 10.1145/170036.170066]

[54] Krainz J, Philippsen M. Diff graphs for a fast incremental pointer analysis. In: Proc. of the 12th Workshop on Implementation, Compilation, Optimization of Object-oriented Languages, Programs and Systems. Barcelona: ACM, 2017. 4. [doi: 10.1145/3098572.3098578]

[55] Whaley J, Lam MS. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proc. of the 2004 ACM SIGPLAN Conf. on Programming Language Design and Implementation. Washington: ACM, 2004. 131–144. [doi: 10.1145/996841.996859]

[56] Zhao ZL, Wang XZ, Xu ZG, Tang ZH, Li YC, Di P. Incremental call graph construction in industrial practice. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP). Melbourne: IEEE, 2023. 471–482. [doi: 10.1109/ICSE-SEIP58684.2023.00048]

[57] Sneha SG, Niha S, Hasan DMM. Evaluating code clone detection and management: A comprehensive comparison among different techniques and tools along with some effective future directions. In: Proc. of the 3rd Int'l Conf. on Computing Advancements. Dhaka: ACM, 2025. 207–215. [doi: 10.1145/3723178.3723206]

[58] Zhong ZX, Liu JC, Wu DY, Di P, Sui YL, 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 (ICSE-SEIP). Melbourne: IEEE, 2023. 110–121. [doi: 10.1109/ICSE-SEIP58684.2023.00015]

[59] 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 European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 1477–1486. [doi: 10.1145/3368089.3417059]

[60] Dillig I, Dillig T, Aiken A, Sagiv M. Precise and compact modular procedure summaries for heap manipulating programs. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Jose: ACM, 2011. 567–577. [doi: 10.1145/1993498.1993565]

[61] Babic D, Hu AJ. Calysto: Scalable and precise extended static checking. In: Proc. of the 30th Int'l Conf. on Software Engineering. Leipzig: ACM, 2008. 211–220. [doi: 10.1145/1368088.1368118]

[62] Dahse J, Holz T. Simulation of Built-in PHP features for precise static code analysis. In: Proc. of the 2014 Network and Distributed System Security Symp. San Diego, 2014. 23–26.

[63] Arzt S, Bodden E. StubDroid: Automatic inference of precise data-flow summaries for the Android framework. In: Proc. of the 38th Int'l Conf. on Software Engineering. Austin: ACM, 2016. 725–735. [doi: 10.1145/2884781.2884816]

[64] Tang H, Wang XY, Zhang LM, Xie B, Zhang L, Mei H. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In: Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Mumbai: ACM, 2015. 83–95. [doi: 10.1145/2676726.2676997]

[65] Wu RX, He YX, Huang JF, Wang CP, Tang WS, Shi QK, Xiao X, Zhang C. LibAlchemy: A two-layer persistent summary design for taming third-party libraries in static bug-finding systems. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 105. [doi: 10.1145/3597503.3639132]

[66] Li T, Bai JJ, Sui YL, Hu SM. SPATA: Effective OS bug detection with summary-based, alias-aware, and path-sensitive typestate analysis. ACM Trans. on Computer Systems, 2024, 42(3–4): 9. [doi: 10.1145/3695250]

[67] 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/Oakland: IEEE, 2006. [doi: 10.1109/SP.2006.29]

[68] Lekies S, Kotowicz K, Groß S, Vela Nava EA, Johns M. Code-reuse attacks for the Web: Breaking cross-site scripting mitigations via script gadgets. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Dallas: ACM, 2017. 1709–1723. [doi: 10.1145/3133956.3134091]

[69] Heiderich M, Schwenk J, Frosch T, Magazinius J, Yang EZ. mXSS Attacks: Attacking well-secured Web-applications by using innerhtml mutations. In: Proc. of the 20th ACM SIGSAC Conf. on Computer and Communications Security. Berlin: ACM, 2013. 777–788. [doi: 10.1145/2508859.2516723]

[70] Wassermann G, Su ZD. Sound and precise analysis of Web applications for injection vulnerabilities. In: Proc. of the 28th ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Diego: ACM, 2007. 32–41. [doi: 10.1145/1250734.1250739]

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

[72] Le DG, Li X, Gong SR, Zheng LX. Research on second-order SQL injection techniques. Journal on Communications, 2015, 36(S1): 85–93 (in Chinese with English abstract). [doi: 10.11959/j.issn.1000-436x.2015285]

[73] Huang J, Li Y, Zhang JJ, Dai R. UChecker: Automatically detecting PHP-based unrestricted file upload vulnerabilities. In: Proc. of the 49th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN). Portland: IEEE, 2019. 581–592. [doi: 10.1109/DSN.2019.00064]

[74] Lee T, Wi S, Lee S, Son S. FUSE: Finding file upload bugs via penetration testing. In: Proc. of the 2020 Network and Distributed System Security Symp. (NDSS). San Diego, 2020. [doi: 10.14722/ndss.2020.23126]

[75] Park S, Kim D, Jana S, Son S. FUGIO: Automatic exploit generation for PHP object injection vulnerabilities. In: Proc. of the 31st USENIX Security Symp. Boston: USENIX Association, 2022. 197–214. [doi: 10.1109/SP61157.2025.00136]

[76] Pang B, Zhang YH, Gao MZ, Zhang JZ, Chen LG, Zhangt M, Liang G. PFortifier: Mitigating PHP object injection through automatic patch generation. In: Proc. of the 2025 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2025. 956–971. [doi: 10.1109/SP61157.2025.00136]

[77] Sun FQ, Xu L, Su ZD. Static detection of access control vulnerabilities in Web applications. In: Proc. of the 20th USENIX Security Symp.. San Francisco: USENIX Association, 2011. 11.

附中文参考文献

[3] Yahoo! Inc. 大规模用户数据泄露事件公告. 2016. https://www.yahoo.com/news/yahoo-announces-data-security-incident-192515010.html

[5] 慢雾安全团队. Bybit 近 15 亿美元被盗事件深度分析报告. 2025. https://www.freebuf.com/articles/blockchain-articles/bybit-hack-analysis

[8] 刘博涵, 张贺, 董黎明. DevOps 中国调查研究. 软件学报, 2019, 30(10): 3206–3226. http://www.jos.org.cn/1000-9825/5796.htm [doi: 10.13328/j.cnki.jos.005796]

[17] 王熙灶, 沈天琪, 宾向荣, 卜磊. LLM 赋能的 Datalog 代码翻译技术及增量程序分析框架. 软件学报, 2025, 36(6): 2515–2534. http://www.jos.org.cn/1000-9825/7330.htm [doi: 10.13328/j.cnki.jos.007330]

[18] 沈天琪, 王熙灶, 宾向荣, 卜磊. DDoop: 基于差分式 Datalog 求解的增量指针分析框架. 软件学报, 2024, 35(6): 2608–2630. http://www.jos.org.cn/1000-9825/7100.htm [doi: 10.13328/j.cnki.jos.007100]

[72] 乐德广, 李鑫, 龚声蓉, 郑力新. 新型二阶 SQL 注入技术研究. 通信学报, 2015, 36(S1): 85–93. [doi: 10.11959/j.issn.1000-436x.2015285]

作者简介

罗天涵, 硕士, 主要研究领域为程序分析, 软件测试及分析, 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, 副研究员, CCF 专业会员, 主要研究领域为漏洞挖掘, AI 系统安全.

谭杰, 硕士, 主要研究领域为软件工程, 软件安全性, 形式化方法.