软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007635] [CSTR: 32375.14.jos.007635]

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

ToxiHeap: LLM 制导和毒性标记的 JavaScript 引擎模糊测试框架*

沙乐天, 龙章伯, 丁加宇, 黄海平, 肖 甫

(南京邮电大学 计算机学院、软件学院与网络空间安全学院, 江苏 南京 210023)

通信作者: 龙章伯, E-mail: jaubertlong@gmail.com

摘 要: 现有浏览器 JavaScript 引擎模糊测试工具在检测潜在漏洞方面仍存在局限, 尤其对不触发崩溃的堆内存错误识别能力不足. 为此, 提出 ToxiHeap, 一种将内存毒性标记 (toxic labeling) 的运行时监测和大语言模型 (large language model, LLM) 制导语义生成, 设计为并行核心轨道的 JavaScript 引擎模糊测试框架. 检测轨在目标引擎的堆分配与访问路径上插桩, 通过影子内存与细粒度毒性标记统一刻画释放后使用 (use-after-free, UAF)、double free、堆越界读写 (out-of-bounds read/write,OOB-R/W)、内存泄漏 (memory leak) 和未初始化堆内存读 (uninitialized memory read,UMR) 等多类堆错误, 并将对已释放内存等非法状态的访问转化为可消费的异常信号, 弥补传统依赖崩溃信号方法对非崩溃漏洞缺乏有效检测信号的不足; 生成轨采用“蒸馏-激励”两阶段的 LLM 变异器, 从历史 PoC 中提炼语义特征并定向构造状态依赖的测试用例, 突破仅依赖语法或轻语义变异在深路径覆盖上的局限. 两条轨道以覆盖增益和非崩溃命中信号在反馈环路中汇合, 协同驱动样本保留、路径权重更新和知识库自增, 同时引入隔离重放验证机制, 在干净进程中二次确认异常, 显著降低非确定性误报. 实验结果表明, 在 JavaScript-Core (JSC) 引擎上, ToxiHeap 在 24 h 运行下的分支覆盖率由 31.17% 提升至 33.52%, 在 V8、SpiderMonkey (SM) 和 ChakraCore (CH) 等其他主流引擎上同样取得了最高或接近最高的分支覆盖率, 有效样本占比稳定在 91% 以上. 在覆盖 UAF 等多类缺陷模式的 50 条 PoC 参考集上, 4 个引擎综合平均的整体检出率达到 89.18%.

关键词: 模糊测试; JavaScript 引擎; 内存检测; 释放后使用; 大语言模型

中图法分类号: TP311

中文引用格式: 沙乐天, 龙章伯, 丁加宇, 黄海平, 肖甫. ToxiHeap: LLM制导和毒性标记的JavaScript引擎模糊测试框架. 软件学报. http://www.jos.org.cn/1000-9825/7635.htm

英文引用格式: Sha LT, Long ZB, Ding JY, Huang HP, Xiao F. ToxiHeap: LLM-guided JavaScript Engine Fuzzing Framework with Toxic Labeling. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7635.htm

ToxiHeap: LLM-guided JavaScript Engine Fuzzing Framework with Toxic Labeling

SHA Le-Tian, LONG Zhang-Bo, DING Jia-Yu, HUANG Hai-Ping, XIAO Fu

(School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China)

Abstract: Existing fuzzer for browser JavaScript engines still have limitations in detecting potential vulnerabilities, especially in identifying non-crashing heap memory errors. This study proposes ToxiHeap, a JavaScript engine fuzzing framework that integrates runtime monitoring based on memory toxic labeling with large language model (LLM)-guided semantic generation as two parallel core components. The detection track instruments heap allocation and access paths in the target engine. Shadow memory and fine-grained toxic labeling provide a unified mechanism for representing multiple heap errors, including use-after-free (UAF), double free, heap out-of-bounds reads and writes (OOB-R/W), memory leak, and uninitialized memory read (UMR). Accesses to freed memory and other illegal states are

* 基金项目: 国家自然科学基金重大项目 (62293503); 江苏省前沿技术研发计划 (BF2024071); 江苏省研究生科研与实践创新计划 (KYCX24_1231)

收稿时间: 2025-09-02; 修改时间: 2025-11-03, 2025-12-30; 采用时间: 2026-01-19; jos 在线出版时间: 2026-05-13

converted into consumable exception signals, which compensates for the lack of effective detection signals for non-crashing vulnerabilities in approaches that rely solely on crash signals. The generation track employs a two-stage “distill-reward” LLM-based mutator that extracts semantic features from historical proofs of concept (PoCs) and generates state-dependent test cases in a targeted manner, thus overcoming the limitations of purely syntax-based or shallow-semantic mutations in exploring deep execution paths. The two tracks are integrated through a feedback loop driven by coverage gains and non-crashing detection signals. This feedback loop jointly drives sample retention, path weight updates, and knowledge base growth. An isolated replay verification mechanism is also introduced to revalidate anomalies in a clean process, which significantly reduces nondeterministic false positives. Experimental results show that on the JavaScriptCore (JSC) engine, ToxiHeap increases branch coverage from 31.17% to 33.52% in a 24-hour run, and achieves the highest or near-highest branch coverage on other mainstream engines including V8, SpiderMonkey (SM), and ChakraCore (CH). The proportion of valid samples remains above 91%. On a set of 50 PoC references covering multiple vulnerability patterns including UAF, the average detection rate across the four engines reaches 89.18%.

Key words: fuzzing; JavaScript engine; memory detection; use-after-free (UAF); large language model (LLM)

JavaScript 引擎是由主流浏览器厂商开发的核心解释器与即时编译系统, 如 Google 的 V8、Mozilla 的 SpiderMonkey、Apple 的 JavaScriptCore (JSC), 其定位是高效执行 Web 应用中的动态脚本并保障安全隔离[1–3]. 因此, JavaScript 引擎的安全性问题直接影响全球数十亿终端用户的数据隐私和系统完整性[1,4].

JavaScript 引擎漏洞根据其表现形式可分为两类: 崩溃型漏洞和非崩溃型堆漏洞. 崩溃型漏洞 (例如堆缓冲区溢出) 会立即导致进程终止, 这种确定性特征使其较易被传统漏洞挖掘工具捕获; 而非崩溃型堆漏洞 (如释放后使用, use-after-free, UAF) 则表现出更高的隐蔽性, 其不会触发程序崩溃但可能造成内存破坏或控制流劫持, 攻击者可通过精心构造的输入逐步操控内存布局实现漏洞利用[5,6]. 这类漏洞的检测面临两大挑战: 一是触发路径的语义复杂性; 二是因为其内存状态变化不产生崩溃等可见信号, 导致现有工具普遍存在检测盲区.

模糊测试 (fuzzing) 是一种自动化的漏洞挖掘技术, 通过向程序输入大量随机或畸形的数据, 迫使程序在处理这些输入时暴露出潜在的错误或漏洞[7–9]. 该技术能够以反馈引导的随机化方式高效探索大规模代码路径空间, 在缺乏精确形式化模型的场景下仍然是浏览器内核这类规模巨大且输入逻辑复杂系统的主要自动化测试手段. 同时, 由于缺乏语义指导, 覆盖提升在实践中容易较早饱和. 在 JavaScript 引擎安全领域, 现代模糊测试技术主要聚焦 3 类高危漏洞的检测: 内存管理缺陷 (如堆溢出)、类型系统混淆以及即时编译 (just in time, JIT) 编译逻辑错误[1,4,10], 这些漏洞一旦被利用, 轻则导致信息泄露, 重则引发完整的远程代码执行攻击链.

尽管模糊测试已成为当前主流的漏洞挖掘方法, 但在应对前述两大挑战时仍显力不从心. 首先, 在应对“触发路径的语义复杂性”方面, 传统基于随机变异或固定语法的模糊测试工具 (fuzzer) 难以生成逻辑连贯且状态依赖的复杂测试用例. 这导致其在探索深层代码路径时效率低下, 难以触及那些需要精巧输入序列才能暴露的漏洞[2,11]. 其次, 针对“非崩溃信号的盲区”问题, 绝大多数 fuzzer 仍依赖程序崩溃作为唯一的漏洞检测机制. 这种机制对于隐蔽的非崩溃型堆漏洞几乎完全失效, 造成了大量的漏报, 使得许多潜在的内存破坏问题被遗漏[12]. 因此, 如何突破传统模糊测试工具在“语义理解”与“漏洞感知”两个维度的瓶颈, 已成为提升 JavaScript 引擎漏洞挖掘能力的关键所在.

针对 JavaScript 引擎模糊测试面临的工程挑战, 本文提出并实现了一套以提升工程实践价值为核心的漏洞检测优化方案. 该方案立足于现有的模糊测试框架, 通过编译器插桩技术构建了一套细粒度的内存无效状态标记系统, 用以实时监控堆内存状态, 从而精准捕获 UAF 等非崩溃型内存异常, 解决了传统工具因依赖崩溃信号而导致的漏报问题. 为提升测试用例生成的语义深度, 该方案将大语言模型 (large language model, LLM) 与常规变异器进行工程化耦合, 从已知漏洞的概念验证 (proof of concept, PoC, 即用于最小化展示某一漏洞可被稳定触发的最小触发样本) 集中提取并归纳语义模式, 生成逻辑结构复杂的定向测试用例, 以解决深层代码覆盖率增长停滞的问题. 为确保漏洞报告的可靠性, 方案引入进程隔离验证机制, 在初始状态的引擎进程中复现潜在漏洞, 以此过滤非确定性异常并降低误报率. 这些技术共同构成了一套完整的工程实践体系, 为 JavaScript 引擎的深度漏洞挖掘提供了可复用的技术范式.

相较于现有方法, 本文核心贡献可归纳为以下 3 个方面.

(1) 运行时内存安全加固的信号触发式诊断: 在引擎内以内存毒性标记 (toxic labeling) 的细粒度插桩持续标注释放/失效对象, 并在读写点触发可编程诊断信号; 相较分配器加固 (FFmalloc、BUDAlloc), 可提供可回归的检测证据; 相较静态方法 (CRed、DCUAF), 降低了并发和深语义近似导致的检测偏差.

(2) 语义闭环的样本生成与检测协同: 采用蒸馏→激励的双阶段 LLM 语义变异器, 从历史 PoC 提炼语义特征, 定向生成状态依赖、序列敏感的 JS 用例; 以毒性诊断作为在线反馈, 使生成与检测协同收敛至更易触发 UAF 的结构, 突破纯语法级变异的深路径覆盖瓶颈.

(3) 可重复性与工程化落地: 通过隔离重放 (fresh-process replay) 抑制 REPRL 长生命周期带来的非确定性误报, 产出稳定的回归样本; 在 Fuzzilli/REPRL 框架下即插即用、跨多引擎部署, 并以非崩溃检出与覆盖增益为主要指标完成评测验证.

本文第 1 节介绍相关工作和研究现状. 第 2 节分析现有模糊测试方法的主要问题并给出本文研究定位. 第 3 节介绍本文主要工作. 第 4 节通过对比实验验证所提方法的有效性和高效性. 最后讨论相关问题并总结全文.

1 相关工作

1.1 JavaScript 引擎

JavaScript 引擎是现代浏览器执行客户端脚本的核心组件, 负责将 JavaScript 代码转换为高效的机器执行指令. 主流浏览器均采用独立研发的高性能引擎, 如 Google 的 V8、Apple 的 JSC 等. 这些引擎为了追求极致性能, 普遍采用了复杂的多层 JIT 架构, 例如 JSC 从低级解释器 (low level interpreter, LLInt) 到基线 JIT (BaselineJIT)、再到深度优化的 DFGJIT 和 FTL. 这种分层架构虽然提升了执行效率, 但也引入了海量的代码和复杂的逻辑状态转换, 使其成为一个充满潜在安全漏洞的复杂攻击面. 本文后续将以 JSC 和 V8 等主流引擎作为主要研究和测试对象[1–3].

如图 1 所示, JSC 执行 JavaScript 源代码时, 流程始于词法分析器 (Lexer): 它将输入脚本拆分为词法单元, 为语法解析提供基础; 接着语法解析器 (Parser) 基于词法单元构建抽象语法树 (abstract syntax tree, AST), 再由字节码编译器 (ByteCompiler) 转换为字节码 (ByteCode, 图 1 中虚线部分). 字节码是 JSC 执行流程的核心输入. 生成的字节码先进入低级解释器 (LLInt) 完成基础执行; 当函数调用、循环迭代等行为的频次触发引擎预设阈值时, 代码优化机制启动, 程序会依据执行需求, 逐级进入 BaselineJIT (编译高频代码块加速执行)、DFGJIT (聚焦低延迟优化, 适配交互响应) 乃至 FTL (面向高吞吐量, 深度优化生成极致机器码) 阶段, 实现执行效率的分层递进提升.

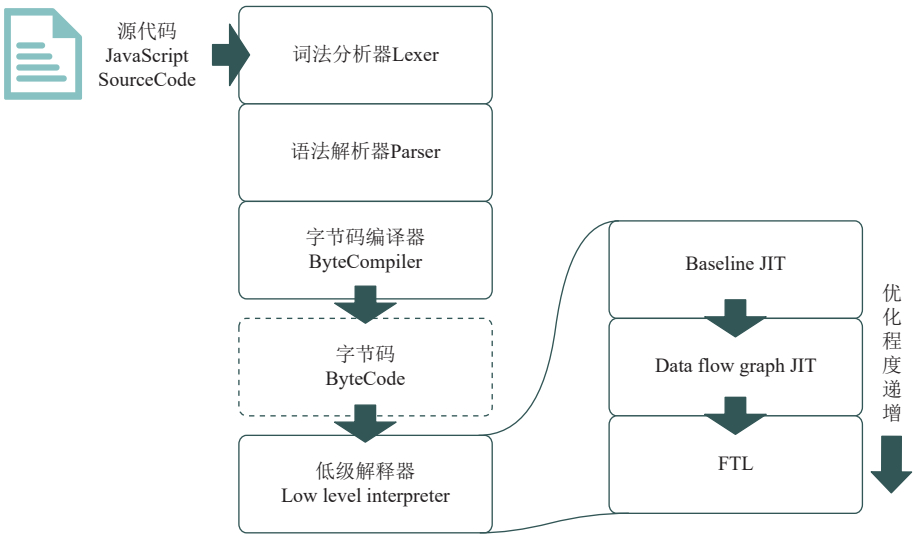


图 1 JavaScript 源代码的处理流程图

1.2 JavaScript 引擎研究现状

模糊测试是挖掘 JavaScript 引擎漏洞的主流技术之一, 其发展历程也反映了研究者们为应对引擎复杂性而进行的持续探索[1,4,10]. 特别是在针对 JIT 引擎的测试中, 由于相关漏洞的触发条件苛刻、挖掘难度高, 研究者们普遍采用差分测试 (differential testing) 方法[3]. 所谓差分测试, 是指为程序的两种不同实现 (或同一实现的不同模式) 提供相同输入, 并通过对比其输出结果是否一致来发现错误的测试技术. 在 JIT 引擎的场景下, 差分测试具体做法是将同一段 JavaScript 代码分别送入未优化的解释器和经过多层优化的 JIT 编译器执行, 并比对两者的最终输出. 该推断的理论基础是代码优化的“语义一致性”原则, 即优化过程只应改变代码的执行效率, 而不应改变其最终的计算结果. 因此, 一旦优化前后的输出不一致, 即可认为 JIT 编译器的优化逻辑中存在错误的假设或设计缺陷. 这类缺陷通常仅在特定的优化路径下被触发, 在常规解释执行中不易暴露, 具有高度的隐蔽性.

早期的模糊测试工具如 Mozilla 开发的 jsfunfuzz[13]主要通过随机变异生成测试用例, 虽然能够发现一些边界漏洞, 但由于 JavaScript 语法的严格性, 生成了大量无效用例, 效率低下. 为解决此问题, LangFuzz[14]等工具引入了语法模板, 确保了生成用例的语法正确性, 提升了有效性, 但仍缺乏对代码深层逻辑的理解. 为探索更深层次的漏洞, 研究者们开始关注测试用例的语义[15]. 例如, Park 等人[4]开发的 Die 通过保留代码的类型特征来生成语义更有效的用例; Wang 等人[16]提出的 FuzzJIT 专注于 JIT 编译器漏洞的模糊测试工具, 通过 AST 突变结合代码包装模板触发 JIT 编译优化, 并利用执行一致性验证机制; Groβ 等人提出的 Fuzzilli[1]则通过自定义的中间语言进行变异, 能够生成语法和语义双重有效的代码, 在挖掘 JIT 编译器等深层漏洞方面展现出巨大优势.

1.3 LLM 制导的模糊测试

大语言模型是一类通过在海量代码语料上进行预训练的深度学习模型[17]. 该技术在代码的语义理解与自动生成方面展现出先进能力, 其生成的代码不仅符合语法规范, 也具备逻辑连贯性. 在 LLM 出现之前, 已有研究将深度学习模型用于灰盒模糊测试并提升输入质量和路径探索能力[18]. 这一特性可被应用于模糊测试领域, 以解决传统模糊测试工具因随机变异而在处理复杂输入时所面临的覆盖率瓶颈问题[19,20]. LLM 能够基于其预训练模型中包含的语义信息, 从两个维度进行优化.

其一, 作为语义化输入生成引擎, 通过解析目标程序的应用程序编程接口 (application programming interface, API) 约束与逻辑, 生成高度符合特定场景的测试用例. 例如, TitanFuzz[21]利用该能力为 TensorFlow 生成形状兼容的张量运算序列; CHATAFL[22]则依据 RFC 文档构建符合网络协议状态的测试包. 其二, 作为结构感知变异器, 对种子文件进行保留程序语义的修改. 相关技术已从早期 Montage[23]采用的神经语言模型引导, 发展至 Fuzz4All[24]、CovRL-Fuzz[19]以及面向非文本输入格式的 LLM 合成输入生成器[25]等利用 LLM 在复杂系统中发现漏洞的应用.

近年来, 针对 JavaScript 引擎和通用软件系统的 LLM 增强模糊测试研究取得了重要进展, 但这些方法在实际应用中仍面临显著挑战. Fuzz4All[24]通过自动提示工程实现了跨语言的通用模糊测试, 但其在多个基准测试中暴露出有效输入占比偏低的问题, 且主要依赖崩溃或用户自定义的判定规则进行漏洞检测, 难以捕获非崩溃型内存错误. CovRL-Fuzz[19]采用覆盖率引导的强化学习训练 LLM 变异器, 在 JavaScript 引擎测试中取得了较好的覆盖率提升, 但需要周期性模型微调 (平均每 2.5 h 微调 10 min), 在 24 h 运行中约 73% 的时间消耗在 LLM 推理与训练上, 整体执行效率约为基线工具的 50% (约 2 倍运行时开销), 且仍依赖 ASan 等外部工具识别内存错误. Asmita 等人[12]针对嵌入式系统的 BusyBox 模糊测试工作结合了 LLM 种子生成与崩溃复用技术, 但其主要面向命令行接口的黑盒测试场景, 以进程崩溃为主要检测信号, 难以适应 JavaScript 引擎这类需要深层语义理解与非崩溃型堆漏洞检测的复杂场景. Jiang 等人[26]的综述研究系统总结了 LLM 辅助模糊测试面临的共性挑战: 驱动程序合成易出错且适用范围有限 (如 GPT-4 生成的驱动程序约有 60% 存在类型或参数初始化错误)、输入多样性不足与有效性受限以及漏洞检测判定规则理解不准确等问题, 并指出运行时监测机制的重要性.

由此可见, LLM 能够将模糊测试从句法层面的随机探索提升至语义层面的定向制导, 其目标在于通过探索深层代码路径实现更高质量的代码覆盖. 然而, 现有 LLM 增强模糊测试方法普遍存在 3 方面局限: 一是多数方法仍依赖粗粒度的崩溃信号作为判定规则, 难以有效检测非崩溃型内存漏洞; 二是在生成有效性与执行效率之间面临

权衡困境; 三是部分方法需要高昂的模型微调成本. 这些局限表明, 若要充分发挥 LLM 在 JavaScript 引擎模糊测试中的潜力, 需要在强化检测能力、优化生成策略与降低计算开销这 3 个维度同时取得突破[19,24,26].

1.4 堆内存漏洞检测方法

在模糊测试中, 用于判断程序是否触发漏洞的自动化判定标准被称为漏洞检测机制. JavaScript 引擎作为复杂的动态内存管理系统, 其堆内存操作容易引发释放后使用 (UAF)、重复释放 (double free)、堆越界读写 (out-of-bounds read/write, OOB-R/W)、内存泄漏 (memory leak) 以及未初始化堆内存读 (uninitialized memory read, UMR) 等多种安全问题[5,6,27]. 这些漏洞往往不一定立即导致程序崩溃, 而是以悄然的数据破坏或资源耗尽形式存在, 使得传统依赖崩溃信号的检测方法难以有效捕获. 本节从“能否给出可靠检测信号”的角度, 分别从静态与动态两个维度概述现有堆内存漏洞检测与防护技术在 JavaScript 引擎动态漏洞挖掘场景中的适用性与局限.

(1) 静态堆内存分析方法

静态分析路线主要面向事前风险标注, 通过对源代码或中间表示进行推断, 给出“可能存在 UAF/泄漏”等告警[28–30]. 其中, 文献 [28,29] 主要关注通用源代码漏洞检测, 文献 [30] 则面向 JavaScript 生态中的静态漏洞分析与依赖关系建模. 代表性工作 CRed[31]采用空间-时间上下文缩减方法, 在指针分析的基础上引入对象生命周期信息, 在 LLVM 的 bitcode 上静态推断 UAF 风险. 该方法通过联合建模指针别名与释放点, 在大型项目中实现了可扩展的 UAF 缺陷挖掘, 并在真实软件中发现了大量缺陷, 体现出较强的检测能力. 同时, 为控制分析成本, CRed 在循环展开、数组和链表建模、递归上下文等环节采用迭代阈值截断、数组单体化处理、在 points-to 环中跳过节点以及在强连通分量上上下文不敏感等多种近似, 论文在局限性部分指出这些设计会带来假阴性与假阳性, 且方法依赖源代码或 LLVM bitcode, 难以直接作用于仅有二进制的目标[26].

Bai 等人[32]提出的 DCUAF 面向 Linux 设备驱动中的并发 UAF[31], 通过本地接口推断与全局锁集摘要结合, 静态刻画设备驱动中的并发访问关系. 该方案在抽象内核并发原语和驱动接口协议的基础上, 在大规模驱动代码上发现了大量真实并发 UAF 缺陷, 显示出较好的可扩展性. 与此同时, DCUAF 强依赖设备驱动这一特定领域的建模, 迁移至用户态 JavaScript 引擎时需要重建并发与对象生命周期模型, 工程成本较高; 作者也指出, 函数指针分析不足、字段级别别名不精确、自并发和新线程未覆盖以及 RCU 等无锁模式未建模都是假阴性的主要来源.

针对内存泄漏, 静态分析通常基于可达性与所有权追踪, 在 JavaScript 引擎及相关生态场景下面临闭包捕获、原型链继承与弱引用 (WeakMap/WeakSet) 等语言特性带来的额外复杂度[30]. 一个看似无引用的对象, 可能仍通过事件回调或闭包间接被持有, 静态分析需要精确建模所有潜在引用路径, 其复杂度随代码规模迅速增长. 基于符号执行与约束求解的方案 (如 KLEE) 在缓冲区越界检测方面具有理论完备性, 但在面对包含 JIT 编译与动态生成代码的 JavaScript 引擎时, 符号状态爆炸与路径覆盖不足的问题尤为突出.

总体来看, 静态分析路线 (包括 CRed[31]、DCUAF[32]及其他泄漏、溢出分析工作[28–30]) 的优势在于不依赖运行时工作负载, 可在发布前对代码进行全面体检; 但在精确建模动态类型系统、原型链查找和 JIT 优化行为方面仍存在明显难度, 通常聚焦于某一特定漏洞类型, 难以形成统一覆盖多类堆漏洞的检测框架. 更重要的是, 这类方法给出的是“某个程序点存在潜在风险”的离线结论, 而非“在当前这条输入下实际触发了某次 UAF/泄漏”, 难以在模糊测试过程中充当在线判定规则, 也不便与覆盖反馈等动态信号整合.

(2) 运行时防护方法的定位差异

动态方法直接作用于程序运行过程, 通过监控实际执行轨迹来发现堆内存错误, 在模糊测试场景中更贴近工作负载. 最传统的一类是基于崩溃信号的检测. 早期 Mozilla 的 jsfunfuzz 这类纯随机变异工具[13], 以及引入语法模板以提升用例有效性的 LangFuzz[14], 均主要依赖段错误、非法指令等异常终止作为漏洞触发信号. 此后, 在生成侧逐步引入语义约束的框架, 如 Die[4]通过类型保留策略构造易触发优化错误的代码; FuzzJIT[16]针对 JIT 编译器设计 AST 级突变配合代码包装模板; Fuzzilli[1,2]采用自定义中间语言 FuzzIL 生成语法和语义双重有效的输入. 这些工具在用例构造能力上不断进化, 但在检测侧仍以进程崩溃为主, 在 JIT/解释器结果对比场景下辅以差分输出作为辅助判定依据. 对于 UAF、double free、OOB-R/W、memory leak 和 UMR 等非必然崩溃的堆错误, 只要尚未触及保护页或关键元数据, 这一路线基本无法感知.

为突破“只看崩溃”的局限, 一类工作通过编译器插桩引入更强的运行时检查. AddressSanitizer (ASan)[5]是代表性的系统级内存错误检测工具, 其核心思想是在编译阶段由编译器在内存访问前后插入检查代码, 在运行时通过影子内存 (shadow memory) 以 1:8 的比例为每 8 字节主内存维护 1 字节标签, 记录该区域是否可访问、是否处于红区或已释放状态. 借助这一机制, ASan 能够在错误发生瞬间捕获堆溢出 (通过在分配块前后插入红区)、UAF (将已释放块标记为不可访问) 与 double free 等问题, 而不必等待进程最终崩溃[5]. 同时, ASan 对内存泄漏需要依赖 LeakSanitizer, 对 UMR 需要配合 MemorySanitizer, 整体部署通常带来约 2 倍的时间开销和 3 倍左右的内存开销[33]. 在 JavaScript 引擎长期、大规模模糊测试场景下, 这种系统级全局插桩难以精细控制启停范围, 也难以与 fuzzer 的样本管理机制深度耦合, 不容易形成“检测-生成”的闭环. 此外, 也有研究围绕 UAF 缺陷设计专用的动态检测与模糊测试框架, 通过重用时间跟踪或定向测试用例生成显著提升 UAF 检出率[33–35], 但这类方案主要面向 C/C++程序, 尚未与包含 JIT 和垃圾回收机制的 JavaScript 引擎模糊测试框架深度集成.

Valgrind/Memcheck 通过二进制翻译和精细状态跟踪提供高精度的泄漏和未初始化读写检测, 但往往会带来 10–20 倍的性能开销[27], 在在线模糊测试中难以承受. 基于 GuardPage 的防护机制在分配块前后插入不可访问页, 可以有效捕获跨页边界的堆溢出, 却对同页内的小幅越界无能为力. 现代分配器如 tcmalloc、jemalloc 通过随机化、元数据隔离等手段降低堆破坏的可利用性, 重点仍在加固系统而非输出适合模糊测试使用的精确告警[36,37].

另一条动态路线以堆分配器硬化与运行时防护为核心, 通过改变地址分配策略或加固分配器行为, 从根本上降低了 UAF 等漏洞被成功利用的可能性. Wickman 等人[36]提出的 FFmalloc 采用快速前向分配 (fast forward allocation) 思想, 为每次堆分配提供一次性虚拟地址, 避免复用已释放地址, 从地址空间层面阻断主流 UAF 利用链. 该策略无需复杂静态分析, 即可显著降低 UAF 利用成功率, 对多种已有利用技术形成有效抑制. 与此同时, FFmalloc 在 SPEC CPU2006 等基准上引入约 2.08 倍内存开销, 且未全面包裹 mmap 与 munmap 调用, 存在潜在绕过路径, 在长时间在线模糊测试场景中的适用性受到限制.

Ahn 等人[37]提出的 BUDAlloc 将虚拟地址管理从内核解耦, 引入独立的别名管理器, 在用户态灵活控制地址分配与复用. 该方案提供预防模式与检测模式两种配置, 在 SPEC CPU2006 基准上实现了约 1.16 倍性能开销和 1.25 倍内存开销, 在保持较强 UAF 防护能力的同时改善了资源使用状况. 然而, BUDAlloc 依赖 eBPF 与内核协作, 通常通过 LD_PRELOAD 替换系统分配器, 工程侵入性较强, 主要聚焦 UAF 防护, 对 double free、OOB-R/W 和 UMR 等其他堆错误的覆盖有限, 难以直接为 JavaScript 引擎模糊测试提供按输入实例化的精确告警.

综合来看, 动态检测与运行时防护方法在“是否能看到真实执行路径上的错误”这一点上具有天然优势, 但各自也存在明显局限. 以崩溃为核心信号的传统模糊测试框架只能捕获少数演化为异常终止的严重错误; 系统级 Sanitizer 与模拟执行工具虽具备较强检测能力, 但插桩与运行开销较高, 部署粒度粗, 难以长期在线启用; 堆分配器硬化与预防型防护抬高了利用门槛, 却通常不输出带输入与程序点信息的结构化报告, 更像“安全加固”而非“检测判定规则”. 在实际模糊测试管线中, 要么检测信号粗糙到几乎只有“是否崩溃”, 要么检测链路的性能成本难以接受, 难以满足 JavaScript 引擎长期模糊测试对“精确、轻量、多类型堆错误检测信号“的需求. 在 Windows 二进制和闭源应用的模糊测试实践中, 研究者通过 harness 合成、目标内嵌快照以及 CreateProcess 路径优化等方式缓解类似的性能瓶颈[38,39], 这些结果表明执行环境和系统原语设计直接影响可接受的检测开销上限.

2 问题分析与研究定位

2.1 核心问题剖析

现有面向 JavaScript 引擎的模糊测试框架在用例生成侧已经能够探索相当复杂的执行路径, 但在漏洞检测侧仍高度依赖是否崩溃这一单一信号[1,4,16]. 在这种检测范式下, 只要内存错误没有立刻演化为段错误或进程终止, 就几乎不会被感知. 非崩溃型堆内存缺陷, 尤其是 UAF, 正是在这种背景下被系统性漏掉的一类代表性问题.

(1) 挑战 1: 非崩溃型堆漏洞的检测缺失

以非崩溃型 UAF 为例, 其在 JavaScript 引擎中的触发常见于脚本层可达路径与引擎内部指针状态不一致的

场景. 图 2 给出一个 JavaScript 层的抽象模型: 脚本通过结构化克隆等接口促使引擎在内部暂存对象指针 (第 1–4 行), 随后触发垃圾回收 (garbage collection, GC) 机制 (第 6 行), 使部分对象在堆上被回收; 当脚本再次访问反序列化后的对象图时, 引擎仍可能沿用旧指针完成读写, 从而访问已释放地址 (第 8 行). 由于访问往往仍落在原分配块或同一内存页内, 程序不一定崩溃, 缺陷可能在较长时间内保持可执行.

划归到 C 语言模型上, 图 3 给出对应的堆内存最小模型. 在 C 语言中, free() 之后继续使用同一指针也可能不触发段错误 (图 3 第 5 行), 只要访问偏移未触及保护页或关键元数据, 程序仍可继续执行. 因此, 以崩溃为主要信号的模糊测试工具 (Die[4]、FuzzJIT[16]、Fuzzilli[1]等) 在这类用例上很难获得有效反馈, 输入通常被当作普通样本丢弃, 从而形成系统性漏报.

```
1. // 创建复杂嵌套对象作为测试数据
2. let msg = buildDeepMap();
3. // 结构化克隆让引擎暂存对象指针
4. let data = await postMessageAndReceive(msg);
5. // 触发 GC 回收暂存对象的堆内存
6. gc();
7. // 用旧指针访问已释放内存触发 UAF 漏洞
8. data.keys().next().value.getTime();
```

图 2　JavaScript 层 UAF 抽象模型

```
1. void foo(){
2.     char* s1 =
   malloc(16);
3.     free(s1);
4.     for(int i = 0; i < 8; i++)
5.         s1[i] = 'c';
6. }
```

图 3　C 语言最小模型 UAF 示例代码

图 3 中的 C 语言代码是图 2 中 JavaScript 引擎层非崩溃型 UAF 漏洞的底层抽象模型. 二者的核心共性为均存在释放后使用的问题, 即内存区域被释放后, 程序仍通过指针访问该区域. 二者的核心区别体现在 3 个方面: 第一, 指针的存在形式不同, 图 3 中悬垂引用直接体现在显式的指针变量 s1 上, 开发者可直接操作该指针; 而图 2 中的悬垂引用不存在于脚本层的显式变量中, 而是隐藏在 JavaScript 引擎内部的结构化克隆、宿主对象缓存或 JIT 优化路径相关的数据结构里. 第二, 影响对象生命周期的方式不同, 图 3 中开发者可通过 malloc、free 等函数直接控制内存的分配与释放, 进而控制指针指向的对象生命周期; 而图 2 中脚本层无法直接操作内存, 只能通过对象别名关系、事件调度或 GC 触发时序等间接方式影响对象的生命周期, 这使得触发 UAF 的序列对系统状态和时序控制的要求更高. 第三, 漏洞检测的难度不同, 图 2 中 JavaScript 层 UAF 的上述特性, 不仅放大了通过静态建模或局部结构变异方法覆盖这类缺陷的难度, 还因缺乏崩溃信号, 进一步放大了模糊测试等依赖崩溃反馈的检测手段在形成有效反馈闭环上的难度.

目前围绕 UAF 已有大量研究, 但既有技术路线与 JavaScript 引擎模糊测试的需求仍存在错位. 静态分析方向的代表性工作 (如 CRed[31]、DCUAF[32]) 以事前风险筛查为目标, 通过空间-时间上下文缩减、本地-全局接口推断以及锁集摘要等技术在大规模代码库中挖掘 UAF 隐患, 并在 Linux 内核与设备驱动场景中发现了大量真实缺陷[31,32]. 这类方法不依赖运行时工作负载, 适合在发布前开展广覆盖体检; 但为了可扩展性往往需要引入循环截断、数据结构近似和并发模型抽象等简化假设, 从而带来假阴性与假阳性. 在 JavaScript 引擎中, 闭包捕获、原型链查找与 JIT 动态优化会使指针别名关系随执行变化, 进一步增加静态建模难度. 更重要的是, 静态分析的输出通常是潜在风险点, 而非与某条输入绑定的运行时触发证据, 因此难以在 Fuzzing 过程中充当在线判定规则, 也难以与覆盖反馈形成闭环.

运行时防护与堆分配器硬化技术 (如 FFmalloc[36]、BUDAlloc[37]以及 Guard Page、加固分配器等) 从防御角度切入 UAF, 目标是降低漏洞被成功利用的概率. FFmalloc[36]通过一次性虚拟地址分配避免地址复用, 抬高通用 UAF 利用门槛; BUDAlloc[37]将虚拟地址管理与内核解耦并引入别名管理器, 在性能与内存开销之间折中. 但从模糊测试视角看, 这一路线仍有两点不足: 第一, 检测到可疑行为时往往直接终止或扰乱程序执行, 或者仅在内部记录事件, 同时缺乏与输入、访问程序点及调用栈绑定的结构化告警, 难以提供可直接消费的反馈信号; 第二, 系统级替换分配器或依赖内核协作会引入额外性能与运维成本, 也难以在引擎内部按不同堆区域进行细粒度启停控制. 此外, 不少方案围绕 UAF 单一缺陷类型设计, 对 double free、OOB-R/W 与 UMR 等问题难以形成统一的可观

测信号.

综上, 在检测轨上面临的核心问题为: 在可接受的运行与工程开销下, 在引擎内部构建统一的堆内存状态监控机制, 使包括非崩溃 UAF 在内的多类堆错误在触发瞬间被转化为带位置与上下文信息的检测事件. 这些事件需要能够驱动样本筛选与保留, 并与 Fuzzilli 等框架的覆盖反馈形成互补闭环[1].

(2) 挑战 2: 测试用例生成的语义局限性

传统 JavaScript 引擎 fuzzer (包括 Die[4]、FuzzJIT[16]、Fuzzilli[1]等) 通过 AST 模板、中间表示和语法约束在很大程度上保证了生成代码的语法正确性, 并覆盖了部分类型系统和 JIT 优化相关路径. 然而, 这类框架的核心仍是预设的、基于局部结构变换的变异策略, 操作粒度主要停留在语句级片段的插入、删除和重排. 对于依赖对象间长期别名关系、跨函数状态演化、GC 与 JIT 多轮交替优化才能暴露的深层缺陷, 这类“结构优先”的变异很容易破坏已经形成的复杂执行上下文, 导致覆盖率在浅层路径被充分探索后快速趋于饱和.

近年来, 基于大语言模型的模糊测试研究试图通过增强代码语义理解和生成能力来缓解这一瓶颈. Fuzz4All[24]提出通用 LLM 驱动模糊测试框架, 通过自动构造提示词针对不同目标程序生成多语言、多格式输入, 在多个基准上取得了较高覆盖率. 不过实验数据显示, 在 GCC 编译器等基准上, Fuzz4All 的有效输入占比显著低于强基线工具, 例如有效样本比例约为 37.26%, 而传统变异工具 GrayC 可达到 95.96%, 单位时间可执行样本数也存在明显差距. 更关键的是, 该框架在漏洞判定上仍主要依赖崩溃或用户自定义判定规则, 对非崩溃堆错误缺乏原生支持, 这意味着即便 LLM 成功探索到深层路径, 只要缺陷以非崩溃形式出现, 仍难以被记录为有效命中.

面向 BusyBox 命令行工具集合[12]的工作将 LLM 用于初始种子生成, 并结合 CrashReuse 技术在不同版本间迁移触发输入. 这一路线证明了 LLM 在黑盒二进制测试中的种子构造能力, 但其测试对象是命令行程序, 检测信号以进程崩溃为主, 目标接口与语义与 JavaScript 引擎存在本质差异, 在堆内存状态极其复杂的引擎环境中难以直接复用. 针对 JavaScript 解释器, Eom 等人[19]提出的 CovRL-Fuzz 采用覆盖引导强化学习训练 LLM 变异器, 将覆盖反馈通过 TF-IDF 加权图集成到 PPO 训练过程中, 在 V8 上显著提升了分支覆盖并发现了 58 个真实漏洞. 然而实验表明, 在 24 h 运行中, 约 73% 的时间消耗在 LLM 推理和训练上, 整体执行效率较基线下降约一半 (约 2 倍运行时开销), 并且在多类型缺陷上的检出率分布不均衡, 对 double free、memory leak 和 UMR 的检出率分别仅为 27.8%、50.0%、42.9%, 误报率约为 22.3%. 这说明现有奖励信号更多围绕覆盖率设计, 对特定漏洞模式的定向收敛能力有限.

Jiang 等人[26]对“LLM+Fuzzing”的系统梳理进一步揭示了共性难点. 一方面, 自动合成驱动程序或 harness 的过程错误率较高, 文中报告基于 GPT-4 生成的驱动中约 60% 存在类型或参数初始化错误, 在 31 个 OSS-Fuzz 项目中仅有 14 个能够无修改通过编译并带来覆盖提升. 另一方面, LLM 在输入多样性和有效性之间难以取得平衡, 倾向于生成结构相似但语义冗余的输入, 导致有效样本比例偏低. 更为关键的是, 很多方案依赖自然语言描述或手工规则构造判定规则, 模型在理解与执行这些规则时容易出现偏差, 难以支撑对复杂内存安全问题的精确判定. Jiang 等人[26]还指出, 引入可靠运行时监测和更强执行时判定规则是提升 LLM 辅助模糊测试实用性的关键方向.

结合上述观察, 可以将生成轨上的挑战概括为 3 点: 第一, 判定规则强度不足: 无论是 Fuzz4All[24]、BusyBox[12]还是 CovRL-Fuzz[19], 大多仍以崩溃或粗粒度用户自定义规则作为主要信号, 对于非崩溃堆错误缺乏原生支持, 形成“LLM 能生成但检测不到”的矛盾. 第二, 生成有效性与执行效率难以兼顾: 一类方法在保持高吞吐量的同时牺牲有效样本占比, 另一类方法通过强化学习提高针对性, 却付出高昂推理与训练开销, 在有限测试预算下难以维持足够样本数. 第三, 生成策略与漏洞模式之间耦合不足: 现有方案大多将覆盖率作为核心优化目标, 奖励信号与具体缺陷类型之间缺乏直接联系, 在多类型非崩溃型堆漏洞上的检出率表现不均衡. 这些问题与第 2.1 节中挑战 1 对检测轨缺口的分析叠加, 构成了 JavaScript 引擎非崩溃内存漏洞模糊测试面临的主要障碍.

在这样的背景下, 本文在挑战 2 中关注的问题是: 在保留 LLM 在语义构造方面优势的前提下, 如何让生成轨与检测轨紧密耦合, 使得用例生成不仅追求覆盖率提升, 更围绕 UAF、double free、OOB-R/W、memory leak 与 UMR 等非崩溃堆错误持续收敛, 从而在检测轨提供强判定规则的基础上, 构建一条语义定向的测试用例生成路径.

表 1 的分析结果显示, 现有 JavaScript 引擎 fuzzer 在检测轨存在严重的信号缺失, 导致挑战 1 所述的非崩溃

型堆漏洞无法被有效感知. 由于检测信号无法为生成轨提供确定性反馈, 系统面临挑战 2 提出的测试用例语义局限性, 难以构造触发深层缺陷的复杂逻辑. 这种检测轨与生成轨的协同缺失, 使得模糊测试过程无法针对堆内存状态演化建立高效的反馈闭环.

表 1　不同技术路线在 UAF 场景下的表现对比分析

技术路线	代表工具	核心检测与变异机制	在图2场景下的表现	局限性分析 (根因)
语义保留变异	Die[4]	通过保留代码类型特征生成语义有效用例. 检测主要依赖进程崩溃信号.	漏报. 无法通过崩溃信号捕获非崩溃型内存访问	仅关注“崩溃”这一单一信号; 类型保留虽提升了有效性, 但检测侧仍存在盲区
语法/语义变异	Fuzzilli[1]	使用中间语言FuzzIL生成语法语义有效的输入. 依赖REPRL模型和崩溃信号判定漏洞	完全漏报. 样本被判定为“普通样本”丢弃, 不留命中记录	核心问题在于检测机制与漏洞特征脱节: 非崩溃UAF不触发崩溃, 导致反馈环路中断
差分测试/JIT增强	FuzzJIT[16]	专注于JIT漏洞, 采用解释器与JIT结果一致性验证	部分发现/漏报. 若UAF未改变可观测输出结果, 差分信号难以触发	判定规则仅适用于“改变计算结果”的逻辑错误, 对隐蔽的内存破坏感知力不足
分配器硬化	BUDAlloc[37]	将地址管理与内核解耦, 通过地址空间隔离抑制UAF利用	静默防护. 阻断了利用但缺乏可供fuzzer消费的反馈告警	以防御为目标, 不输出结构化告警 (输入、访问点等信息), 难以形成“检测-生成”闭环
LLM强化学习	CovRL-Fuzz[19]	基于覆盖率引导的强化学习训练LLM变异器, 依赖ASan识别错误	低效探索. 奖励信号与缺陷模式脱节, 难以定向收敛至特定UAF结构	奖励主要围绕代码覆盖率设计; 判定规则依赖外部工具, 且推理/微调开销极高 (50%性能损耗)

2.2 研究目标与价值

图 2 与图 3 展示了同一类非崩溃 UAF 在脚本抽象层与堆内存层的共同特征: 对象已被回收但内部仍存在可达路径, 访问行为不一定触发崩溃, 从而难以被依赖 Crash 信号的测试框架识别. 结合挑战 1 和挑战 2 可以看出, 现有方法的主要瓶颈并不只来自路径覆盖不足, 更来自缺陷状态难以被信号化、用例生成难以稳定表达跨对象依赖与时序约束这两个环节的断裂. 因此, 本文需要同时回答两个问题: 如何为非崩溃型堆错误构造可在线消费的判定信号; 如何围绕该信号组织用例生成与样本管理, 使搜索过程能够向缺陷模式收敛.

围绕上述问题, 本文的研究目标可以概括为: 面向 JavaScript 引擎长期漏洞检测, 构建一套能够在线感知非崩溃型堆错误的判定信号体系, 并在此基础上形成可持续收敛的语义化用例生成与调度策略, 使样本演化能够围绕第 1.4 节论述的缺陷模式持续推进.

为实现该目标, 本文从 3 个方面展开方法设计与工程实现. 首先, 在检测侧, 在引擎内部对堆对象分配、释放与访问路径进行细粒度插桩, 用统一的状态编码刻画对象生命周期与边界条件, 并将对无效状态的访问转化为包含输入、对象标识与访问点信息的结构化事件, 用于补足非崩溃场景下的反馈信号缺失. 其次, 在生成侧, 结合历史 PoC 与已验证触发样本提炼可复用的语义特征, 将大语言模型与常规变异器在工程上耦合, 使生成过程能够表达跨作用域的对象别名关系、序列敏感的调用依赖和 GC 等时序约束, 并以检测侧事件作为反馈驱动样本演化. 最后, 在验证侧, 引入隔离重放机制对可疑样本进行二次裁决, 用于抑制长生命周期执行模型与引擎内部缓存带来的非确定性影响, 并产出可复现的回归用例.

上述设计的价值在于: 一方面, 它为 JavaScript 引擎模糊测试提供了面向非崩溃型堆错误的在线判定基础, 使隐蔽的内存状态异常能够进入反馈闭环; 另一方面, 它为 LLM 辅助用例生成提供了与缺陷模式直接相关的反馈入口, 避免生成策略长期停留在覆盖率导向的浅层收敛. 围绕这一目标, 本文在 JSC、V8、SM 与 CH 等主流引擎上开展对比评测, 用于量化检测信号强度、生成有效性和工程开销之间的折中关系.

3 方法设计

为同时提升非崩溃型堆漏洞的可观测性和深层状态相关测试用例的生成能力, 本文围绕“检测”与“生成”两条主线设计了 ToxiHeap 框架. 从概念上看, ToxiHeap 将覆盖导向模糊测试拆分为两项相互协同的核心任务: 一方面,

通过一条专门的检测轨持续监控堆内存状态, 将原本仅以偶发崩溃表现出来的 UAF、OOB-R/W 等行为显式化为可消费的运行时信号; 另一方面, 通过一条专门的生成轨面向这些信号进行定向输入空间搜索, 利用大模型构造更容易触发深层堆缺陷的高价值测试用例.

从系统视角出发, ToxiHeap 由以下两大核心组件构成.

1) 检测轨 (ToxiHeap monitor): 在目标 JavaScript 引擎的堆分配与访问路径上插桩 (instrumentation), 结合影子内存 (shadow memory) 与毒性标记 (toxic labeling) 机制, 细粒度追踪堆对象的生命周期与边界条件. 一旦访问命中已释放堆内存, 检测轨立即生成结构化事件并上报 fuzzer, 同时触发隔离重放, 以在受控环境中验证并固化可复现证据.

2) 生成轨 (LLM-Mutator): 以既有漏洞 PoC 集合作为知识基座, 通过“语义蒸馏”阶段提取类型、对象、调用序列与依赖关系等结构化特征, 在“激励”阶段将这些特征注入种子用例的定向变异过程中, 形成面向特定缺陷模式的语义感知变异器, 从而提高命中复杂堆缺陷的概率.

两条轨道在实现上彼此解耦, 在由覆盖增益与异常事件驱动的反馈回路中紧密耦合: 检测轨侧重于提升非崩溃型堆漏洞的可观测性, 生成轨侧重于提升针对这些漏洞的触达能力与搜索效率, 二者共同通过该反馈回路影响后续样本选择与调度策略.

3.1 工具链基础适配

在展开两条核心轨道之前, 需要先确定 ToxiHeap 落地在哪一类模糊测试框架上以及如何进行工程化集成. 本节选用 Fuzzilli 作为执行骨架, 说明本文如何复用其高效的 REPRL 执行模型, 并对其异常判定机制进行改造, 使之能够承载 ToxiHeap 的检测轨与生成轨.

Fuzzilli[1,2]作为一款以 Swift 语言开发的 JavaScript 引擎专用模糊测试框架, 其核心价值在于通过深度探测编译器及优化器组件以发掘潜在安全缺陷. 相较于传统模糊测试工具, 该框架创新性地采用基于语义模板的代码生成策略 (现存的大部分是漏洞样本的 PoC), 结合多维度智能决策机制, 可产出既符合语言规范又蕴含丰富执行语义的测试用例, 从而有效触发 JIT 编译器优化路径如 DFG 与 FTL 中的深层逻辑缺陷. 同时, 尤为关键的是其交互架构设计, Fuzzilli 框架通过 REPRL (读取-求值-输出-重置循环) 机制与目标引擎构建持久化通信通道, 其技术精髓在于在初始化阶段即与引擎进程预分配共享内存区域作为双向数据交换枢纽, 测试用例生成后直接写入该缓冲区, 引擎进程实时读取执行并将执行状态回传至同一共享区域, 由此实现单次进程启动即可持续处理海量测试用例的高效工作模式. 此种架构规避了传统方案中频繁创建销毁进程的系统开销[1,2], 使测试引擎得以维持长时间稳定运行状态, 显著提升了对复杂执行路径的探索深度与漏洞捕获概率. 然而, 当前 Fuzzilli 框架的异常判定机制仍存在结构性局限: 其依赖程序崩溃信号作为漏洞判定依据的策略, 虽能有效捕获导致进程终止的显性缺陷, 却难以识别诸如释放后重用等内存安全问题. 此类漏洞在操作已释放内存区域时往往不会立即引发崩溃, 而是潜伏至内存边界越界等特定条件触发, 使得基于崩溃检测的判定机制在这类非显性缺陷面前几乎无法发挥作用. 因此, 现有异常检测范式亟须向多维度监控体系演进, 通过整合内存状态追踪与语义一致性校验等技术手段, 方能具有覆盖崩溃型与非崩溃型堆漏洞的完整检测能力.

3.2 检测轨: 内存监控和无崩溃检测

在完成底层框架适配之后, 本节展开 ToxiHeap 的第 1 条核心轨道, 检测轨 (ToxiHeap monitor). 检测轨的职责是把原本只有“崩溃/不崩溃”的粗粒度反馈, 细化为可被 fuzzer 消费的堆内存状态信号, 为后续样本生成和调度提供高置信度的非崩溃判定信号.

作为并行核心之一, 检测轨在程序执行期间以毒性标记实时追踪堆对象的有效性, 生成可被 fuzzer 消费的非崩溃告警信号, 并通过隔离重放确保可复现与高置信度. 在 Fuzzilli 框架中, 漏洞检测主要依赖于程序是否发生 Crash (如内存溢出、访问非法内存、堆栈破坏等), 这一方法在检测大部分显式内存错误时具有较高的效率[1]. 然而, 对于某些类型的漏洞, 尤其是非崩溃型堆漏洞 (如 UAF、double free、OOB-R/W、memory leak 和 UMR 等), 这种方法存在明显的检测盲点. UAF 漏洞的核心特征是在内存块被释放后, 程序仍保留其引用指针 (即悬垂指针),

当后续通过该指针访问或修改内存时可能导致数据损坏或执行流劫持; double free 漏洞表现为对已释放内存的重复释放操作, 可能破坏分配器元数据; OOB-R/W 漏洞涉及对分配块边界外内存的非法访问, 可能导致相邻对象损坏; memory leak 表现为已分配但无引用指向的内存未被释放, 长期运行会导致资源耗尽; UMR 漏洞则是对未初始化内存的读取操作, 可能泄露敏感信息. 由于这些漏洞在初期可能不会立即引发崩溃, 只有在特定操作时才会显现异常. 这种延迟性使得传统基于即时崩溃的漏洞检测方法难以有效捕获非崩溃型堆漏洞, 从而显著降低了检测的覆盖率和有效性.

为了有效检测非崩溃型堆漏洞, 本文提出一种基于内存状态实时监控的检测方法. 传统模糊测试主要依赖程序崩溃来发现漏洞, 但非崩溃型堆漏洞具有延迟触发的特性, 访问已释放内存、越界访问或未初始化内存往往不会立即导致程序崩溃, 因此容易被现有方法遗漏[5,6,27]. 为了直观说明基本思想, 可以先将内存状态抽象为“有毒/无毒”的二元划分: 每当内存块被释放时, 系统会将其视为“有毒”; 当该内存被重新分配使用时, 则恢复为“无毒”. 在实现中, 本文进一步将该二元划分细化为包含 V、F、Q、RL、RR、U、M 的多状态编码, 以统一覆盖多种非崩溃缺陷. 对于 OOB-R/W, 通过在堆块两端设置红区检测边界访问; 对于 memory leak, 通过引用计数监控无引用但未释放的对象; 对于 UMR, 通过标记未初始化区域实现检测. 在此机制下, 任何对仍处于“有毒”状态内存的访问操作、对红区的越界访问、长期无引用的内存块或未初始化区域的读取都会被识别为非法行为并触发预设的异常响应, 即使没有发生实际崩溃, 也能及时暴露潜在的多种堆漏洞, 从而显著提升检测覆盖率.

这种方法的关键优势在于, 它能够在没有触发崩溃的情况下, 监控并检测到涉及已释放内存、越界访问、未初始化内存等多种潜在问题, 特别是对一些无法通过传统崩溃检测方法识别的非崩溃型堆漏洞进行有效识别. 通过对内存状态的精确控制与监控, 本方法能够提升模糊测试对多种潜在堆漏洞的检测能力. 具体设计逻辑分为以下两个方面.

1) 为高效实现上述内存状态的跟踪, 本文设计了一种基于内存粒度的映射机制. 如图 4 所示, 由于堆内存分配通常遵循一定的对齐规则和最小单位, 系统可以将每个内存地址按固定粒度划分, 并将其映射到一个独立的监控区域中对应的标志位上. 该标志位用于记录相应内存区域的当前安全状态. 当某块内存被释放时, 系统会将对应的所有标志位设置为“有毒”; 当该内存被再次分配时, 则将其标志位清除, 恢复为“无毒”. 通过这种细粒度的状态映射方式, 系统能够以较低的开销实现对整个堆空间的全面监控.

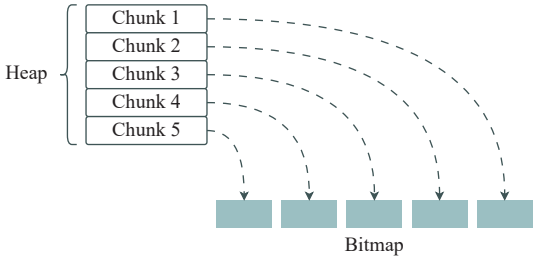


图 4　映射机制图

2) 为确保运行时每一次堆内存访问都得到有效检查, 本文采用编译器插桩技术, 在目标 JavaScript 引擎中与堆对象相关的内存访问指令前插入检查逻辑. 该检查逻辑会在访问发生前查询目标地址所映射的状态标志位. 如果发现目标内存处于“有毒”状态, 则立即中断执行并触发中毒异常; 若状态为“无毒”, 则允许访问正常进行. 这一机制保证了内存访问检查的全面性和实时性, 能够在错误发生的第一时间做出反应, 避免问题被掩盖或延迟暴露. 触发中毒异常后会启动异常处理例程, 其利用 fuzzer 和 JavaScript 引擎之间的通信机制, 将此类特殊异常信息发送到 fuzzer. 当 fuzzer 接收到 JavaScript 引擎的执行结果后, 会判断是否由访问有毒内存引起的异常. 如果确认是由有毒内存访问触发的异常, fuzzer 会在实现上复用既有的崩溃处理管线, 将其编码为一种“中毒异常”类型, 并据此调整后续的样本生成策略; 从程序语义上看, 这类事件对应的是由堆中毒标记触发的非崩溃堆缺陷告警.

Fuzzilli 采用的 REPRL 机制通过复用单个 JavaScript 引擎进程实现了高速模糊测试, 极大地提升了效率. 在测

试过程中, 一旦某个测试用例引发了崩溃或中毒异常, 系统会先尝试对其进行二次验证以过滤误报. 然而, 一个直接的验证方法, 即在当前仍在运行的、其内部状态已受先前大量测试影响的进程中重跑该用例. 但该方法存在着严重的弊端, 由于该进程的内部状态 (如内存布局、JIT 编译缓存等) 已变得高度复杂, 可能导致产生大量难以分析、难以复现的报告, 降低缺陷定位效率.

为了根除上述环境因素的干扰并确保漏洞报告的可靠性, 本文引入更为严格的验证策略, 核心步骤是在一个全新的隔离环境中启动一个独立的 JavaScript 引擎进程, 使其处于初始运行状态, 如图 5 所示. 只有当引发问题的测试用例在此初始状态下的进程中依然能稳定触发相同的异常时, 才将其确认为一个确定性的、有价值的漏洞. 这种在隔离环境中进行最终确认的方法, 是排除不确定性、确保漏洞可复现性的关键环节, 从根本上提升了模糊测试的成果质量并有效减少了误报的发生.

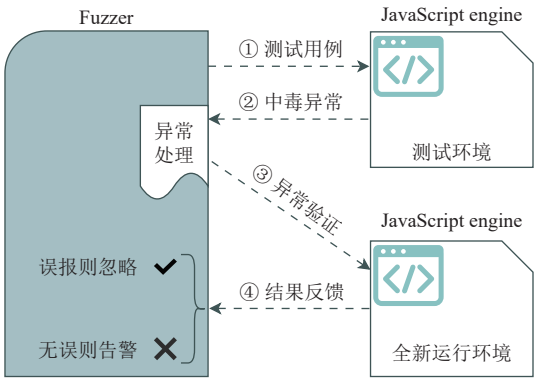


图 5 验证机制图

上文从执行流程角度介绍了检测轨如何通过影子内存和毒性标记捕获非崩溃事件. 下面进一步从抽象模型层面, 给出堆对象统一的中毒标签集合、状态机以及“标签+操作”的映射规则, 用以刻画 ToxiHeap 能覆盖哪些堆内存缺陷, 以及这些缺陷是如何在运行时被编码为统一的命中事件.

(1) 堆中毒统一层的多类型缺陷覆盖

本部分对堆对象的生命周期与边界进行一体化建模, 在统一的中毒标记层上同时覆盖多种非崩溃内存缺陷. 具体而言, 在每个堆块的用户数据区左右两侧分别插入一段不可访问的保护区, 记为左红区 *RL* (redzone-left) 和右红区 *RR* (redzone-right), 用于检测向前和向后两个方向的跨界访问; 同时在影子内存中为每个堆块维护状态集合 $S=\{V,F,Q,RL,RR,U,M\}$. 其中, *V* (valid) 是唯一的正常状态, 表示当前字节已分配且已初始化, 可以被正常读写; 其余标签均为中毒状态, 用于刻画不同的异常条件: *F* (freed) 表示已释放对象; *Q* (quarantined) 表示暂时隔离、暂不复用的堆块; *RL*/*RR* (redzone-left/right) 分别对应堆块两端的红区; *U* (uninitialized) 表示尚未初始化的数据; *M* (marked as leaked) 表示满足泄漏判定条件的对象. 运行时检查以 *V* 为唯一允许普通访问的标签, 其余任一标签一旦被访存命中或在周期扫描中被检测到, 系统将通过影子内存映射机制而非操作系统页保护触发软中断, 并与当前操作类型结合, 映射为具体的缺陷事件.

(2) 状态转移与漏洞触发规则

图 6 展示了堆块标签的状态演化. 分配新堆块时, 运行时在数据区两侧预留 *RL*/*RR* 红区, 同时将数据区内的影子标签初始化为 *U*. 某个字节第 1 次被写入后, 其标签由 *U* 变为 *V*. 程序调用释放操作时, 所有处于 *V* 状态的字节整体转为 *F*. 为延迟复用并保留重放所需信息, 系统随后把 *F* 提升为 *Q*, 将该堆块放入隔离区, 在一段时间内不参与新的分配; 当分配器再次选择复用这块隔离内存时, 对应标签由 *Q* 重新变为 *V*, 两端 *RL*/*RR* 保持不变, 用于持续进行越界检测.

内存泄漏检测结合引用计数和标签状态实现. 记 $R(x,t)$ 为堆对象 *x* 在时刻 *t* 的引用计数. 当 $R(x,t)$ 为 0, 说明不

再有活跃引用指向该对象; 若该对象在 V 状态下持续的时间间隔 Δt 大于阈值 T_{th}, 则将其标签由 V 更新为 M, 用于表示潜在泄漏风险.

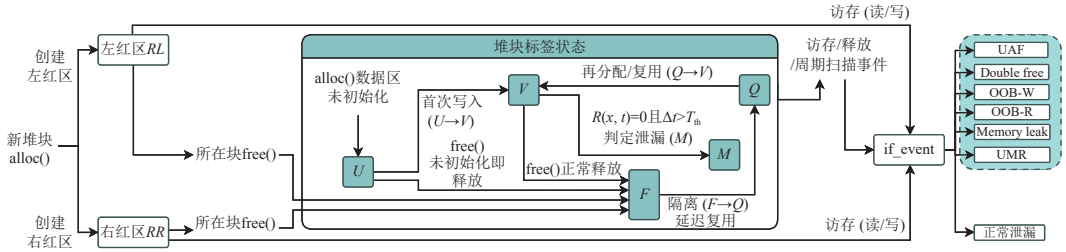


图 6　状态转移图与触发标签

(3) 标签-操作映射与缺陷分类

基于上述状态编码, ToxiHeap 按照“影子标签+运行时操作”的组合对缺陷进行分类, 表 2 概括了当前实现支持的各类缺陷及其触发条件. 访问操作命中 F/Q 时区分为两类事件: 读写访问对应 UAF; 重复释放对应 double free. 写访问命中时判定为 OOB-W, 读访问命中时判定为 OOB-R; 周期扫描命中 M 触发 memory leak 告警; 读访问命中 U 触发 UMR 告警.

表 2　ToxiHeap 实现支持的缺陷类型

缺陷类型	触发源	影子标签命中
UAF	访存	F/Q
Double free	free()	F/Q
OOB-W	访存 (写)	RL/RR
OOB-R	访存 (读)	RL/RR
Memory leak	周期扫描	M
UMR	访存 (读)	U

为支撑基于中毒标记与影子内存的检测机制, ToxiHeap 在运行时为每一次命中事件生成统一的异常记录. 核心字段包括访问地址 addr、对象标识 obj_ id、事件类型 event、程序计数器 pc、线程标识 thread_ id、访问大小 access_size、越界方向/偏移量 oob_ dir/oob_ off、访问前后对应字节的影子标签 state_ before/state_ after 以及分配、释放调用栈等信息. 该异常记录在一处集中刻画了中毒状态的演化轨迹, 为后续隔离重放、重复报告合并和证据保全提供统一输入, 从而保证每一个被报告的缺陷都可以在受控环境下稳定复现.

3.3　生成轨: LLM 驱动的变异器设计

与检测轨相对, 生成轨 (LLM-Mutator) 负责在输入空间上主动探索, 构造更有可能触发中毒事件的高价值测试用例, 是 ToxiHeap 的第 2 条核心轨道. 本节说明如何将常规变异与 LLM 制导变异组合起来, 并利用覆盖增益和命中事件构成的反馈信号, 对不同生成路径进行强化式权重更新. LLM-Mutator 通过“蒸馏阶段→激励阶段”将漏洞模式转化为定向变异能力, 并与检测轨产生的非崩溃命中信号联动, 将覆盖增益与缺陷命中联合建模为反馈信号, 从而同时实现覆盖导向和缺陷导向的闭环优化. 在第 3.2 节为模糊测试工具配备了可靠的非崩溃型堆漏洞检测与验证机制后, 核心挑战便转移至如何高效生成能够触发此类深层漏洞的测试用例. 常规变异器虽然速度快, 但在构造需要复杂逻辑与状态依赖的测试用例方面能力有限, 难以触及深层代码路径. 为此, 本文在解决检测与验证的难题之后, 进一步引入 LLM 技术, 旨在从根本上增强测试用例生成的语义深度与目标性.

本文所设计的增强型变异器是一个结合常规变异与大语言模型制导的混合框架, 其整体架构如图 7 所示. 该框架由种子库、常规变异器、大语言模型模块以及一个动态扩展的已知 PoC 集共同构成. 此设计的核心思路是发挥两类变异器的互补优势: 利用常规变异器进行快速、广泛的语法结构探索, 同时调度大语言模型完成有针对性的、开销较大的深度语义逻辑构造, 从而在测试效率与代码覆盖深度之间取得平衡. 在该框架中, PoC 集作为承

载漏洞模式知识的专家库, 不仅存储历史样本, 还能在测试过程中动态吸纳由本工具新发现并验证的有效用例, 实现知识的持续迭代与增强.

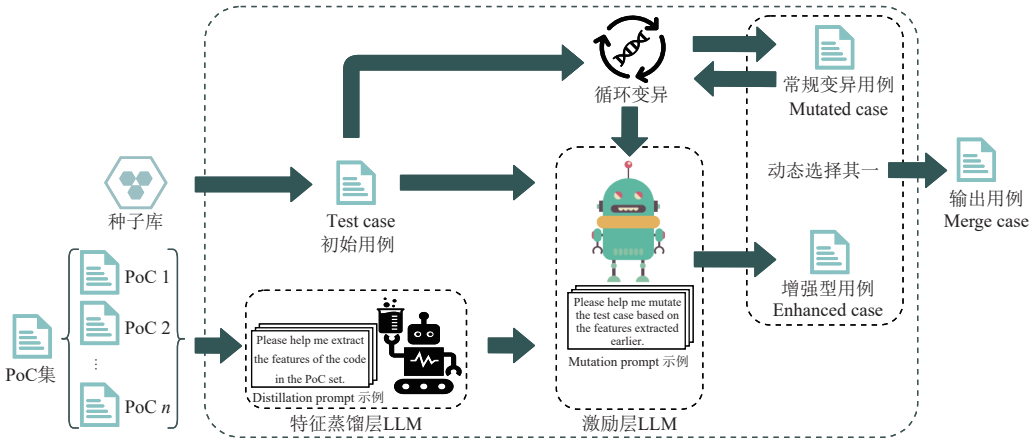


图 7 LLM-Mutator 流程图

大语言模型在本框架中的应用被设计为两个任务阶段的协作机制, 以实现从漏洞模式理解到目标测试用例生成的完整流程. 第 1 阶段是蒸馏阶段, 其提示词核心设计原则是执行语义特征提取 (semantic feature extraction). 其依据一组预设的蒸馏提示词 (distillation prompt), 对 PoC 集中的代码样本进行深度分析, 从中提炼并归纳出与漏洞触发高度相关的语法结构、函数调用序列以及数据对象间的依赖关系, 最终形成结构化的特征数据. 第 2 阶段是激励阶段, 其提示词核心设计原则是执行特征引导的定向生成. 该阶段接收一个来自种子库的初始用例 (test case), 并将在蒸馏阶段提取到的漏洞特征融入其中. 在变异提示词 (mutation prompt) 的指导下, 在激励阶段对初始用例进行改造, 使其在保持自身逻辑的同时, 也携带已知漏洞的典型模式, 从而更有可能触发相似的深层缺陷.

为协同调度常规变异器与大语言模型, 系统采用了一套结合了概率路径选择与周期性调度的混合生成策略, 具体实现如算法 1 所示. 在变异循环中, 系统通过一个计数器 (*count*) 来控制调用大语言模型的频率, 每当执行达到预设次数 (*NUM*) 后, 便会触发一次基于 LLM 的定向变异. 该算法首先从种子队列 (*QS*) 中获取一个基础用例 (*case*), 并随机选取蒸馏提示词, 调用蒸馏阶段对 PoC 集 (*SP*) 进行特征提取, 获得特征 (*features*). 随后, 系统会根据预设的路径权重, 以概率方式选择本次变异所采用的生成路径 (*path*), 该路径决定了是使用常规变异器还是激励阶段. 最后, 系统调用统一的变异函数 (*MM*), 结合选定的路径、基础用例、提取的特征以及随机选择的变异提示词, 生成最终的测试用例 (*merge_case*). 此外, 为了确保生成样本的多样性, 两个任务阶段均配备了多个候选提示词, 通过随机选择的方式为 LLM 的生成过程注入结构化的随机性, 避免了输出的单一化, 从而在保证语义正确性的前提下, 最大化地提升了测试的探索能力. 具体而言, 前者的注入结构化的随机性更多依照与 LLM 的上下文和 *temperature* (影响 LLM 输出多样性的参数) 决定的, 具有语境相关性.

算法 1. 增强型变异器的操作流程.

输入: 蒸馏提示词集合 *PD*, 变异提示词集合 *PM*, 种子队列 *QS*, PoC 集合 *SP*;
输出: 变异的路径 *path*, 变异后的测试用例 *merge_case*.

1. function *mutation_with_LLM*
2. if *count* == 0 then
3. *case* ← [*QS*] // 从种子队列随机选择一个种子用于变异
4. end if
5. *features* ← *MD*(*SP*, [*PD*]) // 随机选择提示词对 PoC 集中的代码片段进行特征提取

6. *path* ← *RandomPath*() // 按照路径权重, 随机选择一条变异路径
7. *merge_case* ← *MM*(*path*, *case*, *features*, [*PM*]) // 对种子进行变异
8. *count* ←(*count* + 1) % *NUM*
9. return *path*, *merge_case*
10. end function

3.4 整体技术架构

在分别介绍了检测轨、统一中毒层和生成轨之后, 本节将这些组件与种子队列管理和调度策略整合起来, 给出 ToxiHeap 的整体技术架构与主测试循环流程, 说明系统如何在时间维度上通过内存监控、隔离验证以及大语言模型制导变异实现各模块之间的协同工作.

该架构旨在构建一个自动化的、持续反馈的闭环模糊测试系统, 其整体设计如图 8 所示. 系统的运作始于负责维护初始测试用例集合的种子队列管理模块, 该模块不仅提供了模糊测试的起点, 更在测试过程中动态地将发现的、能够探索到新代码路径的高效用例添加回队列, 以此持续扩充和优化样本库, 提升测试的多样性. 种子队列中的样本被递送至混合式变异器模块, 该模块结合了两种策略: 一是包含随机插入、替换关键值与修改结构化内容的常规变异, 用于进行广泛的执行路径探索; 二是由大语言模型驱动的定向变异, 它基于 PoC 集中的漏洞特征进行精准构造. 生成的测试用例继而被送入用例执行模块, 在经过适配的 JSC 引擎中运行, 其执行结果最终由异常反馈模块进行分析和验证, 形成一个完整的迭代优化闭环.

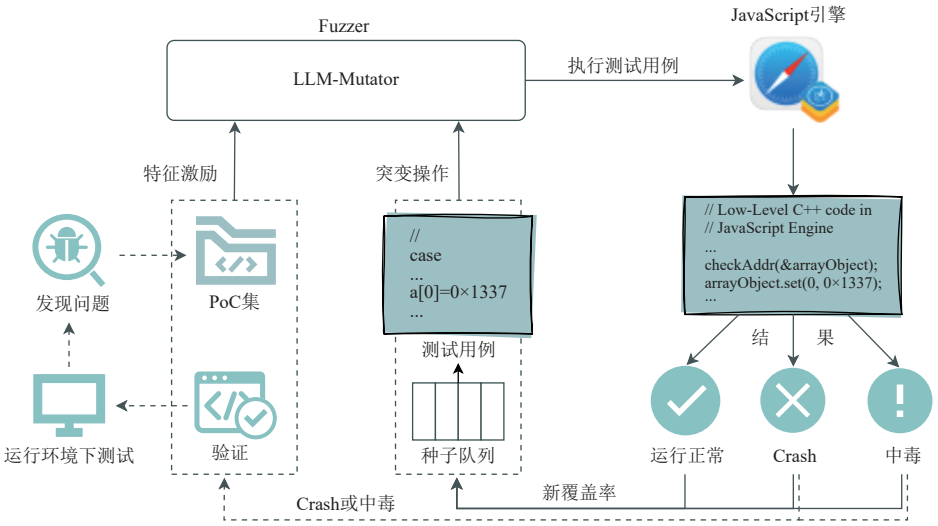


图 8 系统工作流程图

该测试架构的动态执行流程由算法 2 中定义的主测试循环驱动. 主测试循环把覆盖增益与毒性命中共同作为奖励, 对常规与 LLM 路径的权重进行自适应更新; 命中事件进入隔离重放并回填至 PoC 集与特征库. 该循环在一个预设的时间预算 (*timeBudget*) 内持续运行, 以充分利用可用时间资源. 在每次迭代中, 系统首先调用算法 1 所定义的 *mutation_with_LLM* 函数, 该函数基于当前策略生成一个变异后的测试用例 (*merge_case*) 及其对应的变异路径 (*path*). 随后, *Exec* 函数将此测试用例送入经过适配的 JSC 引擎中执行, 并同步收集其运行时信息, 其中关键的反馈指标是新产生的代码覆盖率. 系统将新旧覆盖率的差值进行归一化处理, 得到一个量化的奖励值 (*r*), 该值直接反映了本次变异的有效性. 若测试用例发现了新的代码路径, 它将被保留并加入种子队列 (*QS*), 以供后续变异使用. 同时, 系统采用一种指数移动平均 (exponential moving average, EMA) 方法来更新变异路径的权重 (ω_{path}), 其中学习率 (learning rate, *a*) 作为一个平滑因子, 用于平衡单次测试结果与路径历史表现的比重. 这种自适应的权重调整机制能够动态地强化和激励那些更高效的变异策略. 循环结束后, 程序还会对超时等异常情况进行处理, 并通

过 *RuntimeVerified* 函数在隔离环境中对潜在漏洞进行最终确认. 一旦验证为真, 该用例将被归档至 PoC 集合 (*SP*), 为大语言模型的蒸馏阶段提供新的学习素材.

算法 2. Fuzz 测试循环.

输入: 变异路径对应的权重 ω_{path}, 用于控制更新权重速度的学习率a, $a\in(0, 1)$, 种子队列 *QS*, PoC 集合 *SP*;
输出: 异常的执行结果 *result*.

```
1. function fuzz_loop
2.    while time < timeBudget do // 在正常时间预算内进行测试
3.       path, merge_case ← mutation_with_LLM(QS, SP) // 获取变异路径和测试用例
4.       result ← Exec(merge_case) // 执行测试用例
5.       r ← Normalized(new_coverage, old_coverage) // 分支覆盖率差值归一化
6.       if new_coverage > old_coverage then
7.          QS ← QS∪{merge_case}
8.       end if
9.       ω_path ← a × r + (1 – a) × ω_path_old // 更新本条变异路径对应的权重值
10.   end while
11.   if isTimeOut(result) then
12.      return timeOut
13.   end if
14.   if RuntimeVerified(result) then // 验证异常是否为触发漏洞造成的
15.      SP ← SP∪{merge_case} // 如果触发了漏洞, 将该测试用例加入 PoC 集中
16.      return result
17.   end if
18. end function
```

为确保方案的工程可用性, 本文以苹果公司的 JSC 引擎作为核心测试目标并完成针对性的平台适配. JSC 是 Safari 浏览器内核 WebKit 的核心组件, 本文解决了模糊测试工具在 MacOS 操作系统上对其进行测试时存在的兼容性与稳定性问题, 提升了工具的跨平台适用性. 在测试执行期间, 系统通过对引擎底层接口的优化, 将本文设计的内存状态监控机制与引擎自身的检查函数, 例如用于在访问内存前检查地址有效性的函数, 进行了深度协同, 从而显著提高了对内存错误的捕获精度. 当任一测试用例触发了引擎崩溃或本文定义的内存中毒异常时, 异常反馈模块会立即启动. 为确保结果的准确性, 该模块会通过启动一个全新的、独立的 JSC 进程来对该测试用例进行最终验证. 这一在隔离环境中进行最终确认的步骤, 是排除由测试环境差异导致的非确定性行为、确保所有报告的漏洞均真实可复现的关键环节, 从流程上保证了测试结果的整体质量.

4 实验分析

4.1 实验目标与维度

为系统性地验证本文所提出的增强型模糊测试工具 ToxiHeap 的综合性能, 本文的实验评估体系围绕如下 4 个方面展开.

1) 内存检测效能对比实验 (第 4.3 节): 本实验旨在验证内存检测机制对非崩溃型堆漏洞的捕获能力, 通过评估其在已知 CVE 测试用例上的检出率来实现. 为进一步评估其综合性能, 实验还将 ToxiHeap 与业界先进水平 (state-of-the-art, SOTA) 的工具, 包括 Die[4]、Fuzzilli[1]、FuzzJIT[16]、BUDAlloc[37]和 CovRL-Fuzz[19]进行基准测试, 在 JavaScriptCore (JSC)、V8、SpiderMonkey (SM) 及 ChakraCore (CH) 引擎上对比代码覆盖率与执行吞吐量.

2) 模块消融实验 (第 4.4 节): 本实验旨在量化各核心模块对整体性能的贡献. 实验通过移除特定组件, 对比不同配置下的代码覆盖率与执行吞吐量.

3) 跨平台适用性实验 (第 4.5 节): 本实验用于评估 ToxiHeap 在主流桌面操作系统上的工程稳定性与系统兼容性. 首先, 考虑到 Fuzzilli 对 MacOS 的支持有限, 而 JSC 又以 MacOS 为主要运行环境, 本文对 MacOS 版本进行了针对性适配, 并在 MacOS 与 Ubuntu 之间, 对比适配前后的样本吞吐量与分支覆盖率变化, 评估适配带来的性能增益. 其次, 以 Ubuntu 作为业界常用的模糊测试基准平台, 对适配后的 MacOS 与 Ubuntu 在 24 h 的分支覆盖率和有效样本占比等指标上的差异进行量化分析, 用于刻画 ToxiHeap 在类 Unix 环境下的稳定性与可移植性. 最后, 作为跨平台能力的补充评估, 本文在 Windows 环境上完成 ToxiHeap 的移植与配置, 并相对于 Ubuntu 基线, 对执行吞吐量、覆盖率指标进行对比, 用于刻画在异构平台下的性能折损与部署可行性.

4) 开源 LLM 适配性评估 (第 4.6 节): 为验证 ToxiHeap 对不同规模大语言模型的适应性, 本实验对比多种开源 LLM (Llama3-70b、DeepSeek-R1-70b、Qwen2.5-72b) 与闭源 GPT-4o-2024-11-20 在端到端模糊测试中的性能, 评估开源模型在 JSC 引擎上的非崩溃漏洞检测、代码覆盖率和样本有效性等指标上的表现.

4.2 环境与配置说明

为确保所有实验结果的公平性、准确性与可复现性, 本研究规定, 在针对任何特定场景进行工具性能对比时, 所有测试均在完全水平的配置下执行, 如表 3 环境参数表所示.

表 3 环境参数表

参数	平台1	平台2	平台3
系统版本	MacOS 14.4	Ubuntu 24.04	Windows 11
运行环境	MacBookPro	VMware	VMware
内存 (GB)	16	16	16
CPU	Intel Core i7	Intel Core i7	Intel Core i7

为确保实验结果的可复现性, 本研究对所有被测模型统一采用相同的推理参数: *max_tokens*=4096、*top_p*=0.7、*temperature*=0.1, 以提高生成代码的逻辑稳定性. 对比对象包括 4 个模型: Llama3-70b、DeepSeek-R1-70b、Qwen2.5-72b 以及 GPT-4o-2024-11-20 (作为基线模型). 在第 4.6 节的适配性评估中, 3 种开源模型部署在一台配备 A100 (80 GB) 的独立 GPU 服务器上, 通过本地推理服务进行调用; GPT-4o-2024-11-20 则通过官方云端 API 访问. 为便于实验控制与日志记录, 实验框架在代码层将本地推理服务与云端 API 统一封装为一致的调用接口, 以对上层组件屏蔽底层差异.

实验采用了兼具广度与深度的堆内存缺陷语料库, 由两部分构成. 第 1 部分为基于历史漏洞的参考集: 本文从 2017–2025 年间公开报告的 4 大 JavaScript 引擎及相关组件中筛选出 41 个具有代表性的 CVE, 依据官方安全公告和公开技术分析构造了 50 条 PoC 脚本, 覆盖 UAF、OBB-R/W、memory leak 等典型漏洞. 第 2 部分为定向非崩溃验证集, 共包含 87 个测试用例, 这些用例基于上述 41 个 CVE 的缺陷模式, 针对第 1.4 节论述的非崩溃堆缺陷场景进行系统化构造与扩展, 并在 4 类主流引擎 (JSC、V8、SM 及 CH) 上逐一校验可触发性. 鉴于不同引擎的堆管理机制差异, 本文将验证集划分为 4 个引擎专有的子集: JSC 子集 (21 例)、V8 子集 (25 例)、SM 子集 (23 例) 及 CH 子集 (18 例). 每个子集内的用例仅针对特定引擎的内存布局和缺陷模式进行构造与验证.

为降低数据分布偏斜对模型行为带来的偏置风险, 本文按照第 1.4 节论述的缺陷模式, 对参考集中 PoC 的主导缺陷模式进行划分, 并控制各类样本规模处于同一量级; 同时在训练与推理过程中, LLM 变异器仅接触脚本本身及运行时反馈信号, 不暴露显式的缺陷类型标签, 并在批次采样中限制单一模式的占比. 这样可以避免 LLM 模型将决策建立在某一类缺陷的频次或标签提示之上, 而是促使其从执行行为差异中学习更通用的判别特征.

4.3 内存检测效能对比实验

4.3.1 实验设置与评估指标

本实验旨在全面评估 ToxiHeap 相较于现有 SOTA 工具的综合性能. 实验设置分为两个阶段: 第 1 阶段为已知

漏洞的定向检测, 采用预设的、包含多种 CVE 类型的 PoC 测试用例集 (即第 4.2 节中定义的基于 41 个 CVE 构造的 50 条 PoC 参考集), 对 ToxiHeap、Die[4]、Fuzzilli[1]、FuzzJIT[16]、BUDAlloc[37]和 CovRL-Fuzz[19]的检出能力进行直接测试. 第 2 阶段为通用模糊测试, 各工具使用相同的初始种子语料库, 在 JSC、V8、SM 及 CH 这 4 款引擎上进行测试. 为确保结果的统计学意义并消除随机性影响, 每组实验均独立重复 10 次, 最后取平均值作分析.

实验评估指标/角度包括如下.

1) 已知漏洞检出率 (detection rate of known vulnerabilities, DRDV): 成功识别并报告预设测试用例语料库 PoC 集中漏洞的百分比.

2) 非崩溃型堆漏洞发现数 (count of non-crash vulnerabilities detected, CNCVD): 在基于 41 个 CVE 的 50 条参考 PoC 按缺陷类型人工扩展构造的 87 个非崩溃测试用例中, 统计各工具在某一 JavaScript 引擎对应测试子集中成功识别并经隔离重放或人工核实的漏洞实例数量.

3) 执行吞吐量 (execution throughput): 指 24 h 内成功执行的测试用例数量, 用于评估工具的基础运行效率. 该实验进行 24 h 运行.

4) 分支覆盖率 (branch coverage, BC): 衡量工具对目标引擎代码路径的探索深度与广度, 是评估测试全面性的关键. 为客观评估该指标的完整性和准确性, 该实验进行 24 h 运行.

5) 多类型缺陷统一检测能力验证: 在 4 类引擎上选取第 4.2 节定义的包含 87 个非崩溃型测试用例进行 24 h 测试, 统计各类型的检出率及隔离重放后的最终检出率, 验证 ToxiHeap 堆中毒统一层对多类非崩溃漏洞的检测效果.

4.3.2 实验结果与分析

对于各模糊测试工具的效能差异, 如表 4 所示, 体现在其对已知漏洞的检出能力和对未知漏洞的发现能力上.

表 4 PoC 集漏洞检出与非崩溃型堆漏洞发现对比

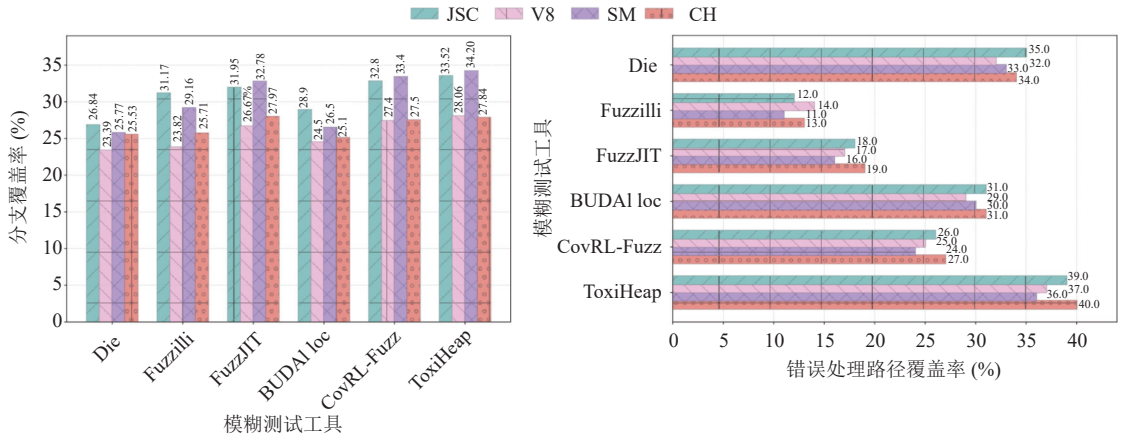
条目	JSC		V8		CH		SM	
	DRDV (%)	CNCVD (个)	DRDV (%)	CNCVD (个)	DRDV (%)	CNCVD (个)	DRDV (%)	CNCVD (个)
Die[4]	61.74	2	67.38	0	51.26	2	56.94	2
Fuzzilli[1]	62.98	3	69.25	4	49.66	2	55.92	3
FuzzJIT[16]	95.39	12	96.28	14	69.29	10	79.02	11
BUDAlloc[37]	89.23	1	91.67	0	45.12	1	52.37	7
CovRL-Fuzz[19]	94.17	8	92.84	9	72.53	12	75.46	7
ToxiHeap	95.53	21	94.23	18	79.14	19	87.83	21

注: DRDV: 针对已知预设的PoC的漏洞检出占比 (共计50个); CNCVD: 非崩溃型堆漏洞发现数 (共计87个)

如表 4 所示, 各工具在 DRDV 与 CNCVD 上的差异与其检测侧信号来源和生成侧目标设置直接相关. 在成熟引擎上, 覆盖率与触发路径通常呈现饱和趋势, 边际增量往往对应更深的 JIT 优化与异常处理逻辑, 因此需要同时关注边际效应与路径质量. Die 与 Fuzzilli 以覆盖导向变异为主, 检测端主要依赖进程崩溃, 在 JSC、V8 上的 DRDV 分别为 61.74%、67.38% 和 62.98%、69.25%, 但在 CH 上两者 DRDV 均低于 52%. 当缺陷仅表现为内存状态异常而不立刻崩溃时, 这类输入难以被标记为高价值样本并进入反馈闭环, 因此其在 4 类引擎上的 CNCVD 长期处于 0–4 个区间, 只能覆盖极少数最终演化为崩溃的样本.

FuzzJIT、BUDAlloc、CovRL-Fuzz 与 ToxiHeap 在表 4 中体现出两类增强方向: 一类通过更强的判定规则或分配器机制补足检测信号; 另一类同时耦合检测信号与生成策略, 使搜索过程围绕缺陷模式收敛. FuzzJIT 依托解释器与 JIT 的差分判定, 在 JSC 和 V8 上的 DRDV 达到 95.39% 与 96.28%, 并在各引擎上发现 10–14 个非崩溃缺陷, 但当 OOB-R/W 或 memory leak 不改变可观测结果时, 差分信号难以触发. BUDAlloc 通过分配器加固抑制地址复用并配合在线检测, JSC 和 V8 上的 DRDV 分别为 89.23% 与 91.67%, 但其检测范围更集中于 UAF 相关场景, 对 double free、OOB-R/W、memory leak 与 UMR 支持有限, 且与 fuzzer 协同较弱, 在 JSC、V8 和 CH 上的 CNCVD 均不超过 1 个, 在 SM 上为 7 个. CovRL-Fuzz 使用 LLM 提升路径探索, 在 JSC 和 V8 上的 DRDV 分别为 94.17% 与 92.84%, 但其奖励主要围绕覆盖率且判定依赖外部工具, 各引擎上的 CNCVD 在 7–12 个之间波动.

相较之下, ToxiHeap 在引擎内部构建统一堆状态监控层, 将 UAF、double free、OOB-R/W、memory leak 与 UMR 转化为可复现的运行时命中事件, 并将命中事件直接反馈给 LLM 变异过程, 使新增覆盖更集中于异常处理与内存边界相关路径. 结合表 4 和图 9 可知, 以 JSC 为例, 分支覆盖率由 31.17% 提升至 33.52%, 同时 CNCVD 由 Fuzzilli 的 3 个提升至 ToxiHeap 的 21 个, 其增量主要反映为非崩溃型堆缺陷可观测性的增强. 对比结果表明, ToxiHeap 在保持较高 DRDV 的同时, 能够稳定扩大多类型非崩溃型堆漏洞的发现规模, 缓解覆盖率提升与漏洞检出之间的脱节问题.



(a) 分支覆盖率对比 (b) 错误处理路径覆盖率

图 9 各模糊测试工具在不同 JavaScript 引擎上运行 24 h 的分支覆盖率和错误处理路径覆盖率对比

关于表 4 中的性能提升, 可从饱和探索与检出质量两个维度理解. 在饱和探索维度, 结合覆盖率评测结果 (见图 9), JSC 在 24 h 内分支覆盖率由 31.17% 提升至 33.52%, 提高 2.35%, 相对提高约 7.5%. 对于已长期测试的成熟引擎, 这一阶段的增量通常来自更深的 JIT 优化与异常处理相关路径, 例如 JIT 的 DFG、FTL 优化管线及其周边分支, 挖掘难度显著高于浅层常规逻辑. 在检出质量维度, CNCVD 在 JSC 上由 3 个提升到 21 个, 增加 18 个, 提升 600%; 在其余引擎上相对 Fuzzilli 的提升为 350%–850%, 相对 CovRL-Fuzz 在 JSC 上的提升为 162.5%. 这说明覆盖率的边际增量在该场景下主要转化为对非崩溃型堆缺陷命中事件的扩大, 并进一步产出可复现的漏洞报告, 从而体现 LLM 引导的语义变异在统一堆状态信号约束下对高风险路径的定向收敛能力.

对于样本执行吞吐量, 图 10 详细记录了 6 款 fuzzer 在 24 h 内的整体走势. 在初始阶段 (约 0–8 h), 所有工具均呈现快速增长, 此阶段主要对应于浅层代码路径探索. 其中, Fuzzilli 和 FuzzJIT 凭借其计算开销较小的变异策略表现出吞吐量优势, 分别达到约 72 万和 77 万. BUDAlloc 由于运行时防护机制的额外开销, 在第 8 h 达到约 44 万, ToxiHeap 在第 8 h 达到约 46 万. 相比之下, CovRL-Fuzz 由于强化学习推理开销, 其增速最慢, 在第 8 h 仅达到约 27 万. 然而, 表 5 的数据从质量维度量化了这一现象背后的深层原因. ToxiHeap 的有效样本占比高达 91% 以上, 在 JSC、V8 和 SM 上普遍高于 CovRL-Fuzz, 在 CH 上两者接近, 同时远高于 BUDAlloc (79.82%–87.65%). 这表明 ToxiHeap 在初始阶段较低的吞吐量是其“质量优先”策略的直接体现: 它牺牲了原始执行速度, 以确保生成的几乎每一个样本在语法上都是正确且可执行的, 从而避免了大量时间浪费在解析无效样本上. 值得注意的是, CovRL-Fuzz 虽然采用了强化学习优化, 但其有效样本占比低于 ToxiHeap, 这表明其外部检测机制限制了其在非崩溃型堆漏洞检测方面的效率.

测试进行约 8 h 后, 模糊测试进入深层语义探索阶段, 此时常规变异策略的效率降低, 导致多数工具的样本生成速率减缓. Die 的变异策略趋于饱和, 24 h 后样本数稳定在约 73 万. 在测试后期 (约 16–24 h), ToxiHeap 的 LLM 变异机制开始发挥作用. 经分析, 系统在 JSC 引擎中定位到 13 个高效的测试用例模板, LLM 随即能围绕其进行快速的语义变异, 即在保持代码逻辑正确性的前提下进行深度语义变异. 根据其自适应的变异策略, 系统会动态增加

高效变异路径的权重, 从而提高 LLM 路径的选用概率, 最终引发样本数的增长. 测试结束时, ToxiHeap 最终生成约 90 万样本, 虽然低于 Fuzzilli (约 103 万) 和 FuzzJIT (约 130 万), 但高于 Die (约 73 万)、BUDAlloc (约 68 万) 和 CovRL-Fuzz (约 49 万). 更重要的是, ToxiHeap 的有效样本占比超过 91%, 显著高于其他被测工具. 这些结果表明, ToxiHeap 依托堆中毒检测信号与 LLM 语义变异构成的协同闭环, 在牺牲少量吞吐量的前提下显著提升了样本质量与缺陷触达能力, 从而在代码覆盖率与非崩溃漏洞检出率之间实现更优的综合平衡.

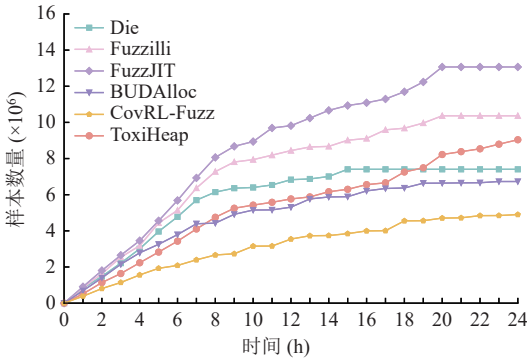


图 10　各模糊测试工具样本数随时间变化对比 (24 h 执行吞吐量)

表 5　各模糊工具样本 24 h 有效样本占比 (%)

Engine	Die	Fuzzilli	FuzzJIT	BUDAlloc	CovRL-Fuzz	ToxiHeap
JSC	61.45	71.72	70.03	85.23	91.56	92.23
V8	60.12	70.98	67.13	87.65	91.68	94.67
SM	59.33	68.09	67.87	82.45	92.87	93.44
CH	58.05	65.23	65.35	79.82	91.84	91.23

注: 有效样本为由fuzzer生成、语法合法且在目标引擎中正常完成1次执行并被纳入覆盖率或缺陷统计的测试用例

图 9(a) 与图 9(b) 分别给出了 6 种工具在 24 h 运行下的分支覆盖率与错误处理路径覆盖率. 整体结果表明, 在这两个维度上 ToxiHeap 均取得了最优表现. 两种覆盖率指标的对比揭示了不同工具的策略权衡. 例如, Die 在分支覆盖率上表现平庸 (23.39%–26.84%), 但在错误处理路径覆盖率上表现突出 (32.0%–35.0%), 这与其侧重于生成可触发已知错误模式代码的变异策略相符. 相反, Fuzzilli 在错误处理路径覆盖率上表现最低 (11.0%–14.0%), 这直接反映出其“优先生成语法有效的 JavaScript”这一设计目标, 从而系统性地规避了异常路径. 此外, 数据亦反映了引擎架构的影响, 例如所有工具在 V8 引擎上的分支覆盖率普遍偏低, 表明其庞大且快速迭代的代码库对现有模糊测试策略构成了共同挑战. 因此, 这两个维度的性能差异证实, 全面的模糊测试工具效能评估需超越单一的覆盖率指标, 并应充分考虑工具设计哲学与目标引擎架构的适配性.

在分支覆盖率方面 (图 9(a)), ToxiHeap 在 JSC、V8、SM 和 CH 上的覆盖率分别为 33.52%、28.06%、34.20% 和 27.84%, 均为对应引擎中的最高值. CovRL-Fuzz 与 FuzzJIT 的覆盖率次之, 例如 CovRL-Fuzz 在 JSC、SM 上分别达到 32.80%、33.40%, FuzzJIT 在 JSC、SM 上分别达到 31.95%、32.78%, 说明覆盖率导向强化学习与 AST 级深度变异在路径探索方面具有明显效果. Fuzzilli 在 JSC、SM 上的覆盖率为 31.17%、29.16%, 略低于上述 3 种工具, BUDAlloc 和 Die 整体处于中等水平, 在 JSC 上分别为 28.90% 与 26.84%. 此外, 所有工具在 V8 引擎上的分支覆盖率均低于 JSC 和 SM 上的数值, 表明 V8 的代码规模与复杂度对现有模糊测试机制构成了共同挑战.

错误处理路径覆盖率 (图 9(b)) 更加直接反映了各工具对异常与错误逻辑的触达能力. ToxiHeap 在 JSC、V8、SM 和 CH 上的错误处理路径覆盖率分别为 39.0%、37.0%、36.0% 和 40.0%, 显著高于其他工具, 其中在 CH 上达到全体最高值 40.0%. Die 与 BUDAlloc 构成第 2 梯队, Die 在 4 个引擎上的错误处理覆盖率为 32.0%–35.0% 区间, BUDAlloc 为 29.0%–31.0% 区间, 反映出前者针对已知错误模式的变异与后者在堆分配与地址复用路径上插桩均有助于提升异常路径触达率. CovRL-Fuzz 的错误处理路径覆盖率介于 24.0%–27.0%, 低于 ToxiHeap、Die 与

BUDAlloc, 说明其覆盖率奖励虽然能够推动整体路径加深, 但并未显式强化与堆异常和错误处理相关的路径. Fuzzilli 与 FuzzJIT 在该指标上最低, 分别约为 11.0%–14.0% 与 16.0%–19.0%, 与其优先保证正常执行语义、仅在有限场景下触及异常逻辑的设计一致.

综合 PoC 检出率、非崩溃漏洞发现数、执行吞吐量以及图 9 所示的两类覆盖率指标可以看出, ToxiHeap 在不同 JavaScript 引擎上的常规分支覆盖与错误处理路径覆盖均处于最高或接近最高水平, 且在多类型非崩溃型堆漏洞检测上显著优于现有工具. 这表明, 将堆状态监控与 LLM 驱动变异紧密耦合的设计, 在不依赖外部系统级 Sanitizer 或堆分配器替换的前提下, 能够在覆盖率与可观测性之间取得较优综合平衡, 从而在本实验设置下实现整体性能最优的模糊测试效果.

为了进一步验证 ToxiHeap 在多类非崩溃型堆漏洞上的统一检测能力, 本实验在覆盖 6 类缺陷的 87 个非崩溃型测试用例上进行了 24 h 连续测试. 如表 6 所示, 各类缺陷的检出率差异体现了影子内存状态机对不同标签的刻画强度: UAF 与 double free 的初始检出率均为 100.0%, 两类场景直接对应堆块由有效状态向 F 或 Q 状态转移, 非法访问一旦发生即可被精确标记, 可在尚未触发崩溃时暴露异常. OOB-W 与 OOB-R 读处于第 2 梯队, 初始检出率分别为 93.3% 与 100.0%, 依托堆块两端的 RL 与 RR 红区可以稳定截获大多数越界访问, 但红区粒度和插桩位置对 sub-byte 级微小越界仍构成限制. Memory leak 的初始检出率为 92.9%, UMR 为 86.7%; 前者依赖周期性可达性扫描, 对复杂循环引用和跨组件持久引用较为敏感, 后者通过 U 标签仅覆盖堆内字段访问, 未对栈与寄存器级未初始化读建模, 导致检出率相对偏低. 多类型场景下初始误报共 6 条, 约占已检出事件的 7.2% (6/83), 主要来自特殊内存复用路径和调试型保护访问. 经过在干净进程中的隔离重放验证, 最终保留 79 条可稳定复现的有效告警, 整体最终检出率达到 90.8%, 初始误报在该阶段全部被剔除, 说明在统一插桩框架下, 堆中毒统一层能够同时覆盖本文第 1.4 节所论述的等多种非崩溃缺陷, 并保持告警精度处于可接受水平.

表 6　ToxiHeap 对 6 类缺陷在 4 种 JS 引擎上的统一检测能力 (24 h)

缺陷类型	测试样本数	初始检出数	初始检出率 (%)	误报数	隔离重放后	最终检出率 (%)
UAF	18	18	100.0	0	18	100.0
Double free	12	12	100.0	1	11	91.7
OOB-W	15	14	93.3	1	13	86.7
OOB-R	13	13	100.0	1	12	92.3
Memory leak	14	13	92.9	0	13	92.9
UMR	15	13	86.7	1	12	80.0
总计	87	83	95.4	4	79	90.8

注: 测试样本数指4个引擎专用子集中该类型漏洞样本的累加总和

4.3.3　案例分析

表 4 和表 6 的结果显示, 各工具在非崩溃型堆漏洞发现数与多类型缺陷检出率上存在显著差异. 为解释这一差异的来源, 本节以 UAF 漏洞为例, 选取 WebKit 的真实漏洞 CVE-2023-28205 进行机理分析, 结合图 2 与图 3 中的模型, 说明该漏洞如何在无崩溃信号的情况下形成 UAF, 以及不同技术路线为何难以将其转化为可消费的反馈.

CVE-2023-28205 发生在结构化克隆模块 SerializedScriptValue 的反序列化路径中. 反序列化过程需要重建复杂对象图, 引擎实现会使用多个临时栈保存尚未挂接到最终返回值结构的中间对象. 图 11 展示了该类临时栈在状态机中的典型使用方式.

结合图 3 的原理, 图 11 第 7、8 行与 9–11 行对应分配、释放和使用这 3 个阶段. 关键在于图 11 第 8 行的 GC 窗口发生在第 7 行完成分配后, 但第 9–11 行仍在使用旧指针时. 而图 11 第 2–5 行的临时栈用 Vector 保存指针, 该 Buffer 不在 GC 根集合里. 第 8 行这类尚未挂接到最终对象图的引用仅通过这些栈可达, 在一次 GC 循环的标记阶段可能被遗漏并在清扫阶段被回收. 随后第 9–11 行继续从栈中取出对象并访问时就使用了已释放的地址, 即形成非崩溃 UAF 漏洞.

在该漏洞上, 各技术路线的结构性局限与挑战 1 (见第 2.1 节) 的分析一致, 其失效点可沿图 11 的 GC 窗口定

位. 覆盖导向 fuzzer 如 Fuzzilli[1]与 Die[4]主要依赖崩溃信号, 而漏洞触发依赖于图 11 第 8 行的一次 GC 循环, 该循环发生在第 7 行与第 9–11 行之间. 即使输入命中该窗口, 第 9–11 行对已释放对象的访问往往不终止进程, 缺乏可消费的判定信号, 样本难以进入反馈闭环. 差分判定路线如 FuzzJIT[16]更关注解释器与 JIT 结果的不一致, 而图 11 第 9–11 行主要体现为堆状态异常, 在多数情况下不改变可观测计算结果或差分输出, 其判定规则难以触发. 静态分析工具, 如 CRed[31]可以给出潜在风险点, 但难以在静态模型中表达出图 11 第 2–5 行临时栈不属于 GC 根集合以及第 8 行 GC 触发时机等运行时条件, 输出难以转化为与具体输入绑定的触发证据. 分配器加固路线, 如 FFmalloc[36]与 BUDAlloc[37]以防御为目标, 其主要关注点在地址复用抑制与利用链阻断, 而该漏洞的核心是图 11 第 8 行 GC 循环对临时引用扫描缺失导致的对象被回收. 这一路线通常不输出与第 9–11 行访问点绑定的结构化告警信号, 因此难以在 Fuzzing 闭环中作为反馈使用. 覆盖率奖励的 LLM 辅助方案如 CovRL-Fuzz[19]能够提升路径探索深度, 但覆盖增益无法稳定刻画第 8 行 GC 窗口是否命中. 在缺乏对第 9–11 行非崩溃内存无效状态的稳定判定信号时, 难以将该使用阶段事件持续转化为可累积奖励并实现收敛.

```
1. // CloneDeserializer::deserialize()中的临时栈
2. Vector<JSObject*, 32> outputObjectStack;
3. Vector<JSValue, 4> mapKeyStack;
4. Vector<JSMap*, 4> mapStack;
5. Vector<JSSet*, 4> setStack;
6. //  构造中间对象并暂存
7. outputObjectStack.append(outObject);
8. //  期间可能发生 GC 操作
9. putProperty(outputObjectStack.last(), propertyName, outValue);
10. JSObject* obj = outputObjectStack.last();
11. outputObjectStack.removeLast();
```

图 11　反序列化临时栈的典型使用路径 (文件节选)

相比之下, ToxiHeap 在该类漏洞上的可用性来自检测轨与生成轨的协同, 关键在于把图 11 第 9–11 行的使用阶段事件变成可用反馈信号. 检测轨通过左侧的内存粒度映射把堆地址映射到影子内存标签, 由状态机追踪释放与失效对象的毒性状态 (如图 3 与图 5 所示的映射与状态转移), 因此第 9–11 行访问旧指针时, 即使没有崩溃也能立刻触发命中事件并上报 fuzzer. 生成轨以命中事件为信号, 借助图 6 中蒸馏与激励的 LLM 变异流程构造包含结构化克隆、显式 GC 和对象图遍历的序列敏感结构, 并由图 5 的隔离重放模块在干净进程中对每一条命中事件进行复现与裁决. 命中事件与覆盖增益共同驱动图 7 所示调度策略与权重更新, 形成可持续收敛的反馈闭环. 因此, 在非崩溃 UAF 场景下, 该方法能够将内存无效状态显式化为在线反馈, 并据此驱动语义化生成策略实现定向收敛.

4.4 模块消融实验

4.4.1 实验设置与评估指标

本研究设计模块消融实验以量化各核心模块对整体性能的贡献. 实验在 JSC 引擎上连续运行 24 h, 并采用 4 种配置进行对比.

1) Fuzzilli (基准模型): 作为本研究的起点和性能基准, 代表了不包含任何新模块的原始工具.

2) ToxiHeap-NoMemTox (集成 LLM 变异): 此配置在 Fuzzilli 基础上仅添加了 LLM-Mutator 变异器, 但未使用内存毒性检测, 旨在评估 LLM 模块在提升测试用例语义复杂度方面的独立价值.

3) ToxiHeap-NoLLM (集成内存毒性检测): 此配置在 Fuzzilli 基础上仅添加了本文提出的内存毒性检测机制, 但未使用 LLM 变异器. 通过与基准模型对比, 旨在评估内存检测模块的独立贡献.

4) ToxiHeap (完整方案): 作为集成了内存检测与 LLM 制导 2 大核心模块的方案. 此配置用于展现两个模块协同工作时的综合性能.

该部分实验评估指标包括代码覆盖率和执行吞吐量, 用于衡量不同配置下的基础性能差异.

4.4.2 实验结果与分析

针对分支覆盖率效能评估, 如图 12(a) 所示, 以 Fuzzilli 的 31.17% 分支覆盖率作为基准, 仅集成内存毒性标记机制的 ToxiHeap-NoLLM 配置, 由于引入了运行时插桩的额外开销, 其覆盖率显著下降至 22.21%. 仅集成 LLM 模块的 ToxiHeap-NoMemTox 配置覆盖率为 30.95%, 与基准 Fuzzilli 基本持平甚至略低, 这证实了在缺乏精确内存状态反馈时, LLM 的语义构造能力难以转化为实际的覆盖率增益. 相比之下, 完整的 ToxiHeap 系统取得了 33.52% 的最高覆盖率. 这一结果表明, 内存毒性检测与 LLM 变异之间存在显著的协同效应: 检测轨通过识别非崩溃型异常提供高价值反馈, 生成轨据此进行定向语义变异, 从而有效突破单纯随机变异或无导向语义生成的路径探索瓶颈.

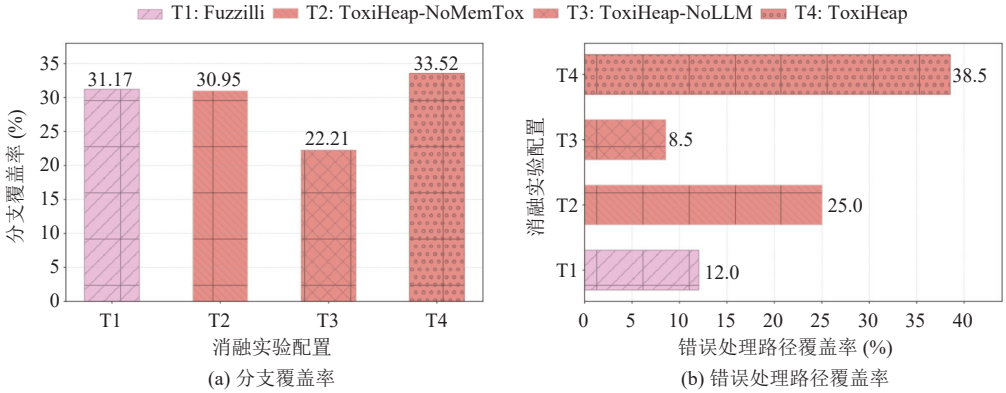


图 12 ToxiHeap 核心模块消融的分支覆盖率和错误处理路径覆盖率对比

针对各模块对执行吞吐量的影响分析, 如图 13 展示了不同配置在 24 h 内的样本生成曲线. 在测试初始阶段, 基准工具 Fuzzilli 因未引入额外检测与推理开销, 表现出最高的样本增长速率. 消融实验数据显示, 仅集成 LLM 的 ToxiHeap-NoMemTox 配置最终样本数量约为 78 万, 在所有实验配置中最低. 该结果表明, 在缺乏内存毒性状态反馈引导的情况下, LLM 变异引入了推理延迟, 且因无法捕捉非崩溃型状态以生成能够触发新路径的高质量种子, 导致生成效率随时间推移而降低. 相比之下, 完整版 ToxiHeap 尽管承载了插桩与推理的双重开销, 其 24 h 内生成的样本数仍达到约 90 万, 仅次于 Fuzzilli 并高于两个消融版本. 这证实了 MemTox 反馈与 LLM 变异之间存在协同效应: 内存毒性检测将非崩溃型异常转化为有效种子, 促进了后续变体的生成并维持了较高的有效样本生成率. ToxiHeap 在 JSC 引擎上的有效样本占比达到 92.23%, 这种高有效性带来的增益抵消了运行时插桩的执行开销. 此外, 单独引入 LLM 变异器对覆盖率提升有限, 其效能依赖于 MemTox 反馈形成的闭环机制.

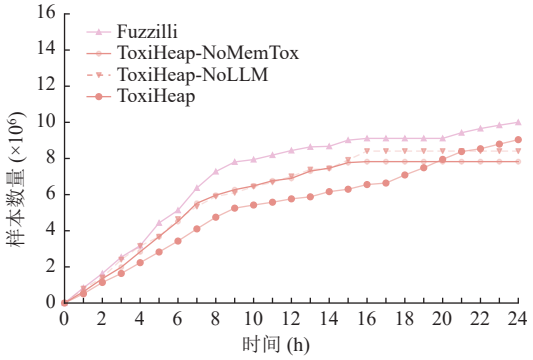


图 13 ToxiHeap 消融实验执行吞吐量对比图

4.5 跨平台适用性实验

4.5.1 实验设置与评估指标

本节的跨平台实验同样采用表 3 给出的统一配置, 在 MacOS 14.4、Ubuntu 24.04 和 Windows 11 这 3 个平台上运行 ToxiHeap. 在 MacOS 14.4 和 Ubuntu 24.04 平台上, 实验使用 24 h 测试窗口, 对比适配前后样本生成数量和分支覆盖率. 在 Windows 11 平台上, 沿用与 Ubuntu 24.04 相同的硬件和时间设置, 记录样本吞吐量与覆盖率, 并以 Ubuntu 24.04 作为基准分析不同操作系统内核机制对模糊测试效率的影响.

针对 3 个平台, 本节实验的评估指标包括如下.

1) 对 MacOS 14.4 进行适配前后 24 h 这 3 个平台的样本生成数对比.

2) 在测试结束时达到的 3 个平台最终分支覆盖率对比.

4.5.2 实验结果与分析

跨平台表现如图 14 所示, ToxiHeap 在类 Unix 环境 (Ubuntu 和适配后 MacOS) 下表现出较好的可移植性和稳定性; 对比 MacOS 14.4 与 Ubuntu 24.04 的分支覆盖率箱式图 (图 14(b)) 可以看到, 两者的覆盖分布高度接近: MacOS 的平均覆盖率 (均值) 为 35.33% (中位数 34.89%, Q1、Q3 分别为 34.59%、35.87%), Ubuntu 的平均覆盖率为 34.21% (中位数 33.93%, Q1、Q3 分别为 33.46%、35.24%), 差异约为 1.1%. 同时, 在 24 h 时间窗口内, Ubuntu 共生成 899 044 个样本, MacOS 14.4 在适配后可生成 983 768 个样本, 而适配前的 MacOS 14.4 版本仅能生成 60 721 个样本, 样本吞吐量提升约 16 倍, 说明在完成针对 JSC 的定制化适配后, ToxiHeap 能在两个类 Unix 平台上以接近的执行效率和覆盖深度运行, 框架本身不存在明显的平台偏好.

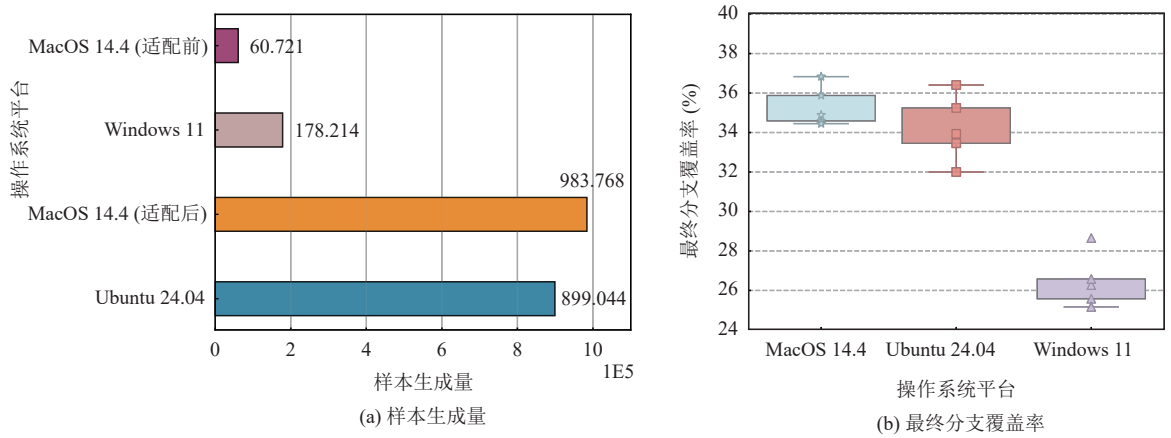


图 14 ToxiHeap 跨平台效能差异性对比 (JSC 引擎 24 h 测试)

Windows 11 的结果受限于操作系统自身进程模型对模糊测试性能的结构性影响. 在相同配置下, Windows 11 的平均覆盖率为 26.45% (中位数 26.27%), 明显低于 MacOS 14.4 和 Ubuntu 24.04; 24 h 内生成的样本数为 178 214, 仅约为 Ubuntu 24.04 的 1/5. 结合已有研究结论[38,39], Ubuntu 和 MacOS 14.4 可以借助 fork()-server 机制以低开销重复复用已初始化进程, 而 Windows 11 只能依赖 CreateProcess() 等重量级接口启动目标, 导致在同一模糊测试框架下, Windows 11 的执行吞吐量在操作系统机制上处于不利位置. 当前实验中, ToxiHeap 在 3 种平台使用相同的插桩与检测逻辑, 差异仅体现在运行时启动路径与宿主环境, 因此可以将 Windows 11 上吞吐与覆盖率的下降归因于操作系统进程创建机制而非框架设计本身. 总体上, 这组 3 平台对比表明: 在类 Unix 平台上, ToxiHeap 经过轻量适配即可达到稳定且接近的覆盖与吞吐, 而在 Windows 11 上也能在先天性能约束下保持可用的执行效率和检测能力, 验证了该框架在异构桌面环境中的工程可移植性.

4.6 开源 LLM 适配性评估

为评估 ToxiHeap 对开源大语言模型的适配性, 本研究对比了多种开源 LLM (Llama3-70b、DeepSeek-R1-70b、

Qwen2.5-72b) 与闭源 GPT-4o-2024-11-20 在端到端模糊测试中的性能. 实验在 JSC 引擎上进行 24 h 测试. 表 7 展示了主要配置的性能对比.

表 7　开源 LLM 端到端性能对比 (JSC, 24 h)

LLM模型	CNCVD (个)	检出率 (%)	分支覆盖率 (%)	样本吞吐量 (万)	有效样本占比 (%)
GPT-4o-2024-11-20	21	100	33.80	89.9	93.1
Llama3-70b	5	23.8	12.40	34.1	88.0
Qwen2.5-72b	5	23.8	12.65	33.7	89.6
DeepSeek-R1-70b	6	28.5	11.10	21.8	91.5

如实验结果表 7 所示, GPT-4o-2024-11-20 与几种开源模型之间存在明显性能鸿沟. GPT-4o-2024-11-20 的 CNCVD 为 21 个, 对应 100% 检出率, 分支覆盖率为 33.80%, 24 h 样本吞吐量为 89.9 万, 有效样本占比为 93.1%. DeepSeek-R1-70b 的 CNCVD 为 6 个, 检出率为 28.5%, 分支覆盖率为 11.10%, 样本吞吐量为 21.8 万, 有效样本占比为 91.5%. Llama3-70b 与 Qwen2.5-72b 的 CNCVD 均为 5 个, 检出率为 23.8%, 分支覆盖率分别为 12.40% 和 12.65%, 样本吞吐量分别为 34.1 万和 33.7 万, 有效样本占比分别为 88.0% 和 89.6%. 这些结果表明, 在相同时长情况下, GPT-4o-2024-11-20 在非崩溃漏洞检出能力、路径覆盖深度以及整体执行效率这 3 个维度上均显著领先; 开源模型整体处于覆盖率约 11%–13%、检出率不足 30%、样本吞吐量仅为基线 1/4 到 1/3 的水平, 仅能给出同量级的有效样本占比, 而无法在关键效能指标上缩小差距.

同时, 实验结果也揭示了开源模型的结构性特征. 几种开源模型在有效样本占比上已经接近 GPT-4o-2024-11-20: DeepSeek-R1-70b 为 91.5%, Llama3-70b 与 Qwen2.5-72b 也均高于 88%. 这一现象说明, 开源模型在遵守语法约束、生成可解析且可执行的 JavaScript 代码方面已相当成熟, 生成的输入大多能够顺利通过引擎前端和基本执行流程. 但在深层语义和状态相关缺陷上仍存在显著短板: DeepSeek-R1-70b 在检出数和有效样本占比上略优于 Llama3-70b 与 Qwen2.5-72b, 却在覆盖率和吞吐量上明显偏低, 说明其面向缺陷模式的语义构造能力尚不足以弥补推理开销带来的效率损失; Llama3-70b 与 Qwen2.5-72b 在吞吐量和覆盖率上略有优势, 却无法将更多的执行路径转化为非崩溃漏洞命中. 综合各项指标可以看出, 当前开源模型在“能生成大量语法合法用例”这一层面已经达标, 但在建模闭包、原型链、多轮 JIT/GC 交互以及异步回调等复杂语义关系时仍显不足, 难以稳定构造出能够深入引擎内部内存状态并触发特定堆错误的高价值变异代码, 导致大量有效样本停留在浅层路径探索, 对整体漏洞检出率的贡献有限.

5 总　结

本文围绕浏览器 JavaScript 引擎中非崩溃型堆漏洞难以被现有模糊测试工具有效暴露的问题, 构建了 ToxiHeap 框架, 将堆中毒统一层的内存安全检测与大语言模型制导的语义变异紧密耦合. 检测侧在引擎堆分配与访问路径上插桩, 通过影子内存与毒性标记统一刻画 UAF、double free、OOB-R/W、memory leak 及 UMR 等多类堆错误, 并将命中事件转化为可直接消费的模糊测试信号; 生成侧以历史 PoC 集为语义先验, 利用大语言模型在保持语法合法与语义相关的前提下对种子脚本进行定向变异, 从而更高效地探索与内存状态强相关的深层执行路径. 配合隔离环境重放与最小化机制, ToxiHeap 在显式提高非崩溃漏洞可观测性的同时, 将误报控制在可接受范围内. 多项实验结果表明, 该框架在多种 JavaScript 引擎和不同平台上, 均能在保持覆盖率和吞吐量可用的前提下, 显著提升已知漏洞检出率与新缺陷挖掘能力.

尽管如此, ToxiHeap 仍存在若干有待完善之处. 首先, 堆中毒统一层与大语言模型推理引入了额外的时间与资源开销, 后续工作将从插桩粒度优化、状态压缩与增量检测等方面进一步降低开销, 提升工程可部署性. 其次, 目前实现主要依赖单机多进程/多引擎调度, 尚未充分利用大规模分布式环境, 未来可结合集群调度与分布式队列, 将检测轨与生成轨解耦部署, 以进一步提升长时间模糊测试的整体效率. 最后, 尽管大语言模型在 PoC 驱动的变异任务中已展现出优势, 但在极度复杂的跨模块缺陷与罕见语义模式下仍存在覆盖不足的问题, 后续将考虑引入更细粒度的反馈信号与结构化约束, 探索面向特定缺陷模式的轻量化专用模型与混合变异策略, 以进一步增强框架在复杂漏洞场景下的适应性与稳定性.

References

[1] Groß S, Koch S, Bernhard L, Holz T, Johns M. Fuzzilli: Fuzzing for JavaScript JIT compiler vulnerabilities. In: Proc. of the 2023 Network and Distributed System Security Symp. San Diego: Internet Society, 2023. [doi: 10.14722/ndss.2023.24290]

[2] Groß S. Fuzzil: Coverage guided fuzzing for JavaScript engines [MS. Thesis]. Karlsruhe: Karlsruhe Institute of Technology, 2018.

[3] Bernhard L, Scharnowski T, Schloegel M, Blazytko T, Holz T. JIT-Picking: Differential fuzzing of JavaScript engines. In: Proc. of the 2022 ACM SIGSAC Conf. on Computer and Communications Security. Los Angeles: ACM, 2022. 351–364. [doi: 10.1145/3548606.3560624]

[4] Park S, Xu W, Yun I, Jang D, Kim T. Fuzzing JavaScript engines with aspect-preserving mutation. In: Proc. of the 2020 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2020. 1629–1642. [doi: 10.1109/SP40000.2020.00067]

[5] Serebryany K, Bruening D, Potapenko A, Vyukov D. AddressSanitizer: A fast address sanity checker. In: Proc. of the 12nd USENIX Annual Technical Conf. Berkeley: USENIX Association, 2012. 309–318.

[6] Xu GQ, Lei WQ, Gong LX, Liu J, Bai HP, Chen K, Wang R, Wang W, Liang KT, Wang WZ, Meng WZ, Liu SY. UAF-GUARD: Defending the use-after-free exploits via fine-grained memory permission management. Computers & Security, 2023, 125: 103048. [doi: 10.1016/j.cose.2022.103048]

[7] Zhu XG, Wen S, Camtepe S, Xiang Y. Fuzzing: A survey for roadmap. ACM Computing Surveys (CSUR), 2022, 54(11s): 230. [doi: 10.1145/3512345]

[8] Eceiza M, Flores JL, Iturbe M. Fuzzing the Internet of Things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems. IEEE Internet of Things Journal, 2021, 8(13): 10390–10411. [doi: 10.1109/JIOT.2021.3056179]

[9] Chafjiri SB, Legg P, Hong J, Tsompanas MA. Vulnerability detection through machine learning-based fuzzing: A systematic review. Computers & Security, 2024, 143: 103903. [doi: 10.1016/j.cose.2024.103903]

[10] Huo W, Dai G, Shi J, Gong XR, Jia XQ, Song ZY, Liu BX, Zou W. Browser fuzzing technique based on pattern-generation. Ruan Jian Xue Bao/Journal of Software, 2018, 29(5): 1275–1287 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5501.htm [doi: 10.13328/j.cnki.jos.005501]

[11] Ispoglou KK, Austin D, Mohan V, Payer M. FuzzGen: Automatic fuzzer generation. In: Proc. of the 29th USENIX Conf. on Security Symp. Berkeley: USENIX Association, 2020. 128.

[12] Asmita, Oliinyk Y, Scott M, Tsang R, Fang CZ, Homayoun H. Fuzzing BusyBox: Leveraging LLM and crash reuse for embedded bug unearthing. In: Proc. of the 33rd USENIX Conf. on Security Symp. Philadelphia: USENIX Association, 2024. 50.

[13] Snyder W. Building and breaking the browser at Blackhat. 2007. https://blog.mozilla.org/security/2007/06/04/building-and-breaking-the-browser-at-blackhat/

[14] Holler C, Herzig K, Zeller A. Fuzzing with code fragments. In: Proc. of the 21st USENIX Security Symposium. Berkeley: USENIX Association, 2012. 445–458.

[15] Ou XF, Jiang YY, Xu C. Semantic aware greybox compiler fuzz testing. Ruan Jian Xue Bao/Journal of Software, 2025, 36(7): 2947–2963 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7333.htm [doi: 10.13328/j.cnki.jos.07333]

[16] Wang JJ, Zhang ZY, Liu S, Du XN, Chen JJ. FuzzJIT: Oracle-enhanced fuzzing for JavaScript engine JIT compiler. In: Proc. of the 32nd USENIX Conf. on Security Symp. Anaheim: USENIX Association, 2023. 105.

[17] Xia CS, Wei YX, Zhang LM. Automated program repair in the era of large pre-trained language models. In: Proc. of the 2023 IEEE/ACM 45th Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 1482–1494. [doi: 10.1109/ICSE48619.2023.00129]

[18] Gao FJ, Wang Y, Situ LY, Wang LZ. Deep learning-based hybrid fuzz testing. Ruan Jian Xue Bao/Journal of Software, 2021, 32(4): 988–1005 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6225.htm [doi: 10.13328/j.cnki.jos.006225]

[19] Eom J, Jeong S, Kwon T. Fuzzing JavaScript interpreters with coverage-guided reinforcement learning for LLM-based mutation. In: Proc. of the 33rd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Vienna: ACM, 2024. 1656–1668. [doi: 10.1145/3650212.3680389]

[20] Yu JH, Lin XW, Yu Z, Xing XY. LLM-Fuzzer: Scaling assessment of large language model jailbreaks. In: Proc. of the 33rd USENIX Conf. on Security Symp. Philadelphia: USENIX Association, 2024. 261.

[21] Deng YL, Xia CS, Peng HR, Yang CY, Zhang LM. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 423–435. [doi: 10.1145/3597926.3598067]

[22] Meng RJ, Mirchev M, Böhme M, Roychoudhury A. Large language model guided protocol fuzzing. In: Proc. of the 31st Annual Network and Distributed System Security Symp. San Diego: Internet Society, 2024. [doi: 10.14722/ndss.2024.24556]

[23] Lee S, Han HS, Cha SK, Oh S. Montage: A neural network language Model-guided JavaScript engine fuzzer. In: Proc. of the 29th USENIX Conf. on Security Symp. Berkeley: USENIX Association, 2020. 147.

[24] Xia CS, Paltenghi M, Tian JL, Pradel M, Zhang LM. Fuzz4All: Universal fuzzing with large language models. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: IEEE, 2024. 126. [doi: 10.1145/3597503.3639121]

[25] Zhang KP, Li ZJ, Wu DY, Wang S, Xia X. Low-cost and comprehensive non-textual input fuzzing with LLM-synthesized input

generators. In: Proc. of the 34th USENIX Security Symp. Seattle: USENIX Association, 2025.
[26] Jiang Y, Liang J, Ma FC, Chen YL, Zhou CJ, Shen YH, Wu ZY, Fu JZ, Wang MZ, Li SS, Zhang Q. When fuzzing meets LLMs: Challenges and opportunities. In: Companion Proc. of the 32nd ACM Int'l Conf. on the Foundations of Software Engineering. Porto: ACM, 2024. 492–496. [doi: 10.1145/3663529.3663784]
[27] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proc. of the 28th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). New York: ACM Press, 2007. 89–100. [doi: 10.1145/1250734.1250746]
[28] Yang HY, Yang HY, Zhang L, Cheng X. Feature dependence graph based source code loophole detection method. Journal on Communications, 2023, 44(1): 103–117 (in Chinese with English abstract). [doi: 10.11959/j.issn.1000-436x.2023018]
[29] Zhang XJ, Zhang FH, Gai JY, Du XG, Zhou WJ, Cai TL, Zhao B. mVulSniffer: A multi-type source code vulnerability sniffer method. Journal on Communications, 2023, 44(9): 149–160 (in Chinese with English abstract). [doi: 10.11959/j.issn.1000-436x.2023184]
[30] Ferreira M, Monteiro M, Brito T, Coimbra ME, Santos N, Jia LM, Santos JF. Efficient static vulnerability analysis for JavaScript with multiversion dependency graphs. Proc. of the ACM on Programming Languages, 2024, 8(PLDI): 417–441. [doi: 10.1145/3656394]
[31] Yan H, Sui YL, Chen SP, Xue JL. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM, 2018. 327–337. [doi: 10.1145/3180155.3180178]
[32] Bai JJ, Lawall J, Chen QL, Hu SM. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In: Proc. of the 2019 Conf. on USENIX Annual Technical Conf. Renton: USENIX Association, 2019. 255–268.
[33] Zhao XQ, Qu HP, Yi JH, Wang JL, Tian MQ, Zhao F. A fuzzer for detecting use-after-free vulnerabilities. Mathematics, 2024, 12(21): 3431. [doi: 10.3390/math12213431]
[34] Du YB, Guo YA, Zhang YT, Yang J. RTT-UAF: Reuse time tracking for use-after-free detection. In: Proc. of the 38th ACM Int'l Conf. on Supercomputing. Kyoto: ACM, 2024. 376–387. [doi: 10.1145/3650200.3656606]
[35] Ba JS, Duck GJ, Roychoudhury A. Efficient greybox fuzzing to detect memory errors. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: IEEE, 2023. 37. [doi: 10.1145/3551349.3561161]
[36] Wickman B, Hu H, Yun I, Jang D, Lim J, Kashyap S, Kim T. Preventing use-after-free attacks with fast forward allocation. In: Proc. of the 30th USENIX Security Symp. Berkeley: USENIX Association, 2021. 2453–2470.
[37] Ahn J, Lee J, Lee K, Gwak W, Hwang M, Kwon Y. BUDAlloc: Defeating use-after-free bugs by decoupling virtual address management from kernel. In: Proc. of the 33rd USENIX Conf. on Security Symp. Philadelphia: USENIX Association, 2024. 11.
[38] Jung J, Tong S, Hu H, Kim T, Jin Y, Kim T. WINNIE: Fuzzing Windows applications with harness synthesis and fast cloning. In: Proc. of the 28th Annual Network and Distributed System Security Symp. San Diego: Internet Society, 2021. [doi: 10.14722/ndss.2021.24334]
[39] Stone L, Ranjan R, Nagy S, Hicks M, Govindavajhala S, Shacham H. No Linux, No problem: Fast and correct Windows binary fuzzing via target-embedded snapshotting. In: Proc. of the 32nd USENIX Conf. on Security Symp. Anaheim: USENIX Association, 2023. 4913–4929.

附中文参考文献

[10] 霍玮, 戴戈, 史记, 龚晓锐, 贾晓启, 宋振宇, 刘宝旭, 邹维. 基于模式生成的浏览器模糊测试技术. 软件学报, 2018, 29(5): 1275–1287. http://www.jos.org.cn/1000-9825/5501.htm [doi: 10.13328/j.cnki.jos.005501]
[15] 欧先飞, 蒋炎岩, 许畅. 语义可感知的灰盒编译器模糊测试. 软件学报, 2025, 36(7): 2947–2963. http://www.jos.org.cn/1000-9825/7333.htm [doi: 10.13328/j.cnki.jos.07333]
[18] 高凤娟, 王豫, 司徒凌云, 王林章. 基于深度学习的混合模糊测试方法. 软件学报, 2021, 32(4): 988–1005. http://www.jos.org.cn/1000-9825/6225.htm [doi: 10.13328/j.cnki.jos.006225]
[28] 杨宏宇, 杨海云, 张良, 成翔. 基于特征依赖图的源代码漏洞检测方法. 通信学报, 2023, 44(1): 103–117. [doi: 10.11959/j.issn.1000-436x.2023018]
[29] 张学军, 张奉鹤, 盖继扬, 杜晓刚, 周文杰, 蔡特立, 赵博. mVulSniffer: 一种多类型源代码漏洞检测方法. 通信学报, 2023, 44(9): 149–160. [doi: 10.11959/j.issn.1000-436x.2023184]

作者简介

沙乐天, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件安全, 网络安全, 物联网安全.
龙章伯, 博士生, 主要研究领域为软件安全, 网络安全.
丁加宇, 硕士, 主要研究领域为软件安全, 网络安全.
黄海平, 博士, 教授, 主要研究领域为物联网安全, 网络安全.
肖甫, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为传感网, 物联网.