软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(10):4590−4611 [doi: 10.13328/j.cnki.jos.007309] [CSTR: 32375.14.jos.007309]

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

基于抽象语法树变异的漏洞样本生成方法*

郑 炜[1], 李云帆[1], 桂 奎[1], 吴潇雪[2], 陈 翔[3], 邓沛然[1]

[1](西北工业大学 软件学院, 陕西 西安 710072)

[2](扬州大学 信息工程学院, 江苏 扬州 225127)

[3](南通大学 人工智能与计算机学院, 江苏 南通 226019)

通信作者: 吴潇雪, E-mail: xiaoxuewu@yzu.edu.cn

摘 要: 随着信息技术的持续发展, 软件产品的数量和种类不断增加, 然而即使是高质量的软件也可能存在漏洞. 此外, 软件更新速度快, 软件架构愈发复杂, 这导致漏洞逐渐进化成新的形态, 传统的漏洞检测方法和规则难以适用于新的漏洞特征. 由于零日漏洞样本的稀缺性, 软件演化过程中出现的零日漏洞难以被发现, 这为软件安全带来很大的潜在风险. 提出一种基于抽象语法树变异的漏洞样本生成方法, 能够模拟真实漏洞的结构和语法规则, 生成更符合实际情况的漏洞样本, 它可以为软件安全性和可靠性提供更加有效的解决方案. 该方法通过分析 Eclipse CDT 生成的抽象语法树结构, 提取节点中的语法信息, 重构节点和抽象语法树, 优化抽象语法树结构, 并设计一系列变异算子, 然后在优化后的抽象语法树上进行变异操作. 该方法可以生成具有 UAF 和 CUAF 漏洞特征的变异样本, 这些样本可以用于零日漏洞的检测, 有助于提高零日漏洞的检测率. 实验结果表明, 该方法比传统检测方法中的随机变异方法平均减少了 34% 的无效样本量, 并且可以生成更加复杂的变异样本; 此外, 该方法可以生成更加复杂的变异样本, 提高检测的覆盖率和准确率.

关键词: 抽象语法树; 零日漏洞; 变异算子; 漏洞样本生成

中图法分类号: TP311

中文引用格式: 郑炜, 李云帆, 桂奎, 吴潇雪, 陈翔, 邓沛然. 基于抽象语法树变异的漏洞样本生成方法. 软件学报, 2025, 36(10): 4590–4611. http://www.jos.org.cn/1000-9825/7309.htm

英文引用格式: Zheng W, Li YF, Gui K, Wu XX, Chen X, Deng PR. Vulnerability Sample Generation Method Based on Abstract Syntax Tree Variation. Ruan Jian Xue Bao/Journal of Software, 2025, 36(10): 4590–4611 (in Chinese). http://www.jos.org.cn/1000-9825/7309.htm

Vulnerability Sample Generation Method Based on Abstract Syntax Tree Variation

ZHENG Wei[1], LI Yun-Fan[1], GUI Kui[1], WU Xiao-Xue[2], CHEN Xiang[3], DENG Pei-Ran[1]

[1](School of Software, Northwestern Polytechnical University, Xi'an 710072, China)

[2](School of Information Engineering, Yangzhou University, Yangzhou 225127, China)

[3](School of Artificial Intelligence and Computer Science, Nantong University, Nantong 226019, China)

Abstract: With the continuous development of information technology, the quantity and variety of software products are increasing, but even high-quality software may contain vulnerabilities. In addition, the software update speed is fast, and the software architecture is increasingly complex, which leads to the gradual evolution of vulnerabilities into new forms. Consequently, traditional vulnerability detection methods and rules are difficult to apply to new vulnerability features. Due to the scarcity of zero-day vulnerability samples, zero-day vulnerabilities that appear in the software evolution process are difficult to find, which brings great potential risks to software security. This study proposes a vulnerability sample generation method based on abstract syntax tree mutation, which can simulate the structure and

* 基金项目: 国家自然科学基金 (62141208); 陕西省重点研发计划 (2021GY-041)
收稿时间: 2023-04-04; 修改时间: 2023-08-01, 2024-05-29; 采用时间: 2024-10-28; jos 在线出版时间: 2025-05-07
CNKI 网络首发时间: 2025-05-08

syntax rules of real vulnerabilities, generate vulnerability samples more in line with the actual situation, and provide a more effective solution for software security and reliability. This method analyzes the abstract syntax tree structure generated by Eclipse CDT, extracts the syntactic information in the nodes, reconstructs the nodes and abstract syntax trees, optimizes the abstract syntax tree structure, and designs a series of mutation operators. Subsequently, it performs mutation operations on the optimized abstract syntax trees. The method proposed in this paper can generate mutation samples with the characteristics of UAF and CUAF vulnerabilities, which can be used for the detection of zero-day vulnerabilities and help to improve the detection rate of zero-day vulnerabilities. Experimental results show that this method reduces the invalid sample size by 34% on average compared with the random variation method in traditional detection methods, and can generate more complex mutated samples. In addition, this method can generate more complex mutated samples, enhancing the coverage and accuracy of detection.

Key words: abstract syntax tree (AST); zero-day vulnerability; variation operator; vulnerable sample generation

在当今不断发展壮大的信息化社会中, 软件行业作为信息技术的核心, 在国民经济和社会发展方面发挥着重要的推动作用[1]. 随着信息企业的迅猛发展, 海量的软件产品不断涌现, 为人们带来了巨大的效益. 但即使是最高质量的软件也可能存在错误, 这可能导致软件失效. 在网络安全领域, 零日漏洞的检测尤为重要. 因此, 在软件开发和使用过程中, 必须高度重视软件的安全性和稳定性, 加强软件漏洞的检测和修复工作, 以确保网络安全和国家安全. 本文中所提到的零日漏洞特指未在各大漏洞网站或平台公开披露的漏洞, 由于它们没有公开补丁的安全漏洞, 因此具有更强的突发性与更大的破坏性. 目前基于网络的应用系统更多地采用了分布式、集群和可扩展架构, 使得软件的内部结构错综复杂. 零日漏洞的增长同软件复杂性、软件架构的演化呈现正相关关系.

软件漏洞是程序中的错误或缺陷, 而零日漏洞是在公开发现前已被恶意利用的未知漏洞. 当前软件漏洞的静态检测方式只能依据已知的漏洞类型进行重复性和被动性检测, 无法对架构演化导致的零日漏洞进行检测. 而以模糊测试为代表的动态检测方式近年来在发现零日漏洞方面取得了较大成功, 但其执行阶段性能开销较大且非常耗时, 无法对零日漏洞做出及时检测. 为了解决上述传统检测方式所存在的问题, 本文特别关注 UAF (use after free) 和 CUAF (concurrent use after free) 这两种漏洞, 提出基于抽象语法树变异的漏洞样本生成技术. UAF 和 CUAF 类型零日漏洞由于其难以检测和高危性质, 对软件系统安全构成严重威胁. 基于抽象语法树变异的漏洞样本生成方法首先通过对 Eclipse CDT 生成源漏洞代码抽象语法树节点结构的分析, 提取节点中的语法信息来重构节点和抽象语法树, 实现对抽象语法树结构的优化. 接着, 在谓词逻辑和统计分析的理论基础上分析软件漏洞演化规律, 得到一系列变异算子. 根据变异算子的变异规则在优化后的抽象语法树上实现变异操作. 最后使用本文提出的 MOPSG (mutation operators perform sequence generation) 算法对变异算子序列进行优化生成变异策略, 对源漏洞代码在抽象语法树层面执行变异策略, 最终得到蕴含漏洞的变异样本. 经过正则过滤和有效性确认后得到的有效变异样本可用于观察测试用例在固定类型漏洞上的有效性, 了解其效果和不足, 间接提高用例在检测此类漏洞时的能力. 整个路线的技术框架如图 1 所示, 该路线从漏洞演化特征和漏洞样本数据出发, 提出了针对具备 UAF 和 CUAF 类型零日漏洞特征的代码的静态模糊变异方法.

为了验证基于抽象语法树变异的漏洞样本生成技术得到的变异样本的有效性, 本文以多线程程序下数据竞争引发的一类安全漏洞展开. 首先对含有漏洞多线程程序进行漏洞特征语句插入处理, 推断数据竞争引发的 CUAF 漏洞模式对应的最优变异算子执行序列, 有效降低变异算子次序带来的无效样本数量. 然后筛选 Linux 内核公开代码中规模合适的且在时间线中含有 CUAF 漏洞的代码片段, 通过抽象语法树变异复现这些代码片段在时间线上触发该类型漏洞的过程, 通过与传统检测方法中的随机变异算子执行过程做横向对比, 传统检测方法由于难以发现尚未公开披露的零日漏洞, 往往效果不佳. 本文方法可以生成包含漏洞特征的变异样本, 作为零日漏洞检测的输入, 从而提高零日漏洞的检测率.

本文第 1 节介绍目前软件漏洞检测和变异测试的相关研究工作. 第 2 节介绍本文的研究内容. 第 3 节介绍基于抽象语法树变异的漏洞样本生成方法的具体实现. 第 4 节通过实验来证明本文提出方法的有效性. 最后总结全文.

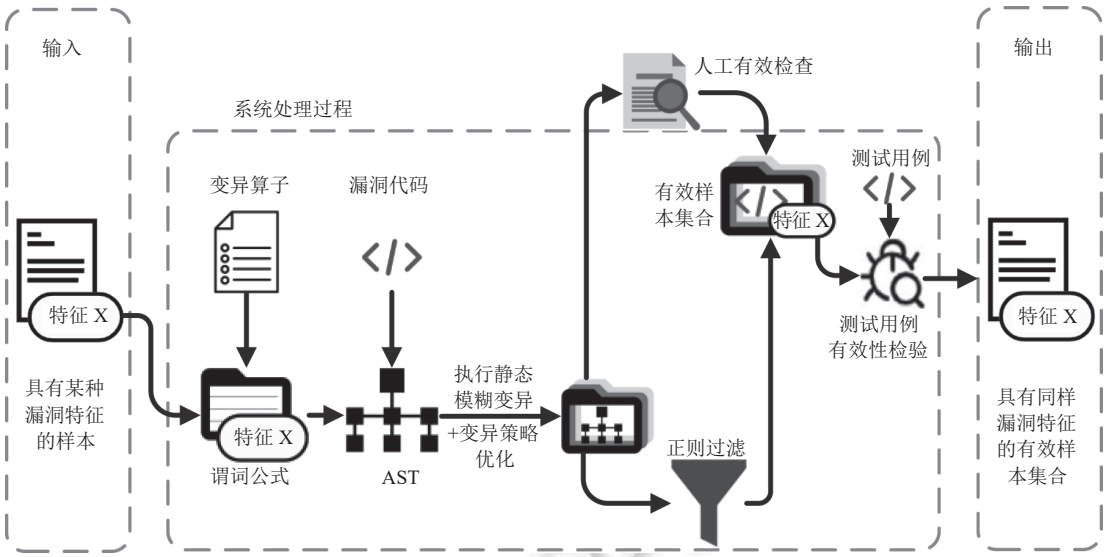

图 1 基于抽象语法树变异的漏洞样本生成方法流程图

1 相关研究工作

1.1 传统软件漏洞检测方法

模糊测试是当前较为成熟的软件漏洞检测方法. You 等人[2]提出一种实时输入类型探索方法, 通过轻量级随机模糊探测算法引导变异种子的演化, 以尽可能发现软件系统潜在的漏洞. Wen 等人[3]针对内存相关漏洞, 提出内存引导的模糊测试技术 MemLock, 以生成过多的内存消耗输入并触发不受控制的内存消耗错误. Wang 等人[4]针对 UAF 漏洞, 提出漏洞状态导向的模糊测试工具 UAFL, 通过执行静态 typestate 分析以查找可能违反该属性的操作序列, 然后, 模糊处理过程以操作序列为指导, 逐步生成触发属性违规的测试用例. Gan 等人[5]提出一种覆盖率敏感的模糊解决方案 CollAFL, 通过提供更准确的覆盖信息来减轻路径冲突, 同时仍保持较低的插桩开销, 其原型实现基于广泛使用的模糊工具 AFL. Li 等人[6]提出模糊测试方法 Cerebro, 其基于在线多目标算法平衡各种指标(例如代码复杂度、覆盖率) 的模糊种子选择方法, 通过度量未覆盖代码的复杂度, 从而设计出动态更新算法. 总之, 模糊测试实现简单, 具有高度可扩展性, 是一种广泛应用的软件漏洞检测方法, 多种研究已经针对不同的目标和场景提出了各种模糊测试技术, 如实时输入类型探索、内存引导的模糊测试技术等. 尽管模糊测试在某些场景下表现出色, 但它仍然存在一些局限性. 首先, 为了获得有效的结果, 它需要大量的测试输入, 这导致了巨大的性能开销和长时间的测试周期. 其次, 模糊测试对专业知识的依赖强烈, 增加了人工成本. 最后, 它在及时应对零日漏洞方面存在困难.

符号执行是一种利用中间语言进行漏洞检测的技术, 它结合了符号执行和约束求解的思想. Liang 等人[7]提出了一些基于 LLVM 中间表征的静态分析工具, 这些工具可以使用符号执行和 Z3 SMT 解释器来发现软件中的一些缺陷, 如除零错误、指针溢出和死代码等. 同时, Cassez 等人[8]提出了一种能够进行静态检测工具, 能检测程序中的一些特定错误块是否能够被访问到. 此外, Thome 等人[9]还提出了一种基于搜索驱动的约束求解技术, 通过混合约束求解和蚁群优化元启发式算法来解决现有字符串求解器的不足. 沈维军等人[10]提出了一种基于动静态相结合的程序分析和符号执行技术, 它通过提取数值变量的符号式、进行静态攻击流程分析和高精度动态攻击验证来检测和分析软件中可能存在的数值稳定性相关的安全漏洞. 总之, 基于符号执行的漏洞检测方法能够生成触发漏洞的具体输入, 随后利用符号执行工具生成的输入验证并分析漏洞, 从而发现除零错误、指针溢出等缺陷. 然而, 符号执行在处理大型程序时效率低下, 且常因路径爆炸问题难以覆盖所有执行路径.

代码相似性检测是一种主要用于检测代码克隆[11]导致的漏洞的方法, 即认为相似的程序可能存在相同的漏洞[12]. 相关研究采用了不同的方法来识别漏洞, 如基于词令牌[13]、代码文件行标记[14]、代码中的函数名和类型名[15]进行向量比较或近似匹配, 或利用抽象语法树[16]、图[17]、子图同构匹配[18]来表示和查找漏洞代码段中的语法结构或语义特征. 总之, 代码相似性检测是一种主要用于检测由代码克隆导致的漏洞的方法, 它基于代码的结构或语义特征来查找和比较相似的代码段. 尽管代码相似性检测在某些场景下有效, 但它的适用范围有限. 对于高难度克隆, 如函数名称改变或代码结构不同但语义相似的情况, 其检测准确率较低.

规则检测是一种能够对非克隆代码进行漏洞检测的方法, 通过安全知识领域专家手工定义漏洞规则, 利用数据流分析和静态污点分析[19]等技术标记程序状态, 分析跟踪的数据信息[20]是否会影响某些程序操作, 进而检测到程序漏洞. 早期出现的 Flawfinder、RATS、ITS4[21]、Clang 等开源工具使用了较为简单的漏洞规则和解析器, 在实际使用时导致误报率和漏报率过高. 基于规则的商业检测产品也陆续出现, 包括 Checkmarx、Fortify、Coverity[22]等. 总之, 规则检测是一种通过专家定义的漏洞规则来检测非克隆代码中的漏洞的方法, 它结合了数据流分析和静态污点分析等技术来标记和分析程序状态. 尽管基于规则的检测方法可以精确地定位到漏洞行, 但它仍然存在一些问题. 首先, 它高度依赖于专家的主观判断来定义检测规则, 这可能导致一些误报和漏报. 其次, 由于规则的固定性, 它可能难以检测出新出现的或未知的零日漏洞.

1.2 基于深度学习的软件漏洞检测方法

随着神经网络技术的成熟和运算能力的提升, 基于深度学习的软件漏洞检测成为研究热点[23−29]. 由于基于深度学习的方法可以自动提取不同漏洞类型的特征, 不依赖于专家的手动提取, 为漏洞检测技术的发展带来了契机. Li 等人[30]提出将深度学习用于代码漏洞检测中, 并指出表征形式的重要性, 其所设计的漏洞检测工具 VulDeePecker 采用基于语义信息的数据依赖图对代码进行表征, 使用 BiLSTM 模型对库/API 函数进行检测. 针对 VulDeePecker 存在的不足 (只适应一种漏洞类型、只采用数据依赖进行代码表征、只采用了一种深度学习模型), 该团队进一步提出 SySeVR[26], 一个基于深度学习的漏洞检测系统框架, 支持基于语法和语义相结合的代码表征形式, 通过辅助构建的数据集对该框架实用性进行了验证, 其检测性能指标 $F1$-Score 可达 0.95 以上. 相对于 VulDeePecker, SySeVR 可支持不同的深度学习模型, 并且理论上可以支持所有类型的漏洞检测 (只要有相应数据集).

Cheng 等人[24]采用图卷积网络将代码片段嵌入低维表征形式中进行尝试, 该表征形式保留了漏洞代码的控制流信息, 实验结果表明该方法优于 VulDeePecker[30]以及基于词嵌入方法的深度学习模型效果.

Gao 等人[27]提出二进制文件漏洞检测工具 VulSeeker, 将控制流图和数据流图相结合进行语义表征, 通过深度神经网络模型进行块级特征 (例如库函数调用次数、逻辑运算次数) 提取并生成函数嵌入向量, 最后通过余弦距离来测量目标函数与训练样本函数的相似性, 进而实现漏洞函数检测. 实验结果显示, VulSeeker 的漏洞检测能力显著优于基准方法 Gemini.

Zhou 等人[28]采用抽象语法分析树、数据流图、控制流图以及词法序列这 4 种不同代码表征形式, 提出基于通用图神经网络的模型 Devign, 通过 Conv 模块从表征图节点信息中抽取特征进行图级分类. 为验证模型有效性, 其团队基于实际 C 语言开源项目花费大量时间手动标注两个公开数据集, 实验结果表明, Devign 优于 3 种基准模型 3-layer BiLSTM、3-layer BiLSTM+Att 和 Metrics+XGBoost.

Russell 等人[29]开发了一个大规模函数级漏洞检测系统, 该系统主要针对 C/C++开源代码漏洞的检测, 通过深度表征学习对代码进行词法解析. 基于实际软件项目数据和源于 NIST、SATE IV 等数据集对深度表征学习方法进行验证, 最大 $F1$-Score 可达 0.80, 显著优于漏洞检测工具 Clang、Flaw 和 Cppcheck.

Chakraborty 等人[31]进一步对 VulDeePecker、SySeVR、Devign 等主流技术进行了表征方式、数据分布、模型选择等方面的比较, 发现工具的性能受数据重复、类不平衡的不利影响, 导致模型学习到大量漏洞无关的特征, 最终这些工具在实际项目评测时的性能大约下降了 50%, 这也直接反映了当前基于深度学习的漏洞检测方法存在的问题.

与传统漏洞检测方法相比, 已有研究表明当前基于深度学习的漏洞检测技术, 在检测准确性和效率方面有所

提升. 但是在迁移到实际项目时会导致准确率大幅度下降. 由于现阶段零日漏洞数据样本的稀缺, 导致该方法很难对零日漏洞进行有效检测.

1.3 变异测试

变异测试是一种常见的软件测试技术, 它通过修改源代码中的语句或表达式, 生成一组具有不同程度变异的变异体来测试用例的质量. 这种测试方法能够检测出源代码中的潜在错误和缺陷, 并帮助开发人员和测试人员改进测试用例的质量. 但是目前变异测试在软件测试中的应用现状和研究进展中仍然存在着一些问题.

变异测试在软件测试中已经得到广泛的关注和应用. 一些研究表明, 变异测试能够检测到源代码中的错误和缺陷, 能够提高测试用例的质量和代码覆盖率, 从而提高软件的质量和稳定性. Beaman 等人[32]研究表明, 变异测试能够有效地发现软件中的漏洞和缺陷, 对软件的安全性和稳定性有重要的意义. Aayush 等人[33]阐明了变异测试对于提高软件测试质量和效率的重要作用, 为变异测试的研究者和实践者提供了一个有价值的参考.

然而, 变异测试也存在一些挑战和问题. 一方面, 变异测试生成的变异体数量很大, 导致测试的时间和资源开销很大, 需要设计高效的测试方法和策略. 另一方面, 不同的变异算子对测试结果的影响有所不同, 需要选择合适的变异算子和测试用例集合, 以达到最佳的测试效果. Ma 等人[34]研究发现, 有些变异算子不是很有效, 需要选择合适的算子来提高测试效果.

近年来, 一些新的变异测试研究方法和工具被提出, 以应对变异测试中的一些问题和挑战. 例如, Jia 等人[35]总结了变异测试的发展趋势和应用领域, 提出了一些新的研究方向和挑战. Khanfir 等人[36]提出了基于预训练语言模型的变异测试技术, 用于生成自然和强大的变异体, 提高故障检测能力.

综上所述, 由于零日漏洞尚未被公开披露, 传统的漏洞检测方法很难检测到这些漏洞. 目前的变异测试生成的变异体数量很大, 导致测试的时间和资源开销很大. 针对这些问题, 本文方法可以生成包含漏洞特征的变异样本, 这些样本可以作为零日漏洞检测的输入, 有助于提高零日漏洞的检测率. 此外, 本文方法的创新点在于使用静态模糊技术来生成变异样本, 相较于传统的动态模糊技术, 可以减少生成变异样本的成本, 并且可以生成更加复杂的变异样本, 从而提高检测的覆盖率和准确率. 因此, 本文的研究对于零日漏洞的检测具有重要的意义, 并且可以为现有工作带来重要的贡献.

2 利用抽象语法树变异生成漏洞样本的研究

漏洞样本生成方法在漏洞检测和安全评估中具有重要的作用. 通过生成包含漏洞特征的变异样本, 可以帮助安全测试人员和研究人员发现和修复软件漏洞, 提高软件系统的安全性. 特别是在零日漏洞检测方面, 漏洞样本生成方法可以生成包含零日漏洞特征的变异样本, 提高零日漏洞的检测率.

然而, 漏洞样本生成方法面临着许多挑战. 首先, 随着软件架构的演化, 软件系统的动态复杂性不断增加, 导致传统的漏洞检测方法和规则难以适应新的漏洞特征. 其次, 由于零日漏洞样本的稀缺, 传统的漏洞检测方法很难检测到这些漏洞. 因此, 需要开发新的漏洞样本生成技术来应对这些挑战. 图 2 展示了本文的主要研究内容.

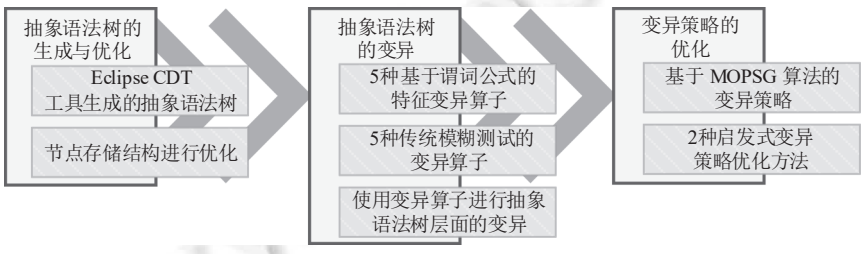

图 2 基于抽象语法树变异的漏洞样本生成方法研究点

1) 抽象语法树的生成与优化. 本文的变异操作是在抽象语法树层面进行的, 因此对多种抽象语法树的生成工具进行了研究, 对每种工具生成抽象语法树的解析方式和生成结果进行了深入的了解, 经过仔细地研究对比后, 本

文选择 Eclipse CDT 作为抽象语法树的生成工具, 并针对 Eclipse CDT 生成的抽象语法树的节点存储问题进行了优化.

2) 抽象语法树的变异. 本文通过对 UAF 到 CUAF 两种漏洞之间的演化规律进行研究得出了 5 种包含漏洞特征的变异算子. 将这 5 种变异算子和 5 种普通变异算子组合在一起, 研究他们在抽象语法树层面的变异规则和变异方式, 最终实现在抽象语法树层面的变异.

3) 变异策略的优化. 无效样本指的是在变异过程中生成的样本中, 不满足目标条件或者不包含期望的特征的样本. 由于传统模糊变异测试在使用变异算子变异时采用随机变异的方式, 生成的无效变异体比较多, 效率低. 本文提出一种根据关联度来对不同变异算子的执行频率进行选择的算法 MOPSG. 该方法的核心是根据每个变异算子权重动态调整变异算子执行次数和顺序, 找到最佳的变异算子执行次序后进行抽象语法树级别的变异, 从而针对某一漏洞模式推断出其漏洞演化路径, 满足在产生尽可能少的样本量的情况下涵盖漏洞样本. 在该方法的基础上对执行变异策略过程中通过采用固定关键算子顺序、部分子树截断两个启发式方法对变异顺序和次数进行限制, 有效降低无效变异样本数量.

2.1 基于抽象语法树的漏洞样本生成方法

本节首先对 Eclipse CDT 工具生成的抽象语法树进行深入分析, 研究其结构和使用所存在的问题, 针对这些问题对抽象语法树进行优化. 然后对 UAF 漏洞和 CUAF 漏洞之间的演化规律进行总结得出 5 种漏洞特征相关的变异算子, 再加上模糊变异中的 5 种普通变异算子一共 10 种变异算子. 根据这 10 种变异算子的变异规则, 对抽象语法树进行不同的变异操作, 最终生成变异体. 整体工作流程如图 3 所示.

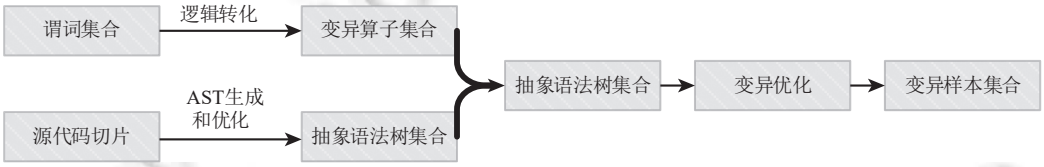

图 3 抽象语法树变异和优化流程

2.2 基于 Eclipse CDT 抽象语法树的优化

由于使用 Eclipse CDT 无法直接生成抽象语法树文本, 而是得到经过解析后的编译单元 IASTTranslationUnit, 这个编译单元中包含了解析源程序过后的全部信息, 不仅仅包括语法树, 还有注释信息、头文件、偏移量等其他信息, 这些信息对后面抽象语法树的节点分析和变异没有作用, 还占用语法树的存储空间. 另外, IASTTranslationUnit 编译单元获取子节点的方式为 IASTNode[] children = node.getChildren(), 从代码中可以看出, 一个 node 节点通过 getChildren 方法获取它的子节点, 而其子节点的存储方式为对象数组 IASTNode[], 使用这种存储方式来对节点进行插入删除操作需要移动大量数组元素, 在变异的过程中效率比较低. 本文针对以上问题, 设计了一种方法来对原本编译单元里节点存储结构进行优化, 每个节点除了保存原抽象语法树节点中的全部语法信息外, 再分配一个抽象语法树中唯一的数字标志值用于变异时对节点进行定位, 最后将各节点存储到链 4 式结构的抽象语法树中.

对 Eclipse CDT 抽象语法树的具体优化过程如算法 1 所示, 在执行算法之前会自定义一个节点结构, 包括节点 *id*, 节点名 *name*, 节点值 *value* 以及一个子节点列表 *sonList*. 该优化算法的输入抽象语法树 T, 输出为抽象语法树 T′, 算法中主要包括了 3 个方法, 第 1 个是 setNode 方法, 该方法是将原抽象语法树的节点作为输入参数, 通过节点获取到节点名 *NodeName* 和节点值 *NodeValue*, 根据节点名和节点值生成一个节点 *id*, 最后将这 3 个值输入到新的节点结构中并返回一个新节点 (算法中 1–7 行). 第 2 个方法为 getTree, 该方法是对编译单元 IASTTranslationUnit 的根节点下的子节点进行深度优先遍历, 对遍历访问到的每一个正常节点, 首先获取该节点的子节点列表, 作为下一次深度优先遍历的输入, 然后对该节点使用 setNode 方法获取一个新节点, 将新节点 *newNode*, 根节点 *rootNode*, 以及当前节点的父节点 *curNodeFatherId* 都输入到 insert 方法中来构建新的抽象语法树, 当所有节点

遍历完成后, 一棵新的语法树就构建完成了. 第 3 个方法为 insert, 这是将新节点插入到语法树中的方法, 该方法的参数分别为根节点 *rootNode*, 新节点的父节点 *fatherId* 以及一个新节点 *newNode*, 从根节点进行判断当前节点的 *id* 与 *newNode* 节点的父节点的 id 是否一致, 如果一致就将 *newNode* 节点插入到当前节点的子节点列表 *sonList* 中, 否则从根节点的子节点列表开始深度优先遍历每一个节点进行判断. 总体的过程为先建立新的抽象语法树根节点, 然后遍历编译单元中的每一个正常节点, 将每个节点转化为一个新的节点插入到新的抽象语法树中. 通过该算法得到的抽象语法树不仅包含了原本抽象语法树的全部语法信息, 而且结构更加简单, 其链式存储方式为后面的变异操作提供了方便.

算法 1. 抽象语法树优化算法.

输入: 抽象语法树 T;
输出: 优化后的抽象语法树 T′.

```
1. function setNode(curNode)
2.         rootNodeName ← T.getName
3.         rootNodeValue ← T.getValue
4.         rootId ← setId(rootNodeName, rootNodeValue)
5.         newNode ← setNode(rootId, rootNodeName, rootNodeValue)
6.         return newNode
7. end function
8. rootNode ← T′.buildrootNode (rootNodeId, rootNodeName, rootNodeValue)
9. while i = 1 to n do
10.        int index ← 1
11.        ti ← getTree(ni, index, sign)
12.        T.add(ti)
13. end while
14. function getTree(rootNode, curNode)
15. childrens ← curNode.getChildren
16. newNode ← setNode(curNode)
17. curNodeFatherId ← curNode.getFatherId
18. T′.insert (rootNode, curNodeFatherId, newNode)
19. for all node ∈ childrens do
20.            getTree(rootNode, childrens)
21.        end for
22. end function
23. function insert(rootNode, fatherId, newNode)
24.        if rootNode.id is fatherId and rootNode.allChilds.notExist(newNode) then
25.            rootNode.sonList.add(newNode)
26.        end if
27.            sonList ← root.getSonList
28.            if sonList is not null then
29.            for all node ∈ sonList do
30.                getTree (node, fatherId, newNode)
```

```
31.                 endfor
32.             endif
33. end function
34. return T′
```

2.3 变异算子生成

随着软件架构的变化, 攻击方式也在不断变化, 从而导致了软件漏洞发生类似 UAF → CUAF 的演化. 本节用谓词逻辑和统计分析的技术方法, 为 UAF 和 CUAF 漏洞特征构建不同的谓词公式, 公式中的个体词代表该漏洞产生的关键位置, 谓词代表漏洞中一个或多个关键位置所蕴含的联系, 以此来构建已有漏洞类型的谓词公式集合. 对集合中的不同个体词和谓词进行重组, 不同的组合方式蕴含了不同软件漏洞之间的演化规律, 再以新的谓词公式集合的形式存储下来, 从而有效指示了零日漏洞的潜在特征. 例如, 在 UAF 演化生成 CUAF 漏洞过程中, 将目标函数在多线程环境线程调用对应一个谓词公式, 映射为演化规律变异算子后对应操作为在目标函数体中增加线程初始化语句. 根据谓词公式集合可转化得到一系列演化规律变异算子, 再结合模糊测试的变异算子得到变异算子序列. 具体的变异算子序列如表 1 所示, 表中前 5 个变异算子为包含演化规律的变异算子, 后 4 个为传统模糊变异测试的变异算子.

表 1 变异算子及其作用

变异算子	作用
AT (add thread and function additions)	增加线程初始化语句
LTGV (local variable to global variable)	将局部变量与全局变量转换
RJ (replace the join operations)	替换join子树
AM (add mutex operations)	增加互斥锁声明语句
AS (add signal operations)	增加并发机制的信号量
LCR (logical connector replacement)	替换逻辑连接符
ROR (relational operator replacement)	替换关系运算符
SD (statement deletion)	按序删除语句
SPS (statement position swap)	按序互换语句位置

变异算子的设计目标是基于已有漏洞引入随机变异生成新的漏洞, 这些漏洞共同组成漏洞样本集合, 其中与原漏洞类型相同的生成漏洞样本集合被称为有效样本集合. 通过变异算子序列优化实验, 结果表明更高频次的算子投入能够更加有效地使有效变异样本快速出现, 证明了这些算子拥有符合预期的效果.

为了确保无效样本的检测准确性, 我们在每轮次中随机抽取一定比例的样本进行详细检查, 通常检查约 30% 的变异样本, 这一比例基于前期实验和经验数据确定. 我们意识到由于人工检查的工作量较大, 确实可能存在因懒得检查而导致无效样本比例偏小的情况. 为了减少这种情况的影响, 我们引入了以下措施: 交叉验证, 每轮次的样本由不同的实验人员进行交叉验证, 确保样本检查的全面性和准确性; 重复检查, 对于关键轮次的样本, 进行重复检查, 以验证初次检查结果的可靠性; 自动化辅助, 引入部分自动化脚本, 辅助人工进行样本的初步筛查, 提高检查效率. 尽管如此, 我们承认在实际应用中, 仍可能存在少量误报和漏报的情况. 这些问题在方法有效性分析中得到了充分考虑. 我们将继续探索和引入更自动化和智能化的验证方法, 以进一步提高变异样本检测的准确性和效率.

2.4 抽象语法树变异

本节介绍了针对给定的变异算子序列进行抽象语法树层面变异的方法. 根据变异算子对节点操作方式的不同, 可将变异执行方式分为两类. 第 1 类变异需要修改抽象语法树中的子树结构, 包括对子树进行增删改操作, 优先寻找语句节点以减少遍历复杂度. 共有 7 种属于此类的变异算子, 分别为 AT、LTGV、RJ、AM、AS、SD 和 SPS. 第 2 类变异不需要改变抽象语法树节点的位置结构, 只需要改变其节点值. 为此需要遍历特殊的表达式节点, 去寻找符合变异规则的节点值, 通常需要遍历到叶子节点. 共有 3 种属于此类的变异算子, 分别为 LCR、ROC 和

AOR. 下面将介绍 AT, LTGV, SD, LCR 这 4 种变异算子的具体变异过程, 由于剩余变异算子的变异形式类似上述 4 种的其中一种, 所以不再详细描述. 本文实验所使用的数据和代码已进行了开源 (https://github.com/Static-bugs-Detection-Club).

在介绍算子之前, 简单解释本文所使用的抽象语法树 (abstract syntax tree, AST) 中节点的含义:

a) CASTCompoundStatement: 这个节点可能代表一个复合语句, 通常在 C 或 C++中表示一个由大括号包围的代码块;

b) CASTExpression: 代表一个表达式, 如算术表达式或逻辑表达式;

c) CASTFunctionDeclaration: 代表一个函数声明;

d) CASTIfStatement: 代表一个 if 语句;

e) CASTLoopStatement: 代表循环语句, 如 for 或 while 循环;

f) CASTDeclarationStatement: 代表一个声明语句, 声明一个变量或常量, 可能包括其类型、名称和初始值;

g) CASTSimpleDeclaration: 代表一个简单的声明, 只涉及基本类型的声明, 如整数、浮点数或字符, 而不包括复杂的数据结构或类的声明.

(1) AT 变异算子

AT 变异算子是变异过程中优先级最高的算子, 必须首先使用该算子变异抽象语法树, 该算子的作用是添加线程. 第 1 步, 通过 Eclipse CDT 将原始代码转换成抽象语法树, 获取所有函数名. 然后在 AT 变异中需要创建函数节点及其子树, 包括线程声明语句节点和线程函数调用节点, 并将其添加到原始代码的 main 函数子树下方. 如果没有 main 函数, 则选择文件中逻辑上适合作为多线程操作入口的其他函数. 这些函数通常是执行重要逻辑或调度任务的函数. 第 2 步, 将整个 main 函数子树或选定函数子树添加到原始代码的语法树中, 使其成为其他函数节点的兄弟节点. 如图 4 所示, 第 1 步时相应的子树创建完成之后, 将子树全部添加到 CASTCompoundStatement 节点的下方. 而第 2 步将整个 main 函数子树添加到原来的语法树中, 在整个语法树中 main 函数节点与其他函数节点为兄弟节点.

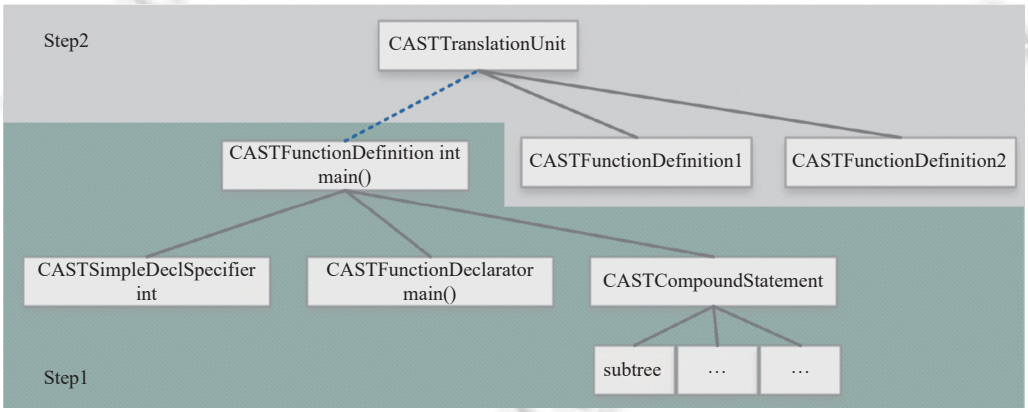

图 4 函数节点及其子树

该算子适用场景: 用于需要增加并发执行线程的场景, 特别是在多线程编程中检测和触发 UAF 和 CUAF 类型漏洞. 该算子主要目的: 通过增加线程来模拟多线程环境, 从而增加漏洞样本的复杂性和多样性. 该算子输入内容为原始代码的 AST, 输出内容为增加线程后的 AST.

(2) LTGV 变异算子

变异算子 LTGV, 作用是将局部变量转换成全局变量. 在 Eclipse CDT 工具生成的抽象语法树中, 局部变量的节点名为 CASTDeclarationStatement, 即图 5(a) 处的橙色节点, 而全局变量的节点名为 CASTSimpleDeclaration, 即图 5(b) 中深绿色背景子树的父节点. 这两种变量在抽象语法树中的表示不仅仅是名称不一样, 它们在语法树中的

位置也是有差异的. 局部变量基本上都是在函数中声明的, 它们所在的位置都会在函数节点的子节点 CASTCompoundStatement 中, 而全局变量所在位置是在类里面以及函数之外, 所以在该变异过程中会将函数中的局部变量节点及其子树进行剪切, 将原本的局部变量变异成与函数节点为兄弟节点的全局变量. 局部变量和全局变量在抽象语法树中的名称和位置以及变异前后的变化都如图 5 中 (a) 和 (b) 过程所示.

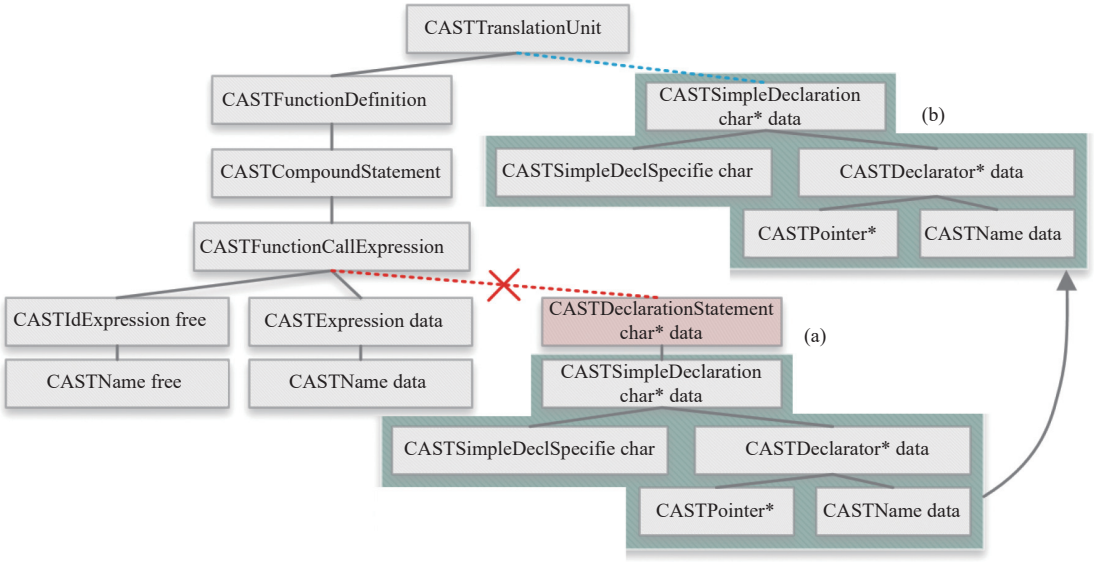

图 5 抽象语法树的局部变量全局化操作

该算子适用场景: 用于需要将局部变量提升为全局变量的场景, 以便在不同作用域之间共享变量, 模拟复杂的变量使用场景. 该算子主要目的: 通过将局部变量变为全局变量, 增加变量的生命周期和作用域, 提升漏洞样本的复杂性. 该算子输入内容为原始代码的 AST, 输出内容为局部变量转换为全局变量后的 AST.

(3) SD 变异算子

SD 变异算子的变异规则为删除语句, 这里默认为只删除一条语句的变异操作. 在代码中删除语句的操作一般为删除某一行, 这种操作比较容易产生错误的变异体, 比如在 for 循环语句中, 循环体会有很多语句, 而在按行的方式进行删除时可能恰好把 for 所在的那一行删除了, 那么这个变异体就是无法编译的无效变异体. 在抽象语法树中, 删除语句是按照语句节点来删除的, 语句节点作为根节点, 它的子树中包含了该语句节点的全部信息, 在删除语句节点时, 会将以语句节点作为根节点的子树都给删掉. 而 for 循环语句节点的子树包括循环体里的全部内容, 所以在抽象语法树层面删除 for 循环语句时会将整个 for 循环里的内容全部删掉. 选定需要删除的点主要依据以下几个规则: 优先选择删除那些在语法树中独立存在且对周围代码影响最小的语句, 如独立的函数调用或变量声明语句; 优先删除非控制流语句, 如简单的赋值语句或函数调用语句, 避免直接删除控制流语句如 if、for 等; 删除的位置应该避免影响代码的主要逻辑结构, 如避免删除主函数中的核心逻辑或关键路径上的语句; 在满足以上规则的前提下, 可以引入一定的随机性来选择删除的点, 增加变异样本的多样性. 虽然删除普通的语句也有可能造成代码出现问题, 但从抽象语法树层面进行删除变异产生的无效变异体会比从代码层面变异少很多. 删除子树的效果如图 6 所示, 图 6 将 free(data) 左侧深色背景的 sem.wait() 子树进行了删除.

该算子适用场景: 用于需要删除语句的场景, 尤其是在简化代码逻辑或去除冗余代码的情况下. 该算子主要目的: 通过删除特定的语句, 减少代码复杂度, 同时可能引入新的缺陷. 该算子输入内容为原始代码的 AST, 输出内容为删除目标语句后的 AST.

(4) LCR 变异算子

LCR 变异算子替换逻辑链接符的规则是将代码中的逻辑运算符“&&”换成“||”, 或者将“||”换成“&&”, 同时将“!”

删除. 在抽象语法树中, 只需要寻找包含逻辑运算符的语句节点 CASTIfStatement, 然后替换其中的逻辑运算符节点的值. 这个变异算子只会修改节点的值, 而不会改变节点的类型或子树结构. 替换逻辑连接符在抽象语法树中的实现效果如图 7 所示, 将包含逻辑连接符&&表达式节点中的值进行替换后, 该节点会变成图中深色节点的样子.

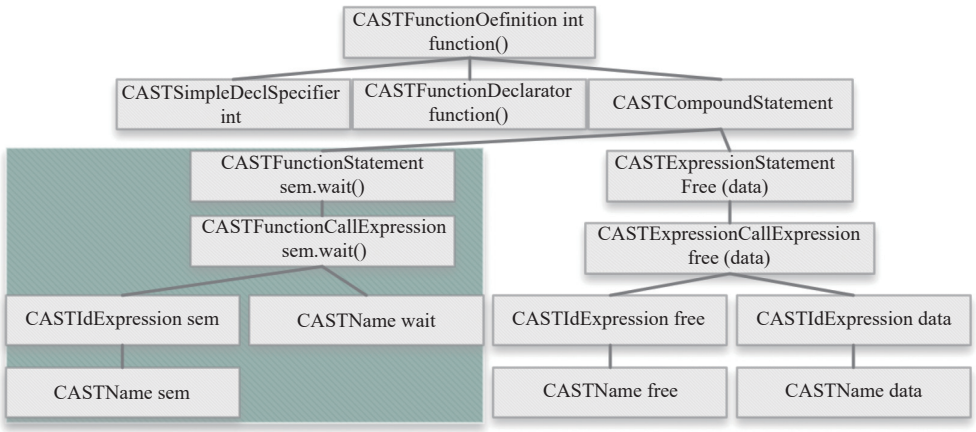

图 6 删除子树

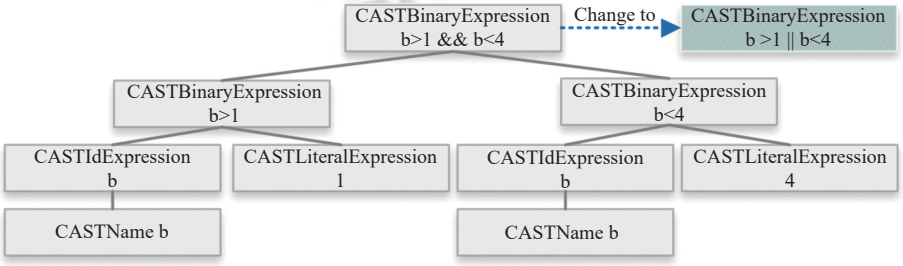

图 7 替换逻辑连接符

该算子适用场景: 用于替换逻辑运算符的场景, 特别是在需要改变逻辑判断条件时. 该算子主要目的: 通过替换逻辑运算符, 改变逻辑判断条件, 增加代码的执行路径. 该算子输入内容为原始代码的 AST, 输出内容为替换逻辑运算符后的 AST.

2.5 基于 MOPSG 算法的变异策略

本文提出了一种基于变异算子关联度引导的次序变异方法 MOPSG. 根据关联度来对不同变异算子的执行频率进行选择, 其中初始关联度由变异算子的权重经过归一化得出. 该方法核心是根据每个变异算子权重动态调整变异算子执行次数和顺序, 找到最佳的变异算子执行次序后进行抽象语法树级别的变异, 从而针对某一漏洞模式推断出其漏洞演化路径, 满足在产生尽可能少的样本量的情况下涵盖漏洞样本. 变异算子执行次数的优化函数如公式 (1) 所示.

$$\begin{cases} \sum_{i=1}^{i=I} x_i = X \\ F(X) = \sum_{i=1}^{i=I} P_i \times x_i \end{cases} \tag{1}$$

其中, X 为变异算子被执行总次数, I 为变异算子序列中算子的总数, 每种变异算子可以用序号 i 进行标注, x_i 为第 i 种变异算子被执行的次数, P_i 为第 i 种变异算子的变异关联度, $F(X)$ 为优化判断值.

使用 CUAF 漏洞的代码切片和全排列删除其中 1–n 阶语句的回溯代码作为样本. 对这些样本进行变异, 每次变异都记录所有算子的执行情况, 根据上一次变异计算判断值, 改变算子执行次数以优化至较大值. 值得一提的

是, 针对 CUAF 漏洞得到的演化规律变异算子每次变异都是在所有可执行分支都执行并生成样本. 所以对这类变异算子的高阶变异是没有意义的, 反而会徒增低阶变异样本数量.

对每轮变异都使用位置向量来记录每个变异算子执行位置, 其中第 1 个变异算子执行位置对应向量分量值为 1. 根据第 k 轮变异后更新的权重和公式 (1) 的优化, 将得到变异算子在第 k+1 轮变异构造出漏洞代码切片时的位置向量, 执行 n 轮变异后根据公式 (2) 调整各变异算子执行位置.

$$\begin{cases} E_{x_i+}=\sum_{k=1}^{s}E_{x_i/s}^{k} \\ E_{x_i-}=\sum_{k=1}^{f}E_{x_i/f}^{k} \\ R_{x_i}=\left\|E_{x_i+}-E_{x_i-}\right\| \end{cases} \tag{2}$$

其中, E_{x_i+} 表示 n 轮变异中出现包含漏洞的样本时总样本数量较少的前 s 轮 (s 可自由定义, 但不超过 n 的一半), E_{x_i-} 表示 n 轮变异中出现包含漏洞的样本时总样本数量较多的前 f 轮, 根据变异算子权重的大小重新调整变异算子执行顺序, 最终得到变异算子执行效率最高的若干个序列. 根据得到的优化变异序列, 对不包含漏洞的源代码样本进行抽象语法树变异, 可以得到该源码可能触发的本序列对应漏洞代码样本. 变异策略如算法 2 所示.

算法 2. MOPSG 算法.

输入: n 个变异算子权重 $\theta=\vartheta_0,\vartheta_1,\ldots,\vartheta_n$, 优化执行总轮次 α, 包含出漏洞的轮次 s, 未包含漏洞的轮次 f, 优化接口 χ;

输出: 变异算子执行序列向量集 $E_i=e_1,e_2,\ldots,e_i$, 变异算子执行次数上限集合 $X_i=x_1,x_2,\ldots,x_i$.

1. **while** θ is not null **do**
2. $\quad P_n \leftarrow \vartheta_n / \sum_{k=1}^{i}\vartheta_k$
3. **end while**
4. **for** $i=1 \rightarrow \alpha$ **do**
5. $\quad$ **if** is first mutation **then**
6. $\quad\quad$ **for** $i=1 \rightarrow$ Total number of mutation operators n **do**
7. $\quad\quad\quad P_i, x_i \leftarrow 1$
8. $\quad\quad$ **end for**
9. $\quad$ **else**
10. $\quad\quad \Gamma_j \leftarrow \sum_{k=1}^{n} P_k \times x_k$
11. $\quad\quad$ if $\Gamma_j > \Gamma_{j-1}$
12. $\quad\quad\quad$ **for** $i=1 \rightarrow$ Total number of mutation operators n **do**
13. $\quad\quad\quad\quad \chi$.FunctionTimesUpdate$(x_i)$
14. $\quad\quad\quad$ **end for**
15. $\quad\quad$ **end if**
16. $\quad$ **end if**
17. $\quad$ **for** $m=1 \rightarrow \sum_{k=1}^{n} x_k$ **do**
18. $\quad\quad$ Select a mutation operator ω according to $P_1,P_2,\ldots,P_i$
19. $\quad\quad$ Set the m component of position vector to $1 \Rightarrow \vec{e}_{\omega}^{\,j}(1,0,0,\ldots,0)(m=1)$
20. $\quad$ **end for**
21. **end for**

22. **if** $\alpha > 10 \cap s \neq 0$ **then**
23. **for** $i = 1 \rightarrow$ Total number of mutation operators n **do**
24. $e_{\omega+} \leftarrow \sum_{k=1}^{s} \overrightarrow{e}_{\omega}^{k} / s$
25. $e_{\omega-} \leftarrow \sum_{k=1}^{f} \overrightarrow{e}_{\omega}^{k} / f$
26. $R_i \leftarrow \| e_{\omega+} - e_{\omega-} \|$
27. χ.FunctionParameterUpdate(R_i, θ_i)
28. **end for**
29. **end if**

算法 2 的第 1–21 行主要利用变异算子的权重来选择算子执行和执行次数. 第 1 次执行输入相同的初始变异算子权重和执行次数限制, 之后每次迭代都使用新的变异算子权重. 输出是每个变异算子的执行次数. 值得一提的是, 由于该算法在 CUAF 漏洞演化研究中用于生成变异样本, 样本量较小, 因此生成更高阶的变异只会增加程序的复杂度, 正因为这个原因, 我们默认为这部分分配了较小的值. 然而, 当算法针对涉及较长代码跨度的漏洞类型时, 需要为 x_i 分配较高的初始值, 以确保突变能够覆盖更极端的情况. 优化函数 χ.FunctionTimesUpdate(x_i) 通过改变 x_i 来使得 $F(X)$ 的值接近局部最大值, 因为 $F(X)$ 的值不可以无限制地增长. 这是因为变异算子执行超过一定次数可能导致变异体偏离现实世界中可能发生的情况.

该算法的第 22–29 行旨在调整变异算子的各自权重, 以适应特定类型的漏洞形成模式. 这是通过为与选择概率 P_i 相对应的权重设置变异算子执行的次数限制来实现的. 接下来, 算法将变异算子执行过程向量化. 然后, 它重复执行突变, 根据生成的无效样本的比例为每个变异算子找到最佳向量. 最后, 函数 χ.FunctionParameterUpdate(R_i, θ_i) 会在每轮执行后根据向量的绝对差异更新变异算子的权重. MOPSG 算法需要知道已经产生的变异样本是否是有效的漏洞, 判定有效的过程由正则匹配过滤和人工观察判断结合完成.

2.6 变异策略优化

本文在抽象语法树层面执行变异算子的变异过程中, 为了解决变异样本数量爆发式增长的问题, 本节提出了两种启发式方法进行优化. 第 1 种是固定关键算子顺序, 这可以避免一些不必要的变异操作在高阶变异过程中错误地介入. 例如, 在演化生成 CUAF 漏洞过程中, 不含有线程的变异样本是无法触发该漏洞的, 则优先执行增加线程变异算子能防止此类无效样本的产生. 但是, 随着变异过程的进行, 再次执行增加线程变异算子会显著降低效率, 因此需要在更新变异算子序列后对其进行人工调整. 第 2 种启发式方法是部分子树截断, 这可以限制变异顺序和次数, 从而降低无效样本的数量. 在执行变异算子时, 先将变异操作应用于抽象语法树的某个子树上, 然后根据其效果再决定是否应该继续执行该变异操作. 这样可以避免无效样本的产生, 同时提高变异操作的效率.

在利用算法 2 对样本代码进行变异来更新变异算子权重的过程结束后, 可以根据结果识别在多个轮次中权重一直保持较高水平的变异算子, 然后考虑固定该变异算子再进行测试. 如果固定顺序整体比例降低, 则足以证明该变异算子是演化形成漏洞应该介入的. 这种方法可以减少无效样本的数量, 同时提高演化算法的效率, 从而更快地找到合适的漏洞.

部分子树截断算法是针对某一类漏洞的特征与共性, 对变异样本的子树进行识别并选择性停止某些样本继续变异的过程. 该过程如算法 3 所示.

算法 3. 部分子树截断算法.

输入: 待变异的抽象语法树 T;
输出: 变异后的抽象语法树集合 $\mathrm{T}' = t_1, t_2, \ldots, t_n$.

```
1. while T .node is not null do
2.         if Node.id = CASTFunctionCallExpression and Node.valuecontainsfree then
3.                 freeNodeList.add(Node)
4.         end if
5. end while
6. if freeNodeList is null then
7.         t_i ← T.mutationOperator; T′← t_i
8.         return T′
9. else
10.        t_i ← T.mutationOperator
11.        if t_i is not contains freeNodeList.nodes then
12.                currentMutation.End
13.                T′ ← t_i
14.                return T′
15.        end if
16. end if
```

例如, UAF 和 CUAF 漏洞是在错误的时机对已经释放的内存进行了访问. 漏洞代码中可以看出 free 操作是该漏洞的重要特征之一, 但并不是所有的 UAF 漏洞代码都包含 free, 可以作为一个参考. 在抽象语法树层面执行变异算子序列变异之前, 会将源代码转化为抽象语法树, 在算法 3 中, 遍历抽象语法树寻找 free 操作, 如果存在 free 操作, 那么在每一次执行抽象语法树变异之前都去遍历寻找包含 free 的子树. 如果一个变体在变异过程中丢失了 free 关键字所对应分支, 就把接下来的对变异体的变异停止.

3 实验分析

所有实验都在一个服务器上进行, 该服务器配备有 Intel(R) Xeon(R) Silver 4 110 CPU 和 64 GB RAM.

3.1 变异算子序列优化实验

3.1.1 实验数据

部分是用于优化变异算子序列, 推断演化路径的数据, 该数据来源于 CVE (common vulnerabilities & exposures) 开源漏洞库中与 UAF 相关的 30 段代码切片. 本文根据切片跨度、对漏洞特征植入的便捷性和 UAF 代码 use 与 free 形成的相关性等方面筛选了 10 段进行后续操作, 最终本文基于机器处理大数据量的时间和存储效率的考虑, 选择了 5 段切片进行漏洞植入. 筛选过程如表 2 所示. 值得强调的是, 该筛选过程的最终评价指标是综合 3 个特征后选择的性价比最高的代码切片, 因为它们可以在尽可能少的代码语句中跨函数植入更多的 CUAF 漏洞特征. 这也正是漏洞形成的一大原因, 即由于改进代码性能 (将单线程代码改为多线程) 时误触发的漏洞. 这样含有大量 CUAF 漏洞特征语句的 UAF 漏洞代码切片一旦实现多线程化, 有极大风险会直接触发 CUAF 漏洞, 但是漏洞是否会对程序本身造成危害是待定的.

表 2 CVE 代码切片

代码切片序号	切片跨度 (行)	漏洞特征植入数量	use与free关系
CVE-2018-21008	43	4	多函数
CVE-2019-15917	16	3	单函数
CVE-2019-20934	31	3	多函数
CVE-2020-27835	32	4	单函数
CVE-2021-33034	33	3	多函数

3.1.2 变异算子序列优化测试

在对用于测试变异效果的代码切片进行变异之前, 首先需要得到一些最优变异算子序列. 这些最优变异算子序列是无漏洞或仅含 UAF 漏洞的公开代码切片, 经过插入特征语句后形成的 CUAF 漏洞代码切片. 对同一切片的两段代码进行抽象语法树变异至其变异样本覆盖了 CUAF 漏洞代码后, 对其变异过程进行多次回溯. 回溯过程中通过对每次变异样本覆盖了 CUAF 漏洞代码时变异算子执行顺序的记录, 根据 MOPSG 算法调整各个变异算子的权重. 除了第 2.3 节中由演化规律得到的 5 个变异算子外, 虽然其他变异算子不具有较强的针对 CUAF 漏洞演化的导向性, 但是为了保证能覆盖可能出现的任何代码缺陷而触发的 CUAF 漏洞, 在本实验研究最优变异算子序列时依然需要对这些普通变异算子的顺序进行调整.

由于本实验会对代码切片进行全排列的剪切操作, 为了初步得到有效的最优变异算子序列, 综合考虑了时间、效率等因素, 从 CVE 中选取了代码切片规模合适且特征明显的 5 个 UAF 漏洞代码切片. 充分考虑并发执行相关的资源分配、数据传递、数据同步和参数设置等因素[37,38], 依据相关研究[39]指出的并发缺陷的死锁、数据竞争、原子性违背和顺序违背这 4 个特征, 运用对并发缺陷的检测和规避技术, 构造出尽可能接近真实缺陷代码的 CUAF 漏洞代码切片.

在对 CVE 的 5 个 UAF 漏洞代码切片进行 CUAF 漏洞植入时, 加入线程声明语句以提高并发执行的并发度, 选择在访问共享变量的某些线程上加入互斥锁与信号量以适当改变代码数据竞争关系, 进而改变代码原子性执行顺序, 例如图 8 所示的以 CVE-2018-21008 代码为基础的缺陷植入切片, 通过蓝色语句的植入, 触发了基于多线程的 UAF 缺陷.

样本: CVE-2018-21008 部分代码

```
1   pthread_mutex_t lock_t;
2   struct ieee80211_hw *hw = adapter->hw;
3   void rsi_mac80211_detach()
4   {
5       enum nl80211_band band;
6       if (hw) {
7           ieee80211_stop_queues(hw);
8           ieee80211_unregister_hw(hw);
9           sem.wait();
10          ieee80211_free_hw(hw);
11      }
12      ...
13  }
14  void rsi_indicate_tx_status(struct rsi_hw *adapter,struct sk_buff *skb,int status)
15  {
16      pthread_mutex_lock(&lock_t);
17      struct ieee80211_tx_info *info = IEEE80211_SKB_CB(skb);
18      struct skb_info *tx_params;
19      ...
20  }
```

图 8 CVE-2018-21008 部分代码

首先对这些仅含有 UAF 漏洞的原始代码片段进行随机变异, 记录他们变异时变异算子执行结果, 并对所有变异样本集合进行对比核验, 找到插入 CUAF 漏洞特征后的样本在所有变异样本集合中首次出现的位置. 如算法 2 所示, 根据变异算子执行时的位置向量, 计算出在一轮变异中每个变异算子的得分. 由于本实验代码切片规模较小, 且参与权重调整的变异算子种类较少, 所以可以根据结果直接按照得分大小增减变异算子权重. 所有变异算子对应新的权重将在下一轮测试中决定其新的执行情况.

3.1.3 变异算子序列优化实验结果分析

对第 3.1.1 节中 5 个代码切片进行 20 轮调整性质的变异, 并定位目标样本的位置, 用无效样本和总样本的比值来表示每轮变异的效率, 显然在尽可能少地生成无效样本情况下变异出 CUAF 漏洞代码切片会使该比值降低, 即比值越小效率越高. 在图 9 中展示了这 20 轮的比值结果, 该迭代过程平均需要 4.8 h.

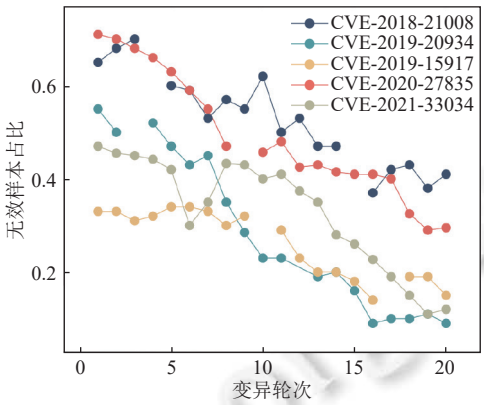

图 9 变异算子权重计算的变异效率趋势

图 9 中的横坐标为变异轮次, 纵坐标为目标样本出现时无效样本的占比. 本文在对变异算子序列进行权重计算过程与后续横向对比实验过程所涉及的无效样本占比都是指首次出现目标漏洞切片时生成的无效样本比例. 该目标样本并不是所有变异样本中唯一一个漏洞切片, 而根据模糊测试与变异测试的经验与相关基于覆盖技术的模糊测试输入优化可以认为, 当找到一个出现漏洞的样本时, 基于此变异算子执行次序的变异方法大概率会触发更多漏洞, 那么尽快地找到某一段变异样本对应的变异算子执行次序就是衡量变异效率的一个重要指标.

从图 9 中很容易观察到随着 MOPSG 算法对变异算子执行频率的改进, 变异的效率从最开始全随机执行平均提高了 33%. 根据分析, 出现的某些波动主要因为某些变异样本对应的目标漏洞代码切片含有的漏洞类型较少. 这造成一部分变异算子的执行可以直接演化出结果, 但是其他变异算子的执行会使其权重大幅降低. 比如 CVE-2021-33034 切片在进行第 7–9 轮时, 因变异算子 AM 执行权重增大而被优先执行, 而此段漏洞本身是因互斥锁语句而触发, 所以对此算子进行变异只会徒增无效样本, 并且增加后续高阶无效样本的数量. 显然, 当对多段代码进行多轮次的调整后, 会得到总体上最优的变异算子权重序列, 即最佳的一些变异算子执行序列. 每个变异轮次得到的变异算子权重情况如表 3 呈现.

表 3 变异算子权重变化

轮次	AS	AM	RJ	LTGV	LCR	ROR	SD	SPS	AOR
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	0.92	1.50	1.08	1.18	1.02	0.90	0.92	0.95	0.89
3	0.95	1.98	1.05	1.26	0.90	0.79	1.01	1.00	0.80
4	1.05	2.40	0.98	1.34	0.81	0.80	1.15	1.12	0.72
5	1.24	2.75	1.11	1.50	0.75	0.72	1.26	1.10	0.67
6	1.30	3.10	1.15	1.61	0.60	0.65	1.43	1.21	0.54
7	1.46	3.34	1.20	1.78	0.52	0.51	1.55	1.25	0.49
8	1.60	3.51	1.21	1.85	0.43	0.46	1.61	1.26	0.45
9	1.65	3.69	1.19	1.93	0.40	0.43	1.65	1.31	0.39
10	1.71	3.80	1.19	2.02	0.35	0.41	1.70	1.35	0.35

前文提到过 5 类多线程演化规律变异算子中例如 AT 算子重复执行的意义不大, 并且一部分变异算子本身就会以排列组合的方式针对所有分支进行剪裁, 因此多次执行同一变异算子反而会增加无效样本数量. 根据熟练程序员假设 (CPH) 和耦合效应假设的部分推论, 我们认为只有当实际代码规模较大时, 才需要涉及同一算子的高阶

变异体的生成, 以应对可能出现的更复杂的 CUAF 漏洞. 当这些变异算子所涉及的范围不再局限于同一漏洞模式时, 由于算子数量增加, 每轮执行后差异化并不会太大, 可能会出现变异算子的权重经过归一化后转化的概率差距不大. 这一点 MOPSG 算法是可以解决的. 在本实验阶段由于代码规模受到计算机性能的限制而被约束, 所以限制了该算法在推断演化路径时初始的算子执行次数, 当规模扩大, 对每个算子的执行次数将改动, 以适应对应的代码规模. MOPSG 算法配合随机子树执行的启发式算法, 将可以约束变异样本数量的完成基于权重的变异执行. 本实验最终运用于实际变异过程时并不会对所有变异算子的执行顺序和次数直接固定, 而是以最终的权重来对不同的变异策略的执行频率和构造的变异样本的多样性做抉择.

模糊测试中的变异算子粒度很小, 变异算子基本都是在字节级别对测试用例进行变异. 变异测试则是在代码语句级别进行修改来验证测试集的完备性, 其本质就是模拟程序员在编程过程中因某些细微错误引发的漏洞. 因此, 横向对比的是在代码语句级别进行的全随机变异过程.

3.2 变异策略实验

3.2.1 实验数据

本节数据用于比较变异算子序列在抽象语法树级别和在代码语句级别执行时的效率差异, 以及变异策略优化算法优化前后的变异数量对比. 本文选取了 GitHub 中开源的 Linux 内核中, 涉及 CUAF 同源漏洞和多线程竞争相关漏洞的错误提交报告, 从 150 多个包含重复提交的记录中筛选出 25 个用于实验, 以及 CWE 漏洞库中的 10 个 CUAF 漏洞代码片段. 本文筛选漏洞的标准是参考开源漏洞在时间线上两个版本 (即修改前后) 的漏洞语句位置跨度, 以及漏洞位置附近代码的线程复杂度. Linux 内核中大部分漏洞代码都处于多线程环境, 尤其是涉及接口函数的. 本文选择跨度不超过 1 000 行的接口漏洞进行分析, 这类漏洞具有真实危害性和广泛影响; 同时将代码片段限制在 50 行以内, 这样不容易导致变异数量过大而无法进行比较.

3.2.2 变异策略有效性实验分析

为了探究在实际代码中基于变异算子优先级调度的变异策略的有效性, 通过对 Linux 内核中关于 CUAF 漏洞的修改代码的前后两个版本进行变异. 通过在 GitHub 中搜索“concurrent use after free”和“concurrent-use-after-free”等关键字, 找到一些规模合适的, 且触发漏洞语句在代码切片中跨度不大于 50 行的代码切片共 25 个. 对这些代码切片进行分析, 它们大多数是针对驱动程序的提交, 在研究日志消息和代码修复以确认报告的漏洞是否为 CUAF 后, 筛选出 10 段代码切片作为对比实验的样本.

接着将把漏洞修复版本的代码切片转化为抽象语法树数据结构并依据上文得到的变异策略进行变异执行. 同时按照变异测试的逻辑对同样的代码切片进行语句级别的变异, 该变异过程对于变异算子的抉择是全随机的. 实时地对变异样本进行核对, 当首次出现 CUAF 漏洞的变异样本后, 统计无效样本总量, 同时设置变异样本阈值为 200 万以强行终止其核对过程, 达到该阈值视为无法有效地获取有效样本 (并非不含有有效样本, 只是没有在一定数量限制内找到目标样本, 证明此变异过程的效率已经低于可与本文变异方法比较的最低值). 后文图 10 展示了对同一代码切片进行两种变异方式的无效样本占比, 比例达到 1.00 表明阈值比例达到前没有找到目标样本.

图 10 中橙红色柱状为使用变异策略执行变异结果, 蓝色柱状为随机变异测试结果. 从图中可知, 除了在限制变异轮次内达到阈值而没有出结果的变异过程外, 基于本文变异策略执行的变异过程要比模糊变异测试中全随机的变异方式平均要少产生 34% 的无效变异样本. 虽然样本数量有限, 但是根据耦合效应假设以及相关推论, 我们认为更复杂的缺陷是在原有程序上依次执行多次单一语法修改形成的. 因此小规模的验证可以反映该算法在演化形成某一同源漏洞家族的变异样本时具有一定的优势, 能高效地生成用于训练零日漏洞检测模型的漏洞样本数据.

3.2.3 变异策略优化算法有效性实验分析

本节测试所涉及的启发式算法主要是部分子树截断算法. 由于当前针对 CUAF 漏洞模式的变异算子种类少, 且前文在变异过程中已经对该实验的固定变异算子顺序进行了测试, 所以对增加线程变异算子进行固定, 可以较容易地达到降低无效样本的产生的效果. 本部分以是否使用部分子树截断算法对样本变异结果产生的影响做对比. 其中, 用于实验的数据来自 Linux 内核中关于 CUAF 漏洞的修改代码 15 段与 CWE 中 10 个关于 CUAF 漏洞的公开代码段.

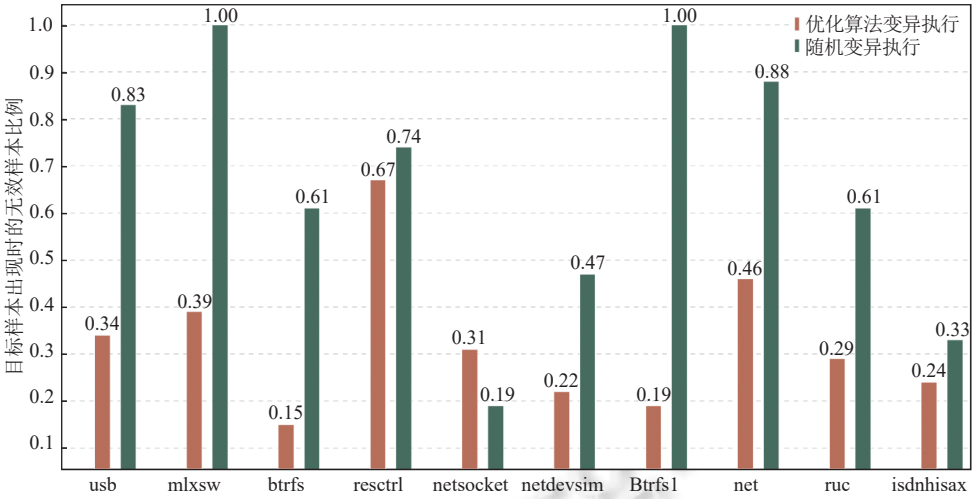

图 10　两种变异方式的无效样本占比

基准数据为这两个来源的代码直接使用算法 2 所得的权重来执行的结果, 执行结果会展示每一组代码切片在检测到对应目标漏洞样本切片首次出现时的累计样本量. 选择用样本数量来作为基准是因为不同代码切片数量差异较大, 且部分切片并不受到子树截断算法的影响. 算法 2 在得到变异算子序列的最终权重后, 依然是随机选择算子来执行, 所以即使重复执行两次也可能会得到不同的结果.

对比数据为子树截断算法介入变异执行过程后的结果. 截断算法会根据抽象语法树中的相关节点 Id 和节点值进行匹配. 一旦在变异过程中出现新的样本不含有关键词, 则会标注该文件以终止其后续更高阶的变异.

图 11 所示 CWE 中 10 个关于 CUAF 漏洞代码经过部分子树截断算法与不经过该算法直接变异的对比结果. CWE 漏洞代码切片较为简洁, 其中函数调用关系清晰且均含有 free 操作相关函数与函数调用语句. 因此, 在 CWE 代码集上进行同样的实验时, 对照数据与基准数据对比明显. 在 CWE 中 10 个关于 CUAF 漏洞代码切片中, 经过部分子树截断启发式算法的变异平均会减少 30.1% 无效样本.

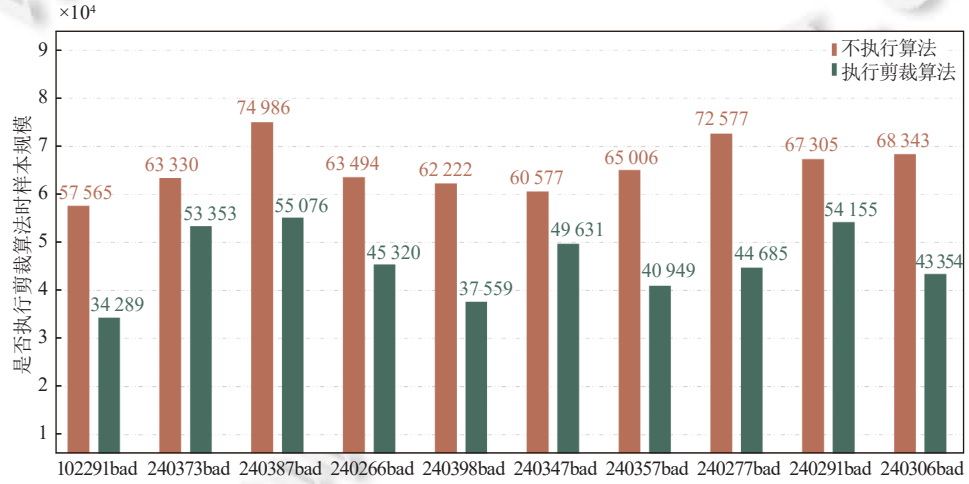

图 11　CWE 漏洞代码两种变异方式结果对比

实验的时间取决于代码片段与漏洞语句的行数之间的跨度. 因此, 排除比较时间后, 完成所有样本突变的平均时间为 3.2 h. 与此同时, 作为基线的突变方法的平均完成时间为 4.9 h (这包括了对超过阈值的代码片段强制结束

突变的时间). 即使我们从中排除了在作为基线的随机突变方法中超过时间的片段, 平均时间也超过了 4.6 h. 然而, 不能直接就执行时间来衡量这种方法与基线的效率, 因为其算法中涉及的突变操作符的优化本身就消耗了一定的时间.

表 4 展示了 Linux 内核实际代码切片经过部分子树截断算法与不经过该算法直接变异的对比结果. 由于内核实际代码跨度较长, 在做代码片段切分时, 可能会有部分对 free 操作进行调用的函数或语句被排除. 值得一提的是, 我们依然保留了变异样本阈值为 100 万以强行终止其核对过程这一机制. 我们根据每段代码切片中是否包含完整的 free 操作流程进行分类. 表中数据可以展示部分子树截断算法在针对 CUAF 漏洞演化过程中对 free 操作的截断是有明显成效的. 在包含完整的 free 操作流的切片中, 经过部分子树截断启发式算法的变异平均会减少 33.1% 无效样本. 在对缺失部分 free 操作的代码切片进行对比实验时, 是否启用部分子树截断算法对降低无效样本量的提升幅度较小, 但是平均也有 9.5% 的提升.

表 4 Linux 内核代码两种变异结果对比

Component	Contains free	No execute	Execute
usb	Contains	524 287	292 143
btrfs	Contains	211 423	136 927
btrfs1	Contains	65 535	32 767
mlxsw	Contains	23 427	21 332
netdevsim	Contains	948 575	700 786
X86resctrl	Contains	16 383	14 335
usbnet	Contains	262 143	150 174
smackfs	Contains	131 071	98 303
rpmsg	Contains	154 306	116 356
isdnhisax	Not fully contains	15 269	14 644
net	Not fully contains	32 767	34 180
netsocket	Not fully contains	15 981	14 802
ruc	Not fully contains	262 016	206 278
cw1200	Not fully contains	445 790	423 514
Ip4	Not fully contains	32 767	34 478

在目前的漏洞代码生成研究中, Nong 等人[40]提出了一种基于神经网络的代码编辑技术, 可以生成逼真的漏洞代码. 首先通过分析现有的漏洞数据集, 发现漏洞代码具有一些常见的结构模式, 例如条件语句、循环语句等. 作者使用神经网络对这些结构模式进行建模, 并提出了一种代码编辑器, 可以在保留代码语法正确性的同时, 改变代码的结构和行为, 但是生成新的漏洞代码数量较少, 且只关注了特定的漏洞类型. Nong 等人[41]还提出了一个新的漏洞生成方法, 该方法可以利用现有的漏洞数据库来学习漏洞的模式, 并生成与这些模式相似的漏洞. 该方法使用了深度学习模型来对漏洞模式进行建模, 并使用生成模型来生成新的漏洞, 这些漏洞具有与真实漏洞相似的特征和语法结构. 但是该方法依赖于已知的漏洞数据库, 因此对于那些没有被记录在数据库中的新型漏洞, 该方法可能无法生成相应的漏洞.

通过上述实验数据对比分析, 在进行小规模的实际代码进行变异后, 本文提出基于抽象语法树变异的漏洞样本生成技术, 非常有效地改善了变异算子特征不明显、执行随机性过强和变异样本高耦合性等问题, 且不过分依赖漏洞数据库. 本文对根据 CUAF 演化规律总结的变异算子和变异测试中针对多线程代码的变异算子进行了处理. 通过获取的变异算子权重执行各个算子, 在 Linux 内核的多段针对 CUAF 修改的代码上进行变异执行都取得了不错的拟合效果. 相较于变异测试中对代码的随机变异过程, 本方法更容易构造出接近真实漏洞的变异样本, 并且相对代价更低.

3.2.4 不足与有效性讨论

下面讨论本文方法在 3 个方面有效性上的可改进点.

外部有效性主要涉及研究发现的普遍性问题. 本研究集中于 Linux 内核代码的静态模糊变异和相关的公开漏洞数据集. 但不同的编程语言有其特有的漏洞家族和演变模式, 因此本研究的结论可能不适用于其他编程语言,

如 Go 语言在多线程漏洞触发上与 C/C++存在差异.

内部有效性与代码测试静态模糊变异相关. 通过分析 Linux 贡献者的提交记录, 研究涉及多个函数间的代码接口. 然而, 正则匹配和人工观察在判断变异样本的有效性时确实存在一定的误差风险. 正则匹配可能无法覆盖所有漏洞特征, 而人工观察则可能受到主观判断的影响. 为了减少这些误差, 我们在实验中引入了多重验证机制, 包括交叉验证和重复实验, 以确保结果的可靠性. 尽管如此, 我们承认在实际应用中, 仍可能存在少量误报和漏报的情况.

结构有效性涉及验证方法和确定的优化变异操作符序列. 尽管尝试在 Linux 代码的变异验证中再次引入算法来优化变异操作符集合, 但由于真实漏洞代码片段的大小限制, 可能存在过度拟合的风险.

4 总 结

本文主要研究了通过抽象语法树变异来生成相关漏洞变异样本的方法. 首先, 对 4 种抽象语法树生成技术进行了研究和分析, 发现它们的生成方式和结果不便于静态分析和变异操作. 然后, 重点研究了 Eclipse CDT 抽象语法树生成技术, 并提出了一种针对其节点存储结构的优化算法, 可以提高变异效率. 本文根据 UAF 漏洞和 CUAF 漏洞之间的演化关系, 设计了 5 种符合漏洞特征的变异算子, 并将其与模糊测试中的 5 种普通变异算子组成变异算子序列, 分别制定了不同的语法树变异方式, 并对每一种方式进行了算法实现. 本文提出了一种基于变异算子关联度引导的次序变异方法 MOPSG, 通过调整变异算子执行的顺序和次数, 得到一个最优的变异算子权重序列来执行变异操作. 通过实验验证了该算法的有效性, 并与传统的模糊变异测试方法进行对比测试, 结果表明通过优化后的变异算子序列来进行变异, 得到的包含漏洞特征有效变异体样本效率更高. 最后, 本文使用部分子树截断算法对基于 MOPSG 算法的变异策略进行了优化, 提高了变异效率.

References:

[1] Liu J, Su PR, Yang M, He L, Zhang Y, Zhu XY, Lin HM. Software and cyber security—A survey. Ruan Jian Xue Bao/Journal of Software, 2018, 29(1): 42–68 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5320.htm [doi: 10.13328/j.cnki.jos.005320]

[2] You W, Wang XQ, Ma SQ, Huang JJ, Zhang XY, Wang XF, Liang B. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In: Proc. of the 2019 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2019. 769–786. [doi: 10.1109/SP.2019.00057]

[3] Wen C, Wang HJ, Li YK, Qin SC, Liu Y, Xu ZW, Chen HX, Xie XF, Pu GG, Liu T. MemLock: Memory usage guided fuzzing. In: Proc. of the 42nd Int'l Conf. on Software Engineering. Seoul: IEEE, 2020. 765–777. [doi: 10.1145/3377811.3380396]

[4] Wang HJ, Xie XF, Li Y, Wen C, Li YK, Liu Y, Qin SC, Chen HX, Sui YL. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In: Proc. of the 42nd Int'l Conf. on Software Engineering. Seoul South: ACM, 2020. 999–1010. [doi: 10.1145/3377811.3380386]

[5] Gan ST, Zhang C, Qin XJ, Tu XW, Li K, Pei ZY, Chen ZN. CollAFL: Path sensitive fuzzing. In: Proc. of the 2018 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2018. 679–696. [doi: 10.1109/SP.2018.00040]

[6] Li YK, Xue YX, Chen HX, Wu XH, Zhang C, Xie XF, Wang HJ, Liu Y. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 533–544. [doi: 10.1145/3338906.3338975]

[7] Liang HL, Wang L, Wu DY, Xu JY. MLSA: A static bugs analysis tool based on LLVM IR. In: Proc. of the 17th IEEE/ACIS Int'l Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). Shanghai: IEEE, 2016. 407–412. [doi: 10.1109/SNPD.2016.7515932]

[8] Cassez F, Sloane AM, Roberts M, Pigram M, Suvanpong P, de Aledo PG. Skink: Static analysis of programs in LLVM intermediate representation: (Competition Contribution). In: Proc. of the 23rd Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Uppsala: Springer, 2017. 380–384. [doi: 10.1007/978-3-662-54580-5_27]

[9] Thomé J, Shar LK, Bianculli D, Briand L. Search-driven string constraint solving for vulnerability detection. In: Proc. of the 39th Int'l Conf. on Software Engineering (ICSE). Buenos Aires: IEEE, 2017. 198–208. [doi: 10.1109/ICSE.2017.26]

[10] Shen WJ, Tang EY, Chen ZY, Chen X, Li B, Zhai J. Method for automated detection of suspicious vulnerability related to numerical stability. Ruan Jian Xue Bao/Journal of Software, 2018, 29(5): 1230–1243 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5503.htm [doi: 10.13328/j.cnki.jos.005503]

[11] Chen QY, Li SP, Yan M, Xia X. Code clone detection: A literature review. Ruan Jian Xue Bao/Journal of Software, 2019, 30(4): 962–980 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5711.htm [doi: 10.13328/j.cnki.jos.005711]

[12] Li JY, Ernst MD. CBCD: Cloned buggy code detector. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). Zurich: IEEE, 2012. 310–320. [doi: 10.1109/ICSE.2012.6227183]

[13] Neuhaus S, Zimmermann T, Holler C, Zeller A. Predicting vulnerable software components. In: Proc. of the 14th ACM Conf. on Computer and Communications Security. Alexandria: ACM, 2007. 529–540. [doi: 10.1145/1315245.1315311]

[14] Jang J, Agrawal A, Brumley D. ReDeBug: Finding unpatched code clones in entire OS distributions. In: Proc. of the 2012 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2012. 48–62. [doi: 10.1109/SP.2012.13]

[15] Li HZ, Kwon H, Kwon J, Lee H. A scalable approach for vulnerability discovery based on security patches. In: Proc. of the 2014 Int'l Conf. on Applications and Techniques in Information Security. Melbourne: Springer, 2014. 109–122. [doi: 10.1007/978-3-662-45670-5_11]

[16] Yamaguchi F, Lottmann M, Rieck K. Generalized vulnerability extrapolation using abstract syntax trees. In: Proc. of the 28th Annual Computer Security Applications Conf. Orlando: ACM, 2012. 359–368. [doi: 10.1145/2420950.2421003]

[17] Pham NH, Nguyen TT, Nguyen HA, Nguyen TN. Detection of recurring software vulnerabilities. In: Proc. of the 25th IEEE/ACM Int'l Conf. on Automated Software Engineering. Antwerp: ACM, 2010. 447–456. [doi: 10.1145/1858996.1859089]

[18] 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]

[19] Wang L, Li F, Li L, Feng XB. Principle and practice of taint analysis. Ruan Jian Xue Bao/Journal of Software, 2017, 28(4): 860–882 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5190.htm [doi: 10.13328/j.cnki.jos.005190]

[20] Liang B, Hou KK, Shi WC, Liang ZH. A static vulnerabilities detection method based on security state tracing and checking. Chinese Journal of Computers, 2009, 32(5): 899–909. (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2009.00899]

[21] Viega J, Bloch JT, Kohno Y, McGraw G. ITS4: A static vulnerability scanner for C and C++ code. In: Proc. of the 16th Annual Computer Security Applications Conf. (ACSAC 2000). New Orleans: IEEE, 2000. 257–267. [doi: 10.1109/ACSAC.2000.898880]

[22] Walden J, Doyle M. SAVI: Static-analysis vulnerability indicator. IEEE Security & Privacy, 2012, 10(3): 32–39. [doi: 10.1109/MSP.2012.1]

[23] Baloglu B. How to find and fix software vulnerabilities with coverity static analysis. In: Proc. of the 2016 IEEE Cybersecurity Development (SecDev). Boston: IEEE, 2016. 153. [doi: 10.1109/SecDev.2016.041]

[24] Cheng X, Wang HY, Hua JY, Zhang M, Xu GA, Yi L, Sui YL. Static detection of control-flow-related vulnerabilities using graph embedding. In: Proc. of the 24th Int'l Conf. on Engineering of Complex Computer Systems (ICECCS). Guangzhou: IEEE, 2019. 41–50. [doi: 10.1109/ICECCS.2019.00012]

[25] Zheng W, Gao JL, Wu XX, Liu FY, Xun YX, Liu GL, Chen X. The impact factors on the performance of machine learning-based vulnerability detection: A comparative study. Journal of Systems and Software, 2020, 168: 110659. [doi: 10.1016/j.jss.2020.110659]

[26] Li Z, Zou DQ, Xu SH, Jin H, Zhu YW, Chen ZX. SySeVR: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. on Dependable and Secure Computing, 2022, 19(4): 2244–2258. [doi: 10.1109/TDSC.2021.3051525]

[27] Gao J, Yang X, Fu Y, Jiang Y, Sun JG. VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary. In: Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering. Montpellier: IEEE, 2018. 896–899. [doi: 10.1145/3238147.3240480]

[28] Zhou YQ, Liu SQ, Siow JK, Du XN, Liu Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proc. of the 33rd Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2019. 10197–10207.

[29] Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M. Automated vulnerability detection in source code using deep representation learning. In: Proc. of the 17th IEEE Int'l Conf. on Machine Learning and Applications (ICMLA). Orlando: IEEE, 2018. 757–762. [doi: 10.1109/ICMLA.2018.00120]

[30] Li Z, Zou DQ, Xu SH, Ou XY, Jin H, Wang SJ, Deng ZJ, Zhong YY. VulDeePecker: A deep learning-based system for vulnerability detection. arXiv:1801.01681, 2018.

[31] Chakraborty S, Krishna R, Ding YRB, Ray B. Deep learning based vulnerability detection: Are we there yet. IEEE Trans. on Software Engineering, 2022, 48(9): 3280–3296. [doi: 10.1109/TSE.2021.3087402]

[32] Beaman C, Redbourne M, Mummery JD, Hakak S. Fuzzing vulnerability discovery techniques: Survey, challenges and future directions. Computers & Security, 2022, 120: 102813. [doi: 10.1016/j.cose.2022.102813]

[33] Garg A, Ojdanic M, Degiovanni R, Chekam TT, Papadakis M, Le Traon Y. Cerebro: Static subsuming mutant selection. IEEE Trans. on Software Engineering, 2023, 49(1): 24–43. [doi: 10.1109/TSE.2022.3140510]

[34] Ma YS, Offutt J, Kwon YR. MuJava: An automated class mutation system. Software Testing, Verification and Reliability, 2005, 15(2):

97–133. [doi: 10.1002/stvr.308]
[35] Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Trans. on Software Engineering, 2011, 37(5): 649–678. [doi: 10.1109/TSE.2010.62]
[36] Khanfir A, Degiovanni R, Papadakis M, Le Traon Y. Efficient mutation testing via pre-trained language models. arXiv:2301.03543, 2023.
[37] Offutt AJ, Voas J, Payne J. Mutation operators for Ada. Technical Report, ISSE-TR-96-09. Fairfax: George Mason University, 1996.
[38] Tian T, Gong DW. Survey on mutation testing of concurrent programs. Acta Electronica Sinica, 2020, 48(11): 2267–2277. (in Chinese with English abstract). [doi: 10.3969/j.issn.0372-2112.2020.11.025]
[39] Su XH, Yu Z, Wang TT, Ma PJ. A survey on exposing, detecting and avoiding concurrency bugs. Chinese Journal of Computers, 2015, 38(11): 2215–2233. (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2015.02215]
[40] Nong Y, Ou YZ, Pradel M, Chen F, Cai HP. Generating realistic vulnerabilities via neural code editing: An empirical study. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Singapore: ACM, 2022. 1097–1109. [doi: 10.1145/3540250.3549128]
[41] Nong Y, Ou YZ, Pradel M, Chen F, Cai HP. VULGEN: Realistic vulnerability generation via pattern mining and deep learning. In: Proc. of the 45th Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 2527–2539. [doi: 10.1109/ICSE48619.2023.00211]

附中文参考文献:

[1] 刘剑, 苏璞睿, 杨珉, 和亮, 张源, 朱雪阳, 林惠民. 软件与网络安全研究综述. 软件学报, 2018, 29(1): 42–68. http://www.jos.org.cn/1000-9825/5320.htm [doi: 10.13328/j.cnki.jos.005320]
[10] 沈维军, 汤恩义, 陈振宇, 陈鑫, 李彬, 翟娟. 数值稳定性相关漏洞隐患的自动化检测方法. 软件学报, 2018, 29(5): 1230–1243. http://www.jos.org.cn/1000-9825/5503.htm [doi: 10.13328/j.cnki.jos.005503]
[11] 陈秋远, 李善平, 鄢萌, 夏鑫. 代码克隆检测研究进展. 软件学报, 2019, 30(4): 962–980. http://www.jos.org.cn/1000-9825/5711.htm [doi: 10.13328/j.cnki.jos.005711]
[19] 王蕾, 李丰, 李炼, 冯晓兵. 污点分析技术的原理和实践应用. 软件学报, 2017, 28(4): 860–882. http://www.jos.org.cn/1000-9825/5190.htm [doi: 10.13328/j.cnki.jos.005190]
[20] 梁彬, 侯看看, 石文昌, 梁朝晖. 一种基于安全状态跟踪检查的漏洞静态检测方法. 计算机学报, 2009, 32(5): 899–909. [doi: 10.3724/SP.J.1016.2009.00899]
[38] 田甜, 巩敦卫. 并发程序变异测试研究综述. 电子学报, 2020, 48(11): 2267–2277. [doi: 10.3969/j.issn.0372-2112.2020.11.025]
[39] 苏小红, 禹振, 王甜甜, 马培军. 并发缺陷暴露、检测与规避研究综述. 计算机学报, 2015, 38(11): 2215–2233. [doi: 10.11897/SP.J.1016.2015.02215]

郑炜(1975－), 男, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为软件测试, 软件安全, 适航验证.

吴潇雪(1983－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为智能化软件测试, 软件漏洞分析.

李云帆(2000－), 男, 硕士生, 主要研究领域为软件质量保证, 形式化验证.

陈翔(1980－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为智能软件工程, 软件仓库挖掘, 经验软件工程.

桂奎(1997－), 男, 硕士, 主要研究领域为软件工程, 软件测试.

邓沛然(1998－), 男, 硕士生, 主要研究领域为智能化软件测试, 软件分析.