软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(1):136−158 [doi: 10.13328/j.cnki.jos.006828]

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

漏洞自动修复研究综述*

徐同同, 刘 逵, 夏 鑫

(华为公司 软件工程应用技术实验室, 浙江 杭州 310007)

通信作者: 刘逵, Email: liukui0811@163.com

摘 要: 软件漏洞是计算机软件系统安全方面的缺陷, 给现代软件及其应用数据的完整性、安全性和可靠性带来巨大威胁. 人工治理漏洞费时且易错, 为了更好应对漏洞治理挑战, 研究者提出多种自动化漏洞治理方案, 其中漏洞自动修复方法近来得到研究者广泛关注. 漏洞自动修复技术旨在辅助开发人员修复漏洞, 涵盖漏洞根因定位、补丁生成、补丁验证等功能. 现有工作缺乏对漏洞修复技术系统性的分类与讨论, 为了促进漏洞修复技术发展, 加深研究人员对漏洞修复问题的认知理解, 对现有漏洞修复方法技术的理论、实践、适用场景和优缺点进行全面洞察, 并撰写了漏洞自动修复技术的研究综述. 主要内容包括: (1) 按照修复漏洞类型不同整理归纳特定类型漏洞的修复方法以及通用类型漏洞的修复方法; (2) 按照所采用的技术原理将不同修复方法进行分类与总结; (3) 归纳漏洞修复主要挑战; (4) 展望漏洞修复未来发展方向.

关键词: 软件漏洞; 漏洞自动修复; 漏洞分析; 漏洞定位

中图法分类号: TP311

中文引用格式: 徐同同, 刘逵, 夏鑫. 漏洞自动修复研究综述. 软件学报, 2024, 35(1): 136–158. http://www.jos.org.cn/1000-9825/6828.htm

英文引用格式: Xu TT, Liu K, Xia X. Survey on Automated Vulnerability Repair. Ruan Jian Xue Bao/Journal of Software, 2024, 35(1): 136–158 (in Chinese). http://www.jos.org.cn/1000-9825/6828.htm

Survey on Automated Vulnerability Repair

XU Tong-Tong, LIU Kui, XIA Xin

(Software Engineering Application Technology Lab, Huawei Technologies Co. Ltd., Hangzhou 310007, China)

Abstract: Software vulnerabilities are known as security defects of computer software systems, and they threaten the completeness, security, and reliability of modern software and application data. Artificial vulnerability management is time-consuming and error-prone. Therefore, in order to better deal with the challenges of vulnerability management, researchers have proposed a variety of automated vulnerability management schemes, among which automated vulnerability repair has attracted wide attention from researchers recently. Automated vulnerability repair consists of three main functions: vulnerability cause localization, patch generation, and patch validation, and it aims to assist developers to repair vulnerabilities. The existing work lacks systematic classification and discussion of vulnerability repair technology. To this end, this study gives a comprehensive insight into the theory, practice, applicable scenarios, advantages, and disadvantages of existing vulnerability repair methods and technologies and writes a research review of automated vulnerability repair technologies, so as to promote the development of vulnerability repair technologies and deepen researchers' cognition and understanding of vulnerability repair problems. The main contents of the study include: (1) sorting out and summarizing the repair methods of specific and general vulnerabilities according to different vulnerability types; (2) classifying and summarizing different repair methods based on technical principles; (3) summarizing the main challenges of vulnerability repair; (4) looking into future development direction of vulnerability repair.

Key words: software vulnerability; automated vulnerability repair; vulnerability analysis; vulnerability localization

---

* 基金项目: 国家自然科学基金 (62172214); 江苏省自然科学基金 (BK20210279); 数学工程与先进计算国家重点实验室开放基金 (2020A06)
收稿时间: 2022-01-17; 修改时间: 2022-05-15, 2022-07-12, 2022-09-26; 采用时间: 2022-11-11; jos 在线出版时间: 2023-06-14
CNKI 网络首发时间: 2023-06-15

1 引 言

软件漏洞 (software vulnerablity, 本文简称为漏洞) 是指计算机系统安全方面的缺陷, 恶意攻击者可以利用漏洞对计算机系统进行攻击, 造成拒绝服务、程序崩溃乃至远程控制宿主机等严重安全事故. 据漏洞公开数据库CVE[1]所示, 自 2016 年以来, CVE 中公开漏洞数量从 6454 急剧增长到了 19351 (截至 2021 年 12 月 21 日), 增长率为 200%. 令人遗憾的是, 即便开发者和研究人员对于漏洞发生机制的认识不断深入, 一些机理简单、但存在广泛、危害性极大的漏洞仍然遍布在各类软件以及开源社区中. 以 2021 年 12 月 10 日公布的 Log4j2 高危漏洞 CVE-2021-44228 为例, 其发生的根因在于 Log4j2 为日志记录提供的 Lookup 功能, 而该功能允许用户以${}的格式在日志中灵活记录某些特殊的值. 具体而言, Lookup 一旦发现形如${}的用户输入, 则会触发字符串解析机制, 在进行解析的过程中, Lookup 支持多种解析协议, 例如 date, Java, ctx 以及 JNDI[2]等. 而 JNDI 中的轻量目录访问协议 (lightweight directory access protocol, LDAP) 支持远程访问, 恶意攻击者可以基于此构造远程访问命令以达到远程控制目标机器的目的. 该漏洞的触发机理相当直观, 但是这样直观的漏洞仍然出现在阿帕奇资助的、维护时间超过 18 年且被大量程序集成的开源代码中, 对后期的漏洞治理带来很大的挑战.

如上所述, 漏洞作为一种危害性极大的软件安全缺陷, 给开发者和用户带来巨大困扰与挑战. 研究者近年来提出各种漏洞治理技术, 主要是漏洞防护 (defense) 和漏洞修复. 为了方便介绍漏洞防护与漏洞修复概念, 本文以 Log4j2 的高危漏洞 CVE-2021-44228 的攻防流程为例进行阐述. 如图 1 所示, 攻击者为了成功获取漏洞服务器 (即存在漏洞问题的服务器) 的远程控制权限, 需要经历以下 5 个步骤.

图 1 漏洞 CVE-2021-44228 攻防流程

步骤 1. 攻击者构造含特殊字符串${jndi:ldap://attact.xa}的 GET 命令, 作为漏洞服务器访问的输入.

步骤 2. 漏洞服务器执行过程调用 Log4j2 应用, 将特殊字符串作为 Log4j2 应用的输入.

步骤 3. Log4j2 程序调用 Lookup 方法将特殊字符串识别为攻击者预设 LDAP 地址并访问攻击者服务器.

步骤 4. 攻击者服务器基于 LDAP 协议将被注入恶意代码的目录信息返回给漏洞服务器.

步骤 5. 漏洞服务器加载并执行相应恶意代码.

为了应对漏洞的恶意攻击, 一种漏洞治理策略便是在漏洞攻击的各个步骤进行漏洞防护 (图 1 红色字体标注部分), 例如图 1 所示的防火墙防护、漏洞软件禁用、漏洞服务禁用以及敏感功能禁用等. 另一种治理策略则是修复代码中的漏洞问题 (图 1 绿色字体标注部分), 即对软件漏洞进行根因分析、定位与修复. 当前, 漏洞防护相关的技术已趋于成熟, 漏洞修复仍处于探索研究阶段.

传统人工治理漏洞的方式存在很大的挑战: 一方面, 人工修复漏洞费时且易错. 以漏洞 CVE-2021-44228 为例, Log4j2 维护者 2021 年 12 月 10 日发布 2.15 版本仅为该漏洞的临时修复版本, 且该版本引入其他漏洞; 截至 2021 年 12 月 29 日, Log4j2 维护者针对修复不完善问题迭代发布 2.16、2.17、2.17.1 等多个修复版本. 现有研究显示, 即使对于公开发布漏洞, 开发者人工修复漏洞平均需要 28 天[3]. 另一方面, 人工修复漏洞困难且易引入新漏洞. 随着开源软件生态不断壮大, 越来越多项目使用大量开源软件例如 Log4j2, 并进行定制化二次开发. 对于该类场景,

即便开源软件发布官方修复补丁, 人工修复本地项目可能影响正常业务逻辑, 且可能引入新的漏洞[4], 此外官方修复补丁的正确性仍有待考量. 为了促进自动化漏洞治理的研究与发展, 漏洞自动修复技术近年来得到广泛关注.

漏洞自动修复遵循漏洞定位、补丁生成、补丁验证的工作流程, 涵盖诸多先进软件开发技术, 如程序动/静态分析、软件测试、机器学习等. 不同于一般意义上的软件缺陷, 软件漏洞存在极大破坏软件安全性的隐患, 因此, 安全研究人员为了更好地理解不同漏洞的发生根因与修复机制, 将软件漏洞根据其内在特征进行分类, 形成各个特定漏洞类型. 根据是否针对特定类型漏洞, 现有的漏洞自动修复方法可分为特定类型漏洞的修复方法和通用类型漏洞的修复方法. 特定类型漏洞的修复方法根据特定类型漏洞发生及修复机制制定专有的漏洞定位与补丁生成算法, 通用类型的漏洞修复方法不针对特定类型漏洞, 而是根据漏洞修复的历史数据基于学习算法训练漏洞修复模型以修复任意类型漏洞. 另一方面, 根据研究者们从漏洞定位、补丁生成、补丁验证与程序分析等方面探索的方法, 现有漏洞自动修复方法从所采用的技术原理看可分为基于静态分析的漏洞修复、基于动态分析的漏洞修复、基于历史修复的漏洞修复和混合式漏洞修复方法. 整体而言, 过去 10 年间, 漏洞自动修复研究工作取得了一定进展, 但仍然存在不足之处, 很多问题仍待解决. 对于特定类型的漏洞修复方法, 自动修复技术可以高效地自动生成各种候选补丁, 同时高效验证候选补丁可行性, 相较于人工修复方式, 可以提升漏洞修复效率, 降低补丁出错的概率. 然而, 当前研究局限于对几种经典漏洞类型的自动修复, 考虑到已知漏洞类型已经超过 900 种[5], 研究者们对于绝大多数特定类型漏洞的发生根因与修复机制依然缺乏理解, 因此难以展开进一步的特定类型漏洞修复技术研究. 通用类型漏洞的修复方法依赖于高质量的漏洞修复数据集, 然而当前最大规模的漏洞修复数据集[6]亦仅包含 3 754 对漏洞程序/修复程序, 无法满足学习技术对于数据多样性和丰富性的要求.

综上所述, 漏洞自动修复技术虽然有改善漏洞治理水平的潜力, 然而该类技术当前依然存在各类挑战与技术瓶颈, 其仍是学术界和工业界需明确研究和探索的课题. 为了促进漏洞治理, 尤其是漏洞修复技术的研究与发展, 加深研究人员对漏洞修复问题的认知, 同时考虑到现有工作缺乏对漏洞修复技术系统性的分类与讨论, 本文作者对现有漏洞修复方法技术涉及到相关文献的理论、实践、适用场景和优缺点进行全面的洞察, 并撰写了漏洞自动修复技术的研究综述.

文献选取方式: 本文采用以下流程完成对文献的索引与选取: 本综述在公开的期刊及会议论文、出版书籍中, 检索在漏洞修复方法研究中提出的新技术, 或为漏洞修复方法提供实证研究支持的文献. 本文根据上述原则按以下 3 步方法在文献库中检索和选取相关研究文献.

(1) 本综述选用 ACM 电子文献数据库、IEEE Xplore 电子文献数据库、Springer Link 电子文献数据库以及 Google 学术搜索引擎等进行原始搜索. 论文检索的关键字包括 vulnerablity、repair、fix、patch 等. 同时, 在标题、摘要、关键词和索引中进行检索.

(2) 本综述依据中国计算机学会 (CCF) 对软件工程与程序语言、网络空间安全以及人工智能领域推荐的国际学术会议和期刊列表进行文献检索, 例如 TOSEM、TSE、EMSE、JSS、ICSE、FSE/ESEC、ASE、MSR、USENIX、CCS、NIPS 等, 搜索时间从 1990 年开始.

(3) 为避免遗漏相关研究, 在之前两步搜索的基础上, 根据每篇文献的参考文献列表寻找与漏洞修复问题相关的研究文献, 将遗漏的相关文献添加到本文的研究文献中.

根据上述选取原则和检索步骤, 本文最终收集选取了 52 篇文献作为综述总结的相关文献. 在这些文献中, 提出了漏洞修复方法的新理论、新算法的直接相关文献有 24 篇; 其他为漏洞修复的背景动机、评估方法提供实证研究支持的部分相关文献有 28 篇. 上述文献分布情况如图 2 所示, 发表过相关研究较多的期刊与会议有: TSE (4 篇), ICSE (6 篇), FSE (4 篇), TOSEM (2 篇), USENIX (4 篇) 和 S&P (3 篇). 此外如图 3 所示, 从总体趋势来看, 关于漏洞修复的研究工作从 20 世纪末就已逐渐展开, 每年研究成果呈上升趋势, 研究成果主要集中在近 10 年, 这与信息技术的发展息息相关, 特别是先进软件开发技术的研究与发展. 从期刊和会议主题来看, 漏洞修复方法的研究主要集中在软件工程领域与安全领域.

本文首先介绍已有漏洞修复相关的综述工作. Ye 等人[7]探究发现基于不同静态分析器扫描缓冲区溢出漏洞假阳性过高, 此外该工作挖掘了现有针对缓冲区溢出漏洞的修复模式. Marchand-Melsom 等人[8]关注于 OWASP[9]

前 10 名软件漏洞类型, 并发现现有漏洞自动修复方法无法很好地解决 OWASP 安全漏洞的修复问题. Kechagia 等人[10]探究测试驱动缺陷修复技术用于修复 API 误用错误的效果. Canfora 等人[11]探究了人工漏洞修复过程中所引入的代码变更模式. 整体而言, 目前尚无相关工作对漏洞修复的研究和应用进展进行系统性梳理, 本文作者基于对现有工作的系统性分析试图从两个不同的维度对漏洞修复的研究工作进行归纳与总结. 一方面, 现有研究工作通过限定其修复的漏洞类型归纳相应的漏洞发生及修复机制, 进而设计并实现专有的漏洞定位与补丁生成算法, 因此本文首先基于不同工作所关注的漏洞类型归纳并阐述现有的研究进展; 另一方面, 即便是针对不同类型漏洞所设计的自动修复方法, 其所采用方法的技术原理可能是相似的, 按照技术原理系统性地分类与总结现有研究工作有助于本文读者更深入地理解现有漏洞修复方法的技术全景, 因此本文紧接着基于不同修复方法所采用的技术原理分类并阐述现有工作. 简而言之, 本文从特定类型漏洞的修复方法以及漏洞修复方法的技术原理分类两个维度对现有研究工作进行阐述. 本文第 2 节介绍漏洞自动修复方法的相关概念与研究概况. 第 3 节介绍特定类型漏洞的修复方法. 第 4 节介绍漏洞修复方法的技术原理分类. 第 5 节、第 6 节分别介绍当前漏洞修复方法的主要挑战、未来研究方向以及对全文进行总结.

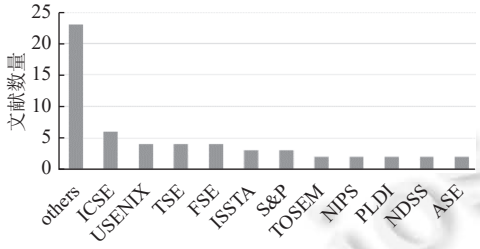

图 2 漏洞自动修复文献分布

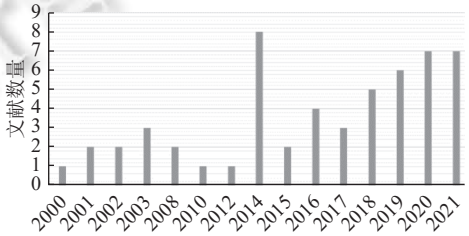

图 3 文献发表年份统计

2 漏洞修复相关概念与研究概况

2.1 漏洞修复相关概念

为了便于本综述的阐述和阅读, 本节将漏洞修复相关概念做如下陈述.

◆ 漏洞是指计算机信息系统安全方面的缺陷[12], 使得系统或其应用数据的安全性、完整性、可靠性等面临威胁. 程序代码错误是导致漏洞的主要原因, 但并不是所有的漏洞都是程序错误导致的. 本文所研究的漏洞为软件程序代码错误导致的漏洞.

◆ 程序错误 (fault)/缺陷 (defect) 是程序设计中的术语, 指在软件运行中因为程序本身有错误而造成的功能不正常、死机、数据丢失、非正常中断等现象. 有一类程序错误/缺陷会造成计算机安全隐患, 此类软件错误/缺陷被称为软件漏洞. 因此, 软件漏洞是一种特殊类型的程序错误/缺陷.

◆ 程序自动修复技术[13,14]是对软件程序缺陷进行缺陷定位、修复方案生成 (修改缺陷程序) 和修复方案验证 (测试验证修改后的缺陷程序) 一体化、自动化的方法和技术.

◆ 漏洞自动修复从程序自动修复衍生而来, 指漏洞检测/定位、修复方案生成 (修改漏洞程序) 和修复方案验证 (测试修改后的漏洞程序) 的一体化、自动化的漏洞修复过程. 需要注意的是, 本文研究的漏洞修复是指对软件漏洞进行根因分析和修复的过程, 而另一类基于漏洞防护、阻断漏洞传播链 (gadget chain) 以防止漏洞发生的工作[15]以及修复迁移的工作[16]不在本文综述研究范围之内.

◆ 漏洞利用 (exploit) 是指能够暴露程序漏洞行为的程序输入. 本文综述主要关注在给定漏洞利用的条件下, 如何自动修复漏洞程序, 然而漏洞利用自动生成技术[17]是研究漏洞检测、该漏洞发生构造有效输入的工作, 故不在本文综述范围内.

◆ CVE (common vulnerabilities and exposures)[1]作为目前权威漏洞公开数据库, 旨在收集各种信息安全弱点及漏洞并给予编号以便于公众查阅. 此数据库现由美国非营利组织 MITRE 所属的 National Cybersecurity FFRDC

营运维护. 例如 CVE-2021-44228、CVE 2021-45046 是最近发现的 Log4j2 严重漏洞维护在此数据库中的对应编号.

◆ CWE (common weakness enumeration)[5]是一个针对 CVE 的分类系统. 目前 CWE 系统中收录的漏洞类型已超过 900 类, 例如数组读写溢出 CWE-787、CWE-125. 本文参考 CWE 分类系统中漏洞分类梳理特定类型漏洞的修复技术.

2.2 漏洞自动修复的流程框架

漏洞自动修复任务的主要流程框架包括 3 大模块: 漏洞定位模块、补丁生成模块和补丁验证模块, 如图 4 所示. 对于给定的存在漏洞的程序, 漏洞自动修复技术根据既定的漏洞定位技术 (如动态执行的漏洞利用、静态分析工具或历史漏洞修复数据等) 鉴别出漏洞代码位置; 然后利用特有的补丁生成技术 (如基于启发式、历史修复、修复模板等) 自动地修改漏洞代码并生成对应的补丁; 最后, 生成的补丁将在补丁验证模块利用动态分析、静态分析等技术验证补丁的正确性、可行性.

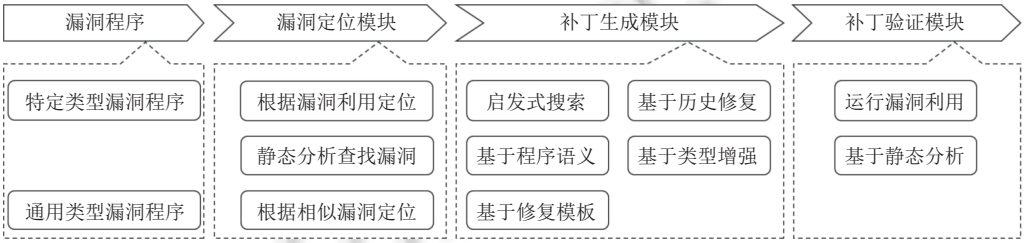

图 4 漏洞修复基本流程

● 漏洞定位模块旨在识别导致漏洞发生的根因代码位置以及补丁生成位置, 漏洞定位阶段输出的一组可疑程序节点构成为补丁生成模块的输入信息. 现有的漏洞定位技术根据其鉴别漏洞的核心技术可以分为 3 类: (1) 基于漏洞利用的定位, 该类技术[18,19]将漏洞利用收集各个程序节点上的动态执行信息与规避特定类型漏洞所需满足的约束进行匹配, 输出一组不满足安全约束的程序节点; (2) 基于静态分析的定位, 该定位方式设计检测特定类型漏洞的静态分析算法, 输出一组可能引发漏洞的静态警报[20,21]; (3) 基于代码相似度的定位, 该类定位方式[22]是将漏洞程序的代码与历史漏洞数据集中漏洞代码进行相似性匹配, 从符号 (token) 相似度、语法语义相似度、上下文相似度等各个角度评价漏洞程序各个程序节点与历史漏洞数据集中漏洞程序的相似程度, 按照相似度大小输出一组可疑程序节点.

● 补丁生成模块专注于遍历可疑程序节点, 修改漏洞代码并生成补丁代码以消除漏洞发生根因. 研究者已提出了多种补丁生成方法. 例如, 启发式搜索方法[23,24]将补丁生成问题规约为搜索问题, 通过设计各类启发式规则更好地生成补丁代码; 基于历史修复[6,25−27]的方法通过收集、过滤、聚类并抽象已有的漏洞修复代码, 对当前漏洞程序进行相似性匹配与代码转换; 基于程序语义的方法[18,19]抽取漏洞程序应满足的安全约束, 通过求解、合成满足约束条件的补丁代码进行漏洞修复; 基于类型增强[28,29]的方法针对特定编程语言, 使用安全语言类型替换非安全语言类型以实现漏洞修复; 此外, 基于修复模板的方法首先配置一组人工给定或自动生成的补丁模板, 通过在模板中填充修复组件 (方法名、API 调用、表达式等) 生成修复补丁.

● 补丁验证模块是验证补丁能否成功消除漏洞根因, 以达到漏洞修复技术修复正确修复漏洞程序的目的. 现有的补丁验证方式主要依赖于静态分析[20,21]和漏洞利用技术[18,19]对修复后的漏洞程序进行程序分析, 如果修复后的漏洞程序中无法分析出修复前的漏洞问题, 且没有产生新的漏洞问题, 则认为该补丁成功消除了该漏洞程序中的漏洞问题.

● 漏洞修复示例. 为了帮助读者理解漏洞修复的过程, 本节以漏洞修复工具 ExtractFix[19]和 Coreutils 程序中一个内存覆盖安全隐患为例介绍漏洞自动修复的一般流程. 图 5 展示了 fillp 函数调用的内存拷贝函数 memcpy(p, q, s) 存在内存安全问题的代码, 该函数将字符串 q (即 r) 中 s (即 i 和 size–i) 个字符复制到 p (即 r+i) 中, 不恰当使用会

导致 p 与 q 出现内存覆盖问题. 为了修复该漏洞, 在漏洞定位阶段, ExtractFix 采取动静态结合的方法, 首先在配置消毒器[30]的程序上动态执行测试用例, 消毒器监控程序动态状态并判断是否违背安全约束, 此安全约束包括预先定义的动态程序性质 (即消毒器)、安全人员编写的测试断言、API 手册提供的功能等. 对于各个程序语句, 当安全约束违背时消毒器强制崩溃程序以暴露漏洞现象, 紧接着 ExtractFix 以程序崩溃位置为源点进行数据流分析, 定位引发内存问题的根因位置. 在该示例中为了防止内存覆盖, memcpy(p, q, s) 应满足预先定义在消毒器中的 $p+s \leqslant q \vee q+s \leqslant p$ 安全约束, 然而当输入 size=13, i=6 时, 第 7 行处 r+(13–6)>r+6, 此时约束不满足, 消毒器强制崩溃程序, ExtractFix 继而从程序崩溃处进行反向数据流分析以定位漏洞根因, 最终定位到第 4 行处代码. 在补丁生成阶段, ExtractFix 基于程序语义合成补丁, 通过将补丁生成任务规约为约束求解与程序合成任务以修复漏洞程序. 整体而言, ExtractFix 采用一种基于反例制导的程序合成技术[31], 生成的补丁会用于验证已有的测试用例, 一旦测试用例无法通过验证, 该测试用例对应的约束会被添加到现有的约束中. 在该示例中, ExtractFix 在程序应满足约束中额外加入 $size = 13 \wedge i = 6 \wedge (p+s \leqslant q \vee q+s \leqslant p)$, 经过约束求解, ExtractFix 最终生成如图 6 所示补丁.

```
1  void fillp (char *r, size_t size){
2    ...
3    r[2] = bits & 255;
4    for (i = 3;  i < size / 2 ; i *= 2)
5      memcpy(r + i, r, i);
6    if (i < size)
7      memcpy(r + i, r, size - i) ;
8    ...
9  }
```

图 5 漏洞示例代码

```
- for (i = 3; i < size / 2; i *= 2)
+ for (i = 3; i <= size / 2; i *= 2)
```

图 6 ExtractFix 生成的补丁示例

在补丁验证阶段, ExtractFix 重新运行测试用例 (size=13, i=6) 以验证该补丁有效性. 漏洞程序在测试输入为 size=13, i=6 时, 执行成功且测试通过, 因而 ExtractFix 判定补丁有效并返回给用户. 该示例基本涵盖漏洞自动修复方法的主要环节, 包括漏洞定位、补丁生成与补丁验证, 本文第 3 节和第 4 节对漏洞自动修复方法展开详细论述.

2.3 漏洞修复效果评价指标

漏洞自动修复工具生成的补丁整体上分为两类: 无效补丁 (nonsensical patch) 和有效补丁 (valid patch). 其中, 无效补丁是指不能通过验证模块的补丁, 有效补丁是指能通过验证模块的补丁. 此外, 由于在补丁验证阶段存在过拟合现象, 即能通过验证的补丁并不一定能够正确地修复漏洞, 因此有效补丁又分为似真补丁 (plausible patch) 和正确补丁 (correct patch). 由于补丁验证存在其局限性, 似真补丁以改变程序原始语义的方式通过了补丁验证, 但没有达到正确修复漏洞的目的, 而正确补丁则为正确修复漏洞问题的补丁. 目前衡量漏洞自动修复工具的漏洞修复效果时, 广泛使用的评价指标是召回率 (recall) 和准确率 (precision). 其定义如下:

$$召回率 = \frac{生成正确补丁的漏洞数}{漏洞总数},\ 准确率 = \frac{生成正确补丁的漏洞数}{生成有效补丁的漏洞数}.$$

召回率衡量漏洞修复工具的修复漏洞能力, 召回率越高表示能修复越多的漏洞. 准确率衡量修复工具生成补丁的质量, 修复工具的准确率越高表示其生成补丁的质量越好. 在实际场景中, 召回率和准确率存在相互制约的可能性, 召回率越高代表工具能够修复更多的漏洞, 这就要求该工具需要在补丁生成方式上具有更大的包容性, 生成更多种类的补丁, 这却会导致生成似真补丁的可能性增大, 致使补丁的准确率降低. 因此, 现有漏洞自动修复技术往往需要在两者之间做出利弊权衡. 例如, 特定类型漏洞修复技术限定在某几种类型漏洞范围内进行修复, 在舍弃一部分修复能力的前提下提高了修复准确率; 反之, 通用类型漏洞修复技术不限定漏洞范围, 在保留较强修复能力的前提下倾向降低修复准确率. 对于漏洞修复任务而言, 部分工作也会考虑 Top-N (正确补丁位于工具输出补丁列表的前 N 个) 补丁的召回率、准确率等.

2.4 漏洞修复的漏洞类型

现有漏洞修复技术根据是否针对特定类型漏洞可以分为特定类型漏洞的修复方法和通用类型漏洞的修复方

法. 表 1 总结了特定类型漏洞修复与通用类型漏洞修复的概述、前置条件与优缺点.

● 特定类型漏洞的修复方法. 针对特定类型漏洞的自动定位、补丁生成以及补丁验证的整体过程. 常见漏洞数据库如 CWE 将各种漏洞分类为不同类型, 同一类型漏洞在发生机制与修复方式上有一定相似性. 以 CWE-787 (https://cwe.mitre.org/data/definitions/787.html)/CWE-125 (https://cwe.mitre.org/data/definitions/125.html) 缓冲区溢出漏洞为例, 缓冲区是操作系统内存中一段连续的地址空间, 当程序尝试读/写该缓冲区边界之外数据时, 便会触发缓冲区溢出漏洞. 为了修复缓冲区溢出漏洞, 研究者们分析程序中出现的缓冲区数据结构以发现数据读写越界的可能性, 对于可能发生读写越界的缓冲区提供数组扩容、安全函数替换等修复模式. 整体而言, 特定类型漏洞的修复方法的前置条件为该类型漏洞的定位、补丁生成具有一般规律, 对于定位与补丁生成不存在规律性的漏洞类型, 现有特定类型漏洞的修复技术方法不适用. 特定类型漏洞的修复方法优点在于该类方法基于特定类型漏洞的语法语意特征设计定位与补丁生成算法, 因而精确度高, 缺点在于现有漏洞类型繁杂、不断增加, 且人工设计特定修复算法门槛高. 本文在第 3 节展开论述这部分内容.

● 通用类型漏洞的修复方法. 不限定漏洞类型的自动定位、补丁生成以及补丁验证的整体过程. 根据 CWE 的数据显示, 截至本文撰写时 (2022 年 1 月) CWE 收录的漏洞类型已经达到了 924 个 (https://cwe.mitre.org/), 因此, 设计实现不限定类型的通用的漏洞修复方法裨益极大. 本文作者针对通用类型漏洞修复方法的论述不考虑复用现有的缺陷自动修复技术 (读者可参考缺陷修复相关综述[13,14]), 而聚焦于如何将漏洞修复规约为通用的文本生成任务, 根据历史漏洞修复数据集基于学习算法训练修复模型进而修复漏洞. 通用类型漏洞的修复方法依赖于较高质量的历史漏洞修复数据集, 否则无法充分训练学习模型. 通用类型漏洞的修复方法优点在于无须针对特定类型漏洞设计修复算法便可提供端到端漏洞修复解决方案, 缺点在于重度依赖数据集, 现有方法精确度较低且可解释性差. 考虑到现有通用类型的漏洞修复方法主要基于历史驱动的方法, 我们不再将通用类型漏洞的修复方法相关内容单独成段而是合并入第 4.3 节 (基于历史驱动的方法) 综合展开论述.

表 1 自动修复漏洞类型

研究方向	方法概述	方法前置条件	优点	缺点
特定类型漏洞修复	针对特定类型漏洞设计实现修复算法	漏洞的定位、补丁生成具有一般规律	基于特定类型漏洞的语法语意特征定位与修复, 精确度高	漏洞类型繁杂; 人工设计特定修复算法门槛高
通用类型漏洞修复	将漏洞修复规约为通用的文本生成任务	较高质量的历史漏洞修复数据集	端到端解决方案; 可复用现有的学习模型	依赖数据集; 精确度低; 可解释性差

2.5 漏洞修复的技术原理分类

除了按照修复漏洞的类型对现有技术进行归纳总结, 本文中我们进一步按照相应方法所采用的技术原理进行分类. 按照此分类方式, 现有漏洞自动修复技术可分为基于静态分析的漏洞修复、基于动态分析的漏洞修复、基于历史修复的漏洞修复和混合式的漏洞修复, 如表 2 所示.

表 2 漏洞修复技术原理概览

技术原理	方法概述	方法前置条件	优点	缺点
静态分析	分析程序静态性质制导定位、补丁生成	静态分析算法	无需执行程序和构造漏洞利用、部署容易	警报多、误报多、验证困难
动态分析	收集分析程序执行动态信息制导修复	程序可执行、漏洞利用可得/可构造	精确度高、补丁验证容易	需要执行程序、漏洞利用构造难、开销大
历史修复	抽取已有漏洞修复信息制导修复	较高质量的历史漏洞修复数据集	端到端解决方案; 可复用现有的学习模型	依赖数据集; 精确度低; 可解释性差
混合式	结合不同类型技术制导修复	各类型技术前置条件满足	可结合各类型技术的优点	具有各类型技术的缺点

基于静态分析修复漏洞在不执行程序的情况下, 分析程序静态性质, 制导漏洞定位与补丁生成. 以静态分析作为技术原理的研究方法需要针对漏洞发生机制密切相关的程序性质进行抽象与解释, 进而利用静态分析框架计算

程序的静态性质. 例如 Lee 等人[32]针对 C 程序中的内存错误操作问题, 分析变量的内存分配与释放操作, 该方法在修复阶段将内存释放问题简化为精确覆盖问题[33], 并使用 SAT (satisfiability) 求解器来求解正确修复程序. 表 2 列举了该类方法的优缺点, 本文在第 4.1 节展开论述这部分内容. 基于动态分析修复漏洞需要实际执行程序, 收集分析程序执行动态信息制导修复. 以动态分析作为技术原理的研究方法需要满足的前置条件包括程序可执行、可插装、漏洞利用可得/可构造等. 表 2 列举了动态分析方法的优缺点. 近年来测试驱动缺陷修复 (test-driven automated program repair, APR) [34−44]取得了长足发展, 考虑到漏洞作为缺陷的一种特定类型, 理论上可以参考这些技术的设计与实现, 部分研究者针对这一方向展开研究. 例如文献 [45] 尝试首先执行模糊测试以生成有效漏洞利用, 进而基于已有的测试驱动缺陷修复工具修复该漏洞, 该工作实证研究显示基于动态分析自动修复漏洞依然存在各类挑战. 本文在第 4.2 节展开论述这部分内容. 基于漏洞的历史修复数据进行漏洞修复, 从历史修复数据集中抽取已有漏洞修复信息制导修复. 2012 年 Hindle 等人[46]提出编程语言是自然语言的一种, 具有可重复、可预测的统计学特征, 因此将漏洞修复问题规约为数据驱动的文本生成任务有一定可行性. 表 2 列举了该类方法的优缺点, 本文在第 4.3 节展开论述这部分内容. 混合式漏洞修复技术通过结合不同类型技术制导修复, 由于漏洞修复问题的复杂性, 一些研究者针对特定类型漏洞合理结合各类型技术的优点设计修复算法. 例如, Gao 等人[47]为了缓解静态分析误报多问题, 在漏洞定位阶段采用静态分析与动态符号执行[48]相结合算法. 本文在第 4.4 节展开论述这部分内容.

3 特定类型漏洞的修复方法

本节阐述现有特定类型漏洞的修复方法. 漏洞数据库 CWE 每年总结当年度危害性最大、影响面最广的漏洞类型, 如后文表 3 所示是 2021 年 CWE 收录的排名第 1–25 综合影响力最大漏洞类型 (https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html), 为了便于更好地阐述, 本文将之进一步分类为缓冲区溢出、跨脚本攻击、输入验证攻击、注入攻击等. 针对每种漏洞类型, 后文表 3 同时列出现有相应修复方法所对应代表文献[15,18−24,28,29,32,47,49−52]. 整体而言, 研究者们普遍关注 (CWE 影响力排名第一的) 缓冲区溢出漏洞的自动修复方法[18−20,23,24,47], 图 7 列举了漏洞自动修复研究工作在修复特定类型漏洞方面的分布情况. 此外, 研究者们针对整型溢出和安全验证错误相关漏洞也发表多篇研究工作. 然而, 即便作为当前影响力最大的漏洞类型之一, 请求伪造和权限设置错误等漏洞类型当前尚未有研究者提出相应的漏洞自动修复方法. 本节紧接着对现有研究工作中针对缓冲区溢出、整型溢出、安全验证错误等特定类型漏洞的修复方法展开阐述.

3.1 缓冲区溢出漏洞的修复

缓冲区溢出是所有漏洞根因中最经典、分布最广泛的一类漏洞. 缓冲区是操作系统内存中一段连续地址空间, 当程序尝试在缓冲区边界之外读写数据时, 便会触发缓冲区溢出问题. 在图 8 展示的缓冲区溢出漏洞示例中, 给定长度为 100 字节的缓冲区 src 以及长度为 10 字节的缓冲区 buf (第 1,2 行), src 被初始化为 50 字节的字符串 (第 3,4 行). 指针 dst 被设置为 buf (第 5 行). 此时调用 strcpy 函数复制 src 中的字符串到 dst 指向的缓冲区 (第 6 行), 由于 buf 仅能写入 10 字节内容, 而 src 包含 50 字节内容, 因此缓冲区溢出发生. 根据 CVE[1]的数据显示, 在 2021 年公开发布的 19 533 漏洞中, 溢出漏洞占比高达 12.6%. 为了更好地解决缓冲区溢出漏洞问题, 一些研究工作如 CCured[53]和 Cyclone[54]通过代码标注机制扩展原生的 C 代码以实现特定语言的类型增强. 但是这类技术要求开发人员人工标注代码, 给开发人员带来额外负担的同时, 无法有效优化历史遗留代码. 大量研究工作在不断探索如何高效检测缓冲区溢出漏洞的方法: 如静态方法[55−61]利用静态分析技术在无须执行程序的条件下输出一组可能引起缓冲区溢出的可疑代码位置, 静态方法主要问题在于误报过高, 即便使用目前业界使用最广泛的分析器, 误报率也达到 30% 以上[7]; 又如动态方法[62−66]则利用模糊测试等技术记录并分析程序的动态信息, 精确度高于静态分析方法. 动态分析方法优点在于结果精确, 主要缺点在于开销过高、代码覆盖率过低等.

为了更好解决缓冲区溢出问题, 研究人员不断提出多种漏洞自动修复方法, 表 4 列举了修复缓冲区溢出漏洞的代表性工作, 并阐述了各个工作在漏洞定位、补丁生成和验证过程的核心方法, 以及其修复方法所依赖技术, 例如 (可动态触发漏洞的) 漏洞利用、静态分析器以及漏洞修复数据集.

表 3 特定类型漏洞的修复方法概览

漏洞类型	CWE编号	CWE描述	影响排名	代表文献
缓冲区溢出	CWE-787	数组越界写	1	Sidiroglou-Douskos等人[23,24], Shaw等人[20], Gao等人[47], Huang等人[18], Gao等人[19]
	CWE-125	数组越界读	3	
	CWE-119	不正确缓冲区内存操作约束	17	
跨脚本攻击	CWE-79	跨脚本攻击	2	Mohammadi等人[49]
输入验证攻击	CWE-20	不正确输入验证	4	Mohammadi等人[49], Long等人[21]
	CWE-22	不正确路径名约束	8	
	CWE-434	未受限制上传文件危险类型	10	
	CWE-611	不正确XML外部实体引用约束	23	
注入攻击	CWE-78	OS命令注入	5	Yang等人[15]
	CWE-89	SQL注入	6	
	CWE-502	反序列化不受信任数据	13	
	CWE-77	命令注入	25	
内存释放错误	CWE-416	释放后使用	7	Lee等人[32]
	CWE-476	空指针引用	15	
请求伪造	CWE-352	跨站请求伪造	9	暂无
	CWE-918	服务器请求伪造	24	
安全验证错误	CWE-306	关键函数身份验证缺失	11	Ma等人[50], Zhang等人[22]
	CWE-287	身份验证错误	14	
	CWE-798	硬编码证书	16	
	CWE-862	身份验证缺失	18	
	CWE-200	非信任用户敏感信息泄露	20	
	CWE-522	非充足证书	21	
整型溢出	CWE-190	整型溢出	12	Cheng等人[28], Coker等人[29], Long等人[21], Wang等人[51], Muntean等人[52], Huang等人[18], Gao等人[19]
权限设置错误	CWE-276	不正确默认权限	19	暂无
	CWE-732	关键资源不正确缺陷控制	22	

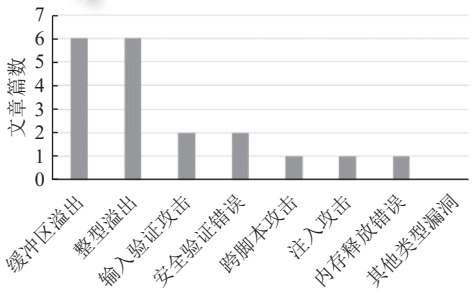

图 7 漏洞自动修复工作在特定类型漏洞方面的分布

```
1     char buf[10];
2     char src[100];
3     memset(src,'c', 50);
4     src[50] = '\0';
5     char *dst = buf;
6 -   strcpy(dst, src); // 引入缓冲区溢出漏洞
6 +   gstrlcpy(dst, src, sizeof(buf)); // gstrlcpy替换strcpy
```

图 8 缓冲区溢出漏洞示例

Sidiroglou-Douskos 等人[23,24]推测漏洞程序与其对应的健康程序应该有相似的行为: 给定一个在漏洞程序上通过的测试输入 t_a 以及一个失败的测试输入 t_b, 而 t_a 和 t_b 在健康程序上都能通过. 基于这一观察, Sidiroglou-Douskos 等人做出一个假设: 健康程序含有漏洞程序缺失的分支检查, 使得健康程序能够正确处理漏洞程序的失败测试输入. 依据这一假设, 他们从漏洞程序的历史版本中搜索出由健康程序组成的供体 (donor) 代码库, 从中找到 t_a 和 t_b 测试均能通过的健康代码, 并相应地搜索漏洞代码缺失的代码分支判断语句. 在漏洞根因定位 (分析补丁所需插入的代码位置) 阶段, 该方法重点关注缓冲区溢出类型漏洞, 因而仅插装与内存操作相关的代码, 记录内

存相关操作对于代码分支判断语句的依赖关系, 进而通过调用 SMT (satisfiability modulo theories) 约束求解器求解可满足的隐式信息流传递的约束条件, 将内存操作含有隐式信息依赖关系的分支语句识别为插入遗漏语句的位置. 其次, 在补丁生成阶段, 该工作基于符号化表达式泛化健康程序中的分支语句, 使其可移植到漏洞程序, 当分支条件成功移植后, 该方法在分支体内插入人工修复模板如 exit() 语句以避免缓冲区溢出发生. 最后, 在验证阶段, 该工作针对修复后程序重新执行 t_a、t_b, 根据测试结果验证修复的正确性. 整体来看, 该工作作为缓冲区漏洞修复的早期工作, 方法相对简单且局限性较大: (1) 在漏洞定位阶段, “健康程序含有漏洞程序缺失的分支检查”是一种理想假设, Zhang 等人[22]指出漏洞补丁所需的关键组件通常具有特殊性, 较难从相似程序中直接复用, 此外该方法定位效果依赖于 SMT 约束求解器; (2) 在漏洞补丁生成阶段, 该方法简单地插入 exit() 语句跳过对缓冲区边界之外读写数据的操作, 修改方式单一, 难以生成高质量的修复补丁, 例如现有工作[47]显示对于修复缓冲区溢出漏洞可以采用增加边界检查、API 替换、扩大缓冲区大小等各种开发者常用的修复模式; 不同缓冲区溢出漏洞需要不同的修复模式才能产生高质量的正确补丁. 综上所述, 缓冲区漏洞程序修复在补丁生成空间和补丁生成质量的探索方面仍存在很大不足, 需要研究人员的进一步探索与尝试.

表 4　缓冲区溢出漏洞修复方法的代表性工作概览

文献	方法依赖	漏洞定位	补丁生成	修复验证
Sidiroglou-Douskos 等人[23,24]	漏洞利用	隐式信息流分析相似程序动态分析	基于人工模板	动态验证
Shaw等人[20]	静态分析器	C代码层次的别名分析数据流分析	安全API/数据结构替代	静态分析算法性质保证
Gao等人[47]	静态分析器	静态分析符号执行	基于历史修复获取到的通用修复模板	动态验证
Huang等人[18]	漏洞利用	符号执行	程序合成	动态验证
Gao等人[19]	漏洞利用	地址消毒、数据流分析、符号执行	约束求解反例制导程序合成	动态验证

Shaw 等人[20]观察到两类很重要的缓冲区漏洞根因: 使用非安全的 API 和使用非安全的数据结构. 早期的 C/C++设计人员在设计 API 过程中并未很好地兼顾安全属性, 因此早期 C/C++的 API 函数存在安全风险; 近年来 C/C++程序暴露出一系列安全漏洞, 尤其是缓冲区溢出类型漏洞, 这一问题促使 C/C++设计人员为非安全的 API 函数重新封装并公布安全版本的 API 实现. 基于这一观察, Shaw 等人[20]扩展 OpenRefeactory/C, 在 C 代码层面添加了定义可达性分析、指针分析、控制/数据类分析等分析能力以支撑源码层面的程序转换. 基于这一程序转换能力, 在不影响程序语义前提下, 该方法将非安全 API 和数据结构替换为安全 API 和数据结构, 从算法设计层面该方法保证了分析结果的安全性. 对于图 8 的缓冲区溢出漏洞, 该方法替换非安全 API strcpy 为安全 API gstrlcpy, 以生成图 8 所示的修复补丁. 该方法的不足之处在于插装程序引入额外运行时开销, 且仅限于修复这两类缓冲区漏洞问题.

Gao 等人[47]试图对静态分析器报告的缓冲区溢出类型漏洞设计自动修复方法. 一个关键挑战在于使用静态分析检测缓冲区溢出类漏洞误报率过高, 为了缓解误报过高的问题, 该方法利用符号执行技术生成测试用例以动态确认缓冲区溢出警报的真实性, 当警报被动态确认, 该方法相应调用补丁生成算法. 为了指导符号执行探索路径, 该方法在警报语句与内存操作敏感语句之间进行可达性分析, 对于不满足可达性的警报语句直接过滤, 对于满足可达性分析的警报语句调用符号执行方法以检查路径约束条件是否满足. 紧接着, 该方法为满足符号执行路径约束条件的警报生成测试用例, 并基于历史修复人工总结出一组通用的缓冲区漏洞修复模板用于补丁生成. 最后, 生成的测试用例与补丁方案被同时推荐给用户, 由用户确认该修复方案的正确性. 该方法能够有效降低静态分析高误报率, 对于中等规模 (小于 10 万行) 的程序表现优秀, 由于方法使用了开销较大的符号执行算法, 其在大型程序上表现不佳.

Huang 等人[18]采用程序合成 (program synthesis) 的方法合成漏洞修复补丁. 不同于基于测试用例执行过程中观察到的具体值生成程序约束, 该方法预先定义了一组针对缓冲区溢出漏洞的安全约束, 进而调用动态符号执行技术以检测不满足安全约束的可疑表达式, 对于检测到的可疑表达式, 该方法基于一组预先定义的修复模板生成

相应的补丁. 受该方法的启发, Gao 等人[19]在编译阶段插装一组检测安全约束的地址消毒器, 当执行漏洞利用时, 消毒器检测到安全约束被违背进而强制崩溃程序; 对于崩溃位置, 该方法使用反向数据流分析定位可能的漏洞根因位置, 并使用符号执行技术在不同代码位置处传播安全约束; 对于候选的漏洞根因位置, 该方法采用反例制导的程序合成技术[31]生成满足安全约束的修复补丁. 实验结果显示该方法修复漏洞的召回率达到 50%, 准确率达到 65.1%.

整体而言, 缓冲区漏洞可规约为对给定缓冲区边界之外非法地址空间的读写访问, 相应的修复关注于地址空间的表示与分析, 以制导补丁生成, 典型的方法包括利用静态分析建模与分析地址指针的取值范围, 以及利用动态分析判断缓冲区溢出相关的安全约束是否违背. 基于静态分析的技术有着部署便捷、分析结果满足安全性等优点; 然而, 现有研究[7]显示静态方法检测漏洞时存在警报多、误报多的问题. 基于动态分析的技术则有着结果精确的优点, 其主要难点在于人工构造漏洞利用对应的程序输入难度大、效率低. 该方向的研究一大潜在机遇在于通过探索先进的动态分析技术精确地检测出程序中漏洞问题, 以提高漏洞自动修复技术的效果. 在各类动态分析技术中, 模糊测试技术结合各类消毒器技术是当前检测零日漏洞的最有效手段之一[67], 该技术通过随机变异测试输入以试图探索更多的程序行为.

3.2 整型溢出漏洞的修复

当程序为变量赋值超过其类型表达范围的数值时, 整型溢出漏洞可能发生. 在图 9 展示的整形数据相乘引入整型溢出漏洞的示例中, 变量 data 没有经过数值大小检查, 此时进行数据相乘其结果可能超出整型变量 result 的取值范围, 进而引发整型溢出. 针对这一类型漏洞, 研究者提出了不同的修复方法, 表 5 列举了修复整型溢出漏洞的代表性工作. 传统的整型溢出漏洞修复不考虑单独的定位阶段, 而是将代码中可能暴露整型溢出漏洞的语句全部进行类型增强. 例如 Coker 等人[29]使用具有二进制补码、无限大小的整数编码来代替原始有界整数, 该方法设计了 3 种代码转换模板: 添加整数转换 (add integer cast, AIC)、替换算术运算符 (replace arithmetic operator, RAO) 以及更改整数类型 (change integer type, CIT) . 该方法可以有效避免一些整型溢出漏洞问题, 却也引入了额外的运行时开销. 此外, 并不是所有整型溢出问题都会导致程序漏洞, 有些整型溢出问题是良性的, 无需修复, 如用于特定哈希值\随机数生成的功能场景, 此时该方法强制替换程序中所有满足代码转换条件的语句可能违背程序语义.

```
1 +    if(data <= sqrt(2147483647) && data >= -sqrt(2147483647) {
2        int result = data * data; // 当result可能超过INT_MAX时，跳过该次执行
3 +      printIntLine(result);
4 +    } else {
5 +      error.log("may lead integer overflow.")
6 +    }
```

图 9 整型溢出修复补丁示例

表 5 整型溢出漏洞修复方法的代表性工作概览

文献	分析输入	漏洞定位	补丁生成	修复验证
Coker等人[28,29]	—	—	类型增强	—
Long等人[21]	静态分析器	On-demand静态分析	插入输入过滤函数	静态分析算法性质保证
Wang等人[51]	漏洞利用	动态污点分析	改变控制流/退出	动态验证
Muntean等人[52]	静态分析器	符号执行、SMT求解	基于人工模板	动态验证
Huang等人[18]	漏洞利用	符号执行	程序合成	动态验证
Gao等人[19]	漏洞利用	地址消毒、数据流分析、符号执行	约束求解反例制导程序合成	动态验证

注: “—”代表该文献对应漏洞修复方法在对应功能模块上无特定设计与实现, 同表6, 表7

Long 等人[21]提出一种可靠的 (sound) 整型溢出输入过滤方法. 在漏洞定位阶段, 该方法实现了一套按需的 (on-demand) 静态分析算法, 将可疑的内存分配等操作设置为起点 (source), 反向遍历控制流图, 在程序输入位置生成符号化的约束条件表达式. 当用户提供程序输入时, 该方法便可根据约束条件判断该输入是否有可能 (may-

analysis) 引发整型溢出漏洞. 在漏洞修复阶段, 该方法直截了当地屏蔽后续代码的执行以达到避免漏洞发生的目的来修复该漏洞. 该算法在静态分析设计上是可靠的, 该性质保证了不会漏报任何可能引发整型溢出的漏洞; 相对的, 该算法在过滤输入时存在误报问题, 可能过滤合法输入.

Wang 等人[51]提出 SoupInt 方法, 尝试区分恶性和良性整型溢出, 针对恶性整形溢出使用动态污点分析来跟踪其传播. 如果 SoupInt 发现整数溢出影响安全敏感操作 (例如, 影响内存分配函数的大小参数), SoupInt 认为这个整数溢出是恶性的. 针对该恶性的整数溢出, SoupInt 在漏洞位置插入改变控制流或者 exit 语句以避免整型溢出发生. 该工作实验评估显示对于给定的 10 个整型溢出漏洞, SoupInt 能够正确修复其中的 9 个漏洞.

Muntean 等人[52]在现有工作基础上, 针对无输入场景, 提出基于约束求解的整型溢出修复方法. 该方法相较于其他工作最大的创新在于利用符号执行将漏洞定位与修复阶段整合在一个约束求解的问题, 进而调用 SMT 约束求解器进行求解. 通过这种方式, 该方法仅针对确认能触发的整型溢出漏洞进行修复. 在补丁生成环节, 该方法针对 4 种常见的整型运算人工设计了修复模板. 图 9 展示了该方法基于相同两数相乘模板[52]修复前述整形溢出漏洞生成的补丁, 该补丁提供了对变量数值的检查, 当待相乘变量的绝对值大于 INT_MAX 的开方时, 不再进行数据相乘, 以避免发现整型溢出. 该工作的实验评估显示, 对于给定的 2 102 个 C 语言程序, 该方法能够以规避整型溢出发生的方式修复所有程序. 但考虑到真实程序中包含良性的整型溢出, 该方法生成的修复补丁可能仅是针对鉴定的假阳性整型溢出漏洞产生的似真 (plausible) 补丁. 如第 3.1 节所述, Huang 等人[18]的工作以及 Gao 等人[19] 的工作提出了基于自定义安全约束的程序合成漏洞修复方法, 通过提供一组针对整型溢出的安全约束, 他们的方法可以合成满足相应约束的补丁程序, 进而修复整型溢出漏洞; 但是, 他们没有考虑整型溢出漏洞检测的假阳性问题.

整体而言, 整型溢出漏洞可规约为超出整型范围的数值读写, 相应的补丁修复应关注针对不同数值操作如何避免整型溢出的发生, 例如典型的数值运算操作包括变量、变量常量、加减乘除等. 研究者针对整型溢出漏洞修复方法在不断优化定位与补丁生成算法的精确度: 静态分析方法能够在保证安全性的前提下修复整型溢出问题, 然而很难在恶性与良性溢出之间做出区分. 动态方法能够部分区分恶性整型溢出, 然而无法保证算法所生成的所有补丁均针对恶性溢出, 可能会修改良性溢出而造成错误修改. 已有历史漏洞修复数据包含修复恶性整型溢出漏洞的样本信息, 未来的整型溢出漏洞修复工作需要更细致地考虑已有历史修复代码的上下文信息、语义乃至代码功能, 以更精确地识别恶性整型溢出漏洞并生成高质量的修复方案.

3.3 安全验证错误型漏洞的修复

现代软件开发大量使用应用程序接口 (application programming interface, API) 的设计与实现, 使用这些 API 可以在无需访问源码、或理解内部工作机制的条件下, 访问硬件/软件的特定资源与功能, 这一方式也存在于安全验证功能的开发当中. 然而, 基于 API 开发安全验证功能的方式加剧了软件安全隐患: 软件程序违反 API 明确行为或隐式使用约束可能引入安全漏洞. 图 10 展示了一个安全验证错误型漏洞示例,

```
1 -   SecretKey $v_0 = new SecretKeySpec(StringLiterals.CONSTANT.getBytes() , "AES"); // 传入常量
1 +   SecureRandom $v_1 = new SecureRandom() ;
2 +   String $v_2 = String.valueOf($v_1.nextInt());
3 +   byte[ ] $v_3 = $v_2.getBytes();
4 +   $v_3 = Arrays.copyOf($v_3, 24);
5 +   SecretKey $v_0 = new SecretKeySpec($v_3, "AES"); // 传入随机生成的变量值
```

图 10 安全验证错误型漏洞示例

该示例中名为 SecretKeySpec 的 API 用于生成安全密钥, 其正确初始化方法需要传入非常量参数, 而该示例中是错误地传入常量参数, 从而引入安全验证错误型漏洞. 现有研究显示当 API 文档缺失或第三方依赖库频繁更新时, 开发者常常无法正确使用安全验证相关的 API. 为了修复安全验证 API 误用导致的程序漏洞, 研究者们提出了若干工作 (见表 6). 由于安全验证错误型漏洞主要由 API 误用导致, 现有专注于该类漏洞的修复工作主要针对 API 误用漏洞问题. API 误用漏洞在 CWE 数据库中有多个实例, 例如 CWE-227 指代 API 滥用 (API abuse)、CWE-475 指代未定义行为的 API 输入 (undefined behavior for input to API)、CWE-1001 指代使用非正确的 API (use of an improper API).

表 6 安全验证错误型漏洞的修复方法的代表性工作概览

文献	分析输入	漏洞定位	补丁生成	修复验证
Ma等人[50]	静态分析器	静态切片	基于人工修复模板	—
Zhang等人[22]	漏洞修复数据集	缺陷/修复代码片段相似匹配	过程间数据流分析; 修复模板泛化	—
Kechagia等人[10]	基于缺陷修复工具	基于缺陷修复工具	基于缺陷修复工具	动态验证

Ma 等人[50]针对安卓应用广泛存在的安全验证类 API 误用问题, 将安卓应用安全验证类 API 误用分为 7 类并且人工收集对应的修复模式修复该类漏洞. 该方法包含 API 误用定位和修复两个阶段, 定位阶段采用轻量级的静态切片技术, 修复阶段则基于人工收集的修复模板进行补丁生成. 该工作成功修复了 1 262 个安全验证类 API 误用漏洞中的 1 193 个, 召回率达到 94.5%. 结合该方法的设计实现与实验结果, 对于具有特定调用范式的 API 代码片段 (不仅是该方法中安全验证类 API), 设计与实现 API 误用漏洞的定位、修复算法具有较高的可行性, 其关键技术在于设计可以表达/分析 API 调用范式的算法.

Zhang 等人[22]提出从 API 误用的漏洞代码和相应修复代码的代码对中自动提取漏洞代码与相应补丁代码的特征, 并依据提取出的特征与代码进行批评来引导漏洞定位和修复. 为了提高匹配精确度, 该方法对于更新操作涉及的变量进行过程间数据流分析以发现使用该表达式的任何安全类 API, 并将其定义为关键 API (critical API). 在修复阶段, 基于关键 API 可以更好地发现其他相似漏洞. 为了提高泛化程度, 该方法进一步将代码对中的具体变量名抽象为临时变量, 以提高所推导程序转换模板应用在其他漏洞代码上的能力. 泛化后的修复模板包含两部分: 漏洞代码模板和对应的修复模板, 模板中所有的具体变量名已被抽象为临时变量. 通过大量收集这样的漏洞-修复代码对, 该方法获得由一组修复模板组成的修复仓库. 给定漏洞程序代码, 该方法匹配修复仓库中与其相似的漏洞代码, 并实例化相应的修复模板进行补丁生成. 相较于 Ma 等人的工作, 该工作无须针对 API 特定调用范式设计定位与修复算法. 针对图 10 所示漏洞, 该方法搜索修复仓库中与之匹配的修复模板, 通过实例化模板中形式参数生成图 10 所示的修复补丁. 现有测试驱动缺陷修复技术具有修复 API 误用的能力, 例如 Kechagia 等人[10]调研了 8 种缺陷修复技术, 包括传统的基于人工定义修复模板的方法以及基于挖掘大规模代码仓泛化学习修复模板的方法. 实验结果显示在包含 101 个简单的 API 误用缺陷的数据集上该方法仅能修复 28% 的缺陷 (召回率), 而正确修复的缺陷仅占 25% (准确率). 这对于特定类型缺陷而言修复效果不佳. 本文认为, 现有的缺陷修复工具并非针对安全验证类 API 误用相关漏洞设计的, 在算法设计层面还有较大改进空间.

整体而言, 安全验证类漏洞在发生机制、修复模式方面通常有其固有特征, 通过设计安全验证类 API 的检测与补丁生成算法, 现有工作能够较好修复部分该类漏洞. 在未来漏洞修复研究中, 一方面, 研究者应关注如何高效地修复存量代码中的安全验证类型漏洞; 另一方面, API 误用导致的安全验证类漏洞发生机制相对固定, 如何在开发阶段实时地检测出此类漏洞, 并给出可行的修复建议, 也是未来可行的研究点之一.

3.4 其他类型漏洞的修复

除了针对上述 3 种类型漏洞的修复工作之外, 研究修复其他特定类型漏洞的工作较少, 因此本节对这部分修复方法归类成其他类型漏洞的修复工作, 并做集中阐述 (见表 7). 表 3 列举的综合危害排名前 25 的注入攻击型漏洞包括 CWE-77[68], 78, 79, 89, 502 多个子类型, 近期发生的 Log4j2 高危漏洞 CVE-2021-44228 也是注入攻击的一种. 学术界对于注入攻击检测方面研究较多, 但对注入攻击的自动修复技术研究相对欠缺.

表 7 其他类型漏洞修复方法的代表性工作概览

文献	分析输入	漏洞定位	补丁生成	修复验证
Mohammadi等人[49]	漏洞利用	—	基于人工修复模板	动态验证
Yang等人[15]	静态分析器	信息流静态分析	基于历史修复模板	—
Lee等人[32]	静态分析器	类型静态分析	解算精确覆盖问题	静态分析算法性质保证
Nguyen等人[69]	静态分析器	符号执行/静态分析	基于人工修复模板	—

在 XSS 跨脚本攻击中, 恶意攻击者往 Web 页面里插入恶意 JavaScript 代码, 当用户浏览该页之时, 嵌入 Web 里面的 JavaScript 代码会被执行, 从而达到恶意攻击用户的目的. 当前业界最佳防御该类漏洞的手段是调用编码器编码不受信任的网页代码. Mohammadi 等人[49]观察到编码过多的代码会影响程序性能, 为了平衡安全性与功能性, 提出了一种动态的针对 XSS 跨脚本攻击 (一种注入攻击型漏洞) 的自动修复方法. 该方法设计了 6 种合理的编码器插入规则作为修复模板, 对于生成的修复重新执行单元测试进行验证. Yang 等人[15]则提出了一种静态的代码注入类漏洞检测与修复方法. 该方法首先使用 AppScan[70]静态扫描工具分析程序的信息流. 由于缺乏动态执行信息, 分析结果中包含大量由不可达路径 (infeasible path) 构成的信息流, 即静态分析结果误报. 因此, 该方法进一步基于两个不完整数据流构建单一完整数据流以减小误报. 具体而言, 该方法观察到一次成功的代码注入通常包含一对数据流: 第 1 条数据流是以不受信任输入为起点, 以数据存储节点为槽点; 第 2 条数据流是以数据存储节点为起点, 以数据使用点为槽点. 因此, 通过合并这两条数据流获得一条完整数据流以降低分析结果的误报率. 此外, 对于定位到的可疑漏洞代码, 该方法基于一组历史已有的修复代码设计修复模板用于补丁生成. 整体而言, 设计实现高效修复注入类漏洞的方法是极具挑战 (动态分析构造有效漏洞利用/静态分析可降低误报率) 且意义重大的, 未来该方向的研究工作应该着眼于特定子类型漏洞的发生根因与对应修复的剖析以提出实用、高效的定位与补丁生成算法.

Lee 等人[32]提出基于静态分析的修复工具 MemFix, 用于修复 C 程序中内存释放错误, 例如内存泄漏、双重释放和释放后使用错误. 它通过静态分析器为每个分配的对象生成候选修复, 其中能正确释放所有内存对象的修复方案为正确修复方案. 它的工作原理是将内存释放问题简化为精确覆盖问题[33], 并使用布尔可满足性问题 (boolean satisfiability problem, SAT) 求解器来求解正确修复程序. 由于静态分析算法设计满足安全性, MemFix 输出的修复方案不会引入新的错误. 在一组包含 100 个人工构造的内存释放错误的数据集上, MemFix 修复漏洞的召回率达到 37%, 准确率达到 100%, 揭示了该方法自动修复内存释放错误漏洞的有效性. 然而, 布尔可满足性问题是 NP 完全问题, 当前不存在指数时间内计算该类问题的通用算法. 因此当程序规模较大时, SAT 求解器在给定时限内通常判定问题不可解, 因此修复大规模程序中的内存释放错误漏洞依然是开放性问题. 其他修复方法如 Nguyen 等人[69]针对智能合约中的漏洞问题如 CVE-2018-10376, 提出了一种静态方法. 该方法首先使用符号执行收集智能合约的一组路径, 紧接着使用静态分析算法检测潜在的漏洞. 对于检测到的漏洞代码, 基于一组特定的修复模板进行补丁生成.

整体而言, 考虑到目前已知漏洞类型超过 900 种, 然而当前已有特定类型漏洞修复方法相对较少: 如图 7 所示, 现有的漏洞自动修复研究工作主要是针对缓冲区溢出、整型溢出等特定类型的漏洞; 对于其他类型的大量漏洞, 研究者仍需在洞悉漏洞发生与修复机制的基础上提出更多可行的修复方案. 例如, 在 Top 25 CWE 漏洞中, 当前研究工作鲜有关注请求伪造 (CWE-352) 类型漏洞的自动修复, 该漏洞允许攻击者通过伪造来自受信任用户的请求实现漏洞利用, 潜在的漏洞自动补丁方案包括提供输入过滤功能, 过滤掉含有站内操作 URL 的链接. 此外, 当前对于权限设置错误 (CWE-276, CWE-732) 类型漏洞, 没有自动修复相关研究, 对于该类漏洞, 未来的研究方向可关注程序行为与权限需求的一致性检测, 并探索相应修复方案.

4 漏洞修复的技术原理分类

在第 3 节, 本文根据修复漏洞类型的差异整理归纳了现有的漏洞修复方法, 列举了针对特定类型漏洞的不同修复方法. 为了进一步整理归纳现有漏洞自动修复方法, 本节对不同的漏洞自动修复方法归纳其技术原理, 包括动态分析、静态分析、历史驱动以及混合式技术 (如表 8 所示), 阐述了基于相似技术原理设计并实现各类漏洞修复方法的共性思路.

4.1 静态分析

以静态分析作为技术原理的研究方法主要是基于漏洞根因的静态特征选取合适的静态分析算法, 在不实际执行程序的情况下, 定位并修复漏洞发生可能 (may-analysis) 的根因代码. 该类方法的技术挑战在于给定特定漏洞的类型, 研究者需要深入理解漏洞发生过程其代码所具备的特征, 进而设计相应的静态分析算法以检测该类特征. 例

如, Shaw 等人[20]观察到一类缓冲区溢出漏洞的发生根因在于调用非安全函数, 该场景下非安全函数的调用便是相应的代码特征. 基于这一观察, Shaw 等人设计并实现了源码层面的指针分析技术以更精确地计算缓冲区长度, 进而将存在缓冲区溢出风险的非安全函数调用替换为安全函数调用以减少缓冲区溢出问题. Long 等人[21]观察到非法输入是一类整型溢出问题的发生根因, 而程序输入的入口点与可能的整型溢出位置满足某种可达性是该类漏洞发生过程的代码特征, 基于该观察 Long 等人设计了相应的静态分析算法: 考虑到以输入为源点设计正向分析算法难以扩展, 因而该方法以可能的整型溢出位置为源点设计反向分析算法, 在此基础上, 通过将输入参数符号化实现了按需静态分析算法. 按需 (on-demand) 分析将分析范围约束在给定的程序位置与数据事实 (data fact), 可以合理优化过程间分析的拓展性问题. Lee 等人[32]基于两步骤检测与修复内存分配与释放问题, 首先, 针对每个内存分配 (allocation) 语句, 该方法使用类型状态分析[33]解算出所有安全/非安全补丁, 此外, 考虑到对于一个 allocation 语句的安全补丁对于另一个 allocation 语句可能是非安全的, 该方法进一步综合不同 allocation 语句生成的补丁, 转化为约束表达式进行求解. Ma 等人[50]观察到, 在检测 Java 安全验证错误时, 基于验证 (verification) 的方法开销过大且需要人工干预, 提出了基于静态程序切片的分析算法以在分析精度和效率间做出权衡. 总体而言, 以静态分析作为技术原理设计漏洞自动修复方法要求研究者深入理解某一类漏洞的发生根因、识别漏洞发生过程代码所具备的特征, 并具备一定的静态分析算法的设计能力, 该类方法有着部署容易、无需配置执行环境以及无需构造漏洞利用的优点, 适合集成于各类门禁系统中以保证软件质量. 另一方面, 静态方法理论上具有无误报或无漏报的性质, 但实际的工程实现中考虑到算法效率、扩展性与兼容性往往牺牲了这些性质, 导致这类方法在实践总存在较高的误报/漏报率问题, 需要人工过滤误报的介入而降低其效率.

表 8　漏洞修复方法及其技术原理总结

技术原理		发表年份	代表文献	编程语言	可修复的漏洞类型	整体技术
静态分析		2013	Coker等人[29]	C	整型溢出	语言类型增强
		2014	Long等人[21]	C/C++	整型溢出	按需静态分析
		2014	Shaw等人[20]	C/C++	缓冲区溢出	别名分析、数据流分析
		2016	Ma等人[50]	Java	安全验证错误	控制流分析、静态程序切片
		2016	Cheng等人[28]	C	整型溢出	语言类型增强
		2018	Lee等人[32]	C/C++	内存释放错误	类型状态分析、约束求解
		2019	Muntean 等人[52]	C	整型溢出	过程间路径敏感分析、符号执行
		2021	Nguyen等人[69]	C/C++	整型溢出	符号执行、静态分析
动态分析		2014	Wang等人[51]	二进制代码	整型溢出	动态污点分析
		2015	Sidiroglou-Douskos等人[24]	C/C++	缓冲区溢出、整型溢出	动态插装、测试结果分析
		2019	Huang等人[18]	C/C++	缓冲区溢出、整型溢出、坏转型	动态符号执行
		2021	da Costa Meireles Barbosa 等人[45]	Java	不限类型	模糊测试、缺陷自动修复
历史驱动		2017	Ma等人[25]	Java	不限类型	基于树结构变更提取与特征匹配
		2018	Harer等人[26]	C/C++	不限类型	对抗神经网络
		2020	Chi等人[27]	Java	不限类型	Transformer
		2021	Chen等人[6]	Java	不限类型	Transformer\迁移学习
混合式技术	动态分析 静态分析	2015	Sidiroglou-Douskos等人[23]	C/C++	缓冲区溢出、整型溢出	约束求解、动态分析
		2019	Mohammadi等人[49]	HTML/JS	跨脚本攻击	静态分析、基于测试补丁验证
		2021	Gao等人[19]	C/C++	缓冲区溢出、整型溢出	符号执行、控制流/数据流分析、约束求解、程序合成
	静态分析 历史驱动	2019	Yang等人[15]	C/C++	注入攻击	静态信息流分析、历史修复模板生成
		2020	Gao等人[47]	C/C++	缓冲区溢出	静态分析、动态符号执行
		2021	Zhang等人[22]	Java	安全验证错误	过程间分析、历史修复模板生成

4.2 动态分析

相较于静态分析, 以动态分析作为技术原理的研究方法主要是基于漏洞发生时程序的动态信息辅助漏洞定位、补丁生成与验证, 以达到更好的修复漏洞的效果. Sidiroglou-Douskos 等人[24]观察到漏洞程序与对应健康程序的动态执行信息存在相似性, 基于这一观察 Sidiroglou 等人插装原程序以收集程序的动态执行信息, 通过比较漏洞程序与对应健康程序的动态执行路径来检测漏洞位置, 对于生成的补丁同样利用测试进行验证. Huang 等人[18]针对传统符号执行状态爆炸、约束求解困难的问题, 使用动态符号执行技术探索程序各个路径以更高效地发现漏洞位置并进行修复. Wang 等人[51]观察到基于静态分析判别良性/恶性整型溢出精度低且误报率高, 因而使用动态污点分析技术以更精确地识别恶性整型溢出问题. 整体而言, 以动态分析作为技术原理有助于在漏洞修复各个阶段采用程序的动态执行信息, 因此不存在误报问题. 当问题场景中动态分析的各个必要条件具备时 (构建、执行环境、漏洞利用可得等), 基于动态分析技术修复漏洞的精确程度更高.

考虑到软件漏洞是一类特殊的软件缺陷, 研究者们尝试了借鉴测试驱动缺陷修复的思想探索测试驱动的漏洞修复方法. Gao 等人[19]观察到程序漏洞往往无法由测试触发, 他们提出使用各种消毒器 (用于监控程序动态执行过程中出现的非安全状态)[71]. 执行过程中, 当程序不满足消毒器约束时, 该方法强制崩溃程序以暴露漏洞. 将崩溃视为程序缺陷, 他们借鉴已有的缺陷修复技术开展漏洞修复. 该方法首先捕获必须满足的输入约束, 进而使用崩溃位置作为起点, 并使用控制/数据依赖关系分析技术找出候选补丁生成位置. 接着, 以约束以及候选修复位置为输入, 该方法合成修复程序, 使修复程序在崩溃位置满足相应的约束. 最后, 该方法通过重新执行测试, 验证修复方案的有效性. 文献 [45] 观察到模糊测试是检测漏洞的最佳实践方案, 将模糊测试与测试驱动缺陷修复结合起来, 理论上应该会是有效的漏洞修复方法. 然而, 该工作的实验结果显示其无法修复任何漏洞: 在漏洞定位阶段, 该方法提供单一的能够暴露错误的漏洞利用, 而现有程序缺陷修复技术在缺陷定位阶段通常采用基于频谱的缺陷定位 (spectrum-based fault localization, SBFL) 方法, SBFL 依赖于一组成功和失败测试用例在执行过程中的程序覆盖信息, 因此对于仅包含单一有效漏洞利用的程序定位效果不佳; 在补丁生成阶段, 漏洞修复的补丁生成往往涉及到特定 API 的少见表达式以及多行修改[22], 现有缺陷修复工具无法有效生成可行的修复方案. 整体而言, 借鉴缺陷修复思想制导漏洞自动修复在理论上是可行的, 但是存在较多技术挑战需要在未来研究工作中进行解决. 此外, 现有的程序缺陷自动修复技术主要依赖于可执行单元测试用例来定位缺陷和缺陷补丁的验证; 不同于一般程序缺陷, 漏洞的环境配置与测试用例编写难度更大[72], 很难通过单元测试来定位漏洞, 这对将现有程序缺陷自动修复技术嫁接到漏洞修复上提出了一大挑战.

4.3 历史驱动

相较于动静态分析, 以历史驱动作为技术原理的研究方法主要是基于历史漏洞代码与当前漏洞代码的相似度进行设计与实现. 早期基于历史驱动的漏洞修复工作是 Ma 等人[25]提出的基于模版的方法: Vurle. Vurle 学习已有的漏洞修复样本, 使用代码变更区分工具 GumTree[73]从漏洞代码和对应的修复代码的抽象语法树 (AST) 中抽取变更差异, 形成 AST 层面的编辑操作 (edit), 这些编辑操作通过聚类来产生相应的修复模板. 最后, Vurle 标识出最佳匹配的修复模板应用于补丁的生成. Vurle 是一个早期的基于语法树提取变更的工作, 使用 279 个已知漏洞来生成修复模板库, 作为早期的通用类型漏洞修复工作, 该方法主要通过聚类和人工总结来挖掘修复模型, 效率较低, 可扩展性较差. 近年来, 随着深度学习技术的快速发展, 基于深度学习的文本到文本的学习模型层出不穷. 通过将漏洞自动修复任务视为代码到代码的翻译任务, 研究使用基于深度学习的文本到文本的学习模型技术探索高效的修复模型挖掘和漏洞修复方案. 表 9 列举了现有修复通用类型漏洞的代表性工作. Harer 等人[26]提出一个使用对抗神经网络 (generative adversarial network, GAN)[74]的漏洞修复框架. GAN 由一个生成器 (generative model) 和一个分类判别器 (discriminative model) 构成, 生成器是一个神经网络机器翻译 (neural machine translation, NMT) 模型, 用判别器的反馈梯度替代 NMT 典型的负似然损失以进行训练; 分类判别器是一个区分 NMT 生成的修复代码和所需的实际修复代码的神经网络模型, 它的损失 (loss) 代表生成代码和实际修复代码之间差异的分布, 其分类结果反馈给生成器以形成反馈对抗的结构. 该工作使用了包含 117 738 个本地合成的漏洞程序作为训练与预测数

据集, 实验评估显示生成补丁的 BLEU-4 能够达到 80%, 然而由于数据集并非真实漏洞程序, 该方法的实验验证完全使用实验室中人为制造的数据集而非业界真实数据, 因此难以评估该方法的实际修复效果. Chi 等人[27]提出了使用 Transformer 的编码解码 (encoder-decoder)[75]框架修复漏洞程序的技术, 并引入了注意力机制以缓解梯度消失问题. Transformer 模型目前在多个自然语言任务中取得了性能最佳的实验结果, 然而将 Transformer 模型直接应用到漏洞修复任务的效果并不理想. 因此, 该工作进一步设计代码复制机制以提高补丁生成能力. 数据集方面该工作从一个包含 624 个漏洞的数据集[76]提取方法级别的代码变更, 然后隐去漏洞和补丁代码中的具体变量名, 减少词汇量, 以降低噪音数据的负面影响. 实验评估显示该方法的修复准确率为 23.3%, 修复能力较强; 但该方法依赖于训练数据的质量. Chen 等人[6]也提出了基于 Transformer 的漏洞修复方法, 相较于 Chi 等人的工作, 该方法针对 Chi 等人工作数据集过小的问题, 采用迁移学习的方式. 首先在包含 2 100 万缺陷补丁的数据集上进行预训练, 接着在包含 3 754 个漏洞的数据集上进行模型微调. 实验结果显示未经预训练的修复准确率仅为 12.58%, 预训练之后的修复准确率提高到了 17.3%, 这一结果部分揭示了利用迁移学习提高修复效果的可能性.

表 9 通用类型漏洞的修复方法的代表性工作概览

文献	输入依赖	学习模型	数据集漏洞修复实例数	缺点
Ma等人[25]	漏洞修复数据集	基于树的变更	279对漏洞修复程序	真实漏洞数据缺乏
Harer等人[26]		对抗神经网络GAN	117 738对合成的修复程序	
Chi等人[27]		Transformer	624对漏洞修复程序	
Chen等人[6]		Tranformer/迁移学习	3 754对漏洞修复程序	

整体而言, 以历史驱动作为技术原理的研究方法优点在于其修复模型的通用性, 相较于特定类型的漏洞修复方法, 无须明确特定的漏洞类型, 因此, 当前主流的通用类型漏洞修复技术大多以历史驱动作为技术原理. 另一方面, 基于历史驱动的研究工作主要依赖于历史漏洞修复数据、数据质量和可适用的模型选择, 然而现有的漏洞修复数据在数量和质量上都存在很大的限制, 且生成补丁的质量较低. 在未来的研究工作中, 除了解决数据问题, 需要考虑的是如何利用有限的数据开展可行的基于学习的漏洞修复方案研究, 例如利用小数据的学习达到高修复效果和有效收集; 还需要研究先进的补丁生成模型以提高生成补丁的质量.

4.4 混合式技术

考虑到不同分类的技术原理有其固有缺点和局限性, 研究者提出混合不同类型的技术原理以提高整体方法的修复能力. 混合式技术的设计要点在于如何合理地将不同类型技术原理的方法整合在一起. 例如静态分析器 Fortify[77]能发现缓冲区溢出静态警报, 针对静态警报误报率高的问题, Gao 等人[19]提出在静态分析部分采用可达性分析过滤掉与异常抛出语句等敏感代码位置不存在可达性的误报静态警报; 并采用动态符号执行技术遍历程序可行路径, 以过滤掉不存在可行路径的误报静态警报. 在补丁生成与验证阶段, 该方法利用模板对过滤后的静态警报生成补丁, 并基于符号执行对应的测试验证补丁的可行性. 通过混合式设计, 该方法基于动态符号执行减小静态分析误报过高的问题, 通过权衡静态分析的安全性提高了整体方法的可用性. Yang 等人[15]基于 AppScan[70]发现一组注入攻击的静态警报, 为了减小误报, 该方法针对存储类注入 (stored injection) 攻击分析非信任数据点 (entry) 到数据存储点的数据流 A 和数据存储点到非信任数据利用点 (exit) 的数据流 B. 若两条数据流中的存储点相同, 该方法在数据流图中连接 A.entry 与 B.exit 以形成一条完整数据流. 基于如上优化, 该方法过滤掉非完整的数据流以有效降低静态警报的误报率. 对于过滤后的警报, 该方法利用从历史漏洞修复数据中抽取出的修复模板指导补丁的生成, 以提高生成补丁的准确率. 整体而言, 混合式技术在修复漏洞工作上有较大的潜力, 而当前该类型研究相对较少.

5 漏洞修复方法的挑战与机遇

5.1 关键挑战

5.1.1 特定类型漏洞修复困难

近年来特定类型漏洞修复技术取得了一定的进展, 然而距离业界实用依然有很大差距. 如前文所述, 研究者针

对缓冲区溢出、整型溢出等特定类型漏洞提出了一些修复方法, 但修复效果仍然不足. 以注入攻击型漏洞为例, 现有的动态分析技术[45]难以构造出有效的程序输入, 而静态分析技术[15]由于静态分析精度不足使得修复结果存在大量误报.

5.1.2 业务强相关漏洞修复困难

现实复杂软件应用往往含有复杂业务逻辑, 对于单处代码的变更可能影响应用中的多处业务逻辑. 因此, 对业务逻辑复杂应用的代码进行漏洞修复可能导致业务正常逻辑受到影响, 甚至引入新的漏洞. 例如, Log4j2 维护者为漏洞 CVE-2021-44228 提交了 2.15 修复版本, 然而该修复引入了新的漏洞 CVE-2021-45046. 因此, 即便是代码维护者, 修复真实软件应用中的漏洞亦很困难, 如何有效地修复该类漏洞依然是漏洞自动修复研究者面临的首要挑战.

5.1.3 静态分析方法误报率过高

针对特定类型漏洞设计对应的静态分析算法的研究工作能够解决一些漏洞的自动修复问题. 然而由于莱斯定理的存在, 静态分析一定存在误报/漏报问题. 在漏洞根因定位这一任务中, 主要是存在误报率过高的问题. 根据 Gao 等人工作[47], 目前成熟商用静态检测工具 Fortify 检测缓冲区溢出漏洞的误报率超过 30%. 虽然现有工作[78−80]尝试缓解静态分析误报率过高的问题, 目前依然没有有效的解决方案.

5.1.4 基于动态分析的漏洞检测能力较弱

传统的单元测试/功能测试很难检测出安全漏洞问题, 一些特定类型消毒器可以作为辅助以暴露特定类型安全漏洞, 然而目前业界依然缺少针对通用类型漏洞的消毒器, 并且现有的动态漏洞检测工具往往依赖于插装程序/编译器, 会引入额外开销而提升漏洞修复成本, 例如常见的地址消毒器[30]会使整体运行时间增加约 73%, 内存使用量增加 240%. 此外, 动态漏洞检测方法检测漏洞的能力尚存不足, 以常见的模糊测试为例, Li 等人[67]研究显示, 模糊测试在检测漏洞程序方面依然存在代码覆盖率低、生成测试难以通过程序过滤/检查条件、高质量种子生成等问题.

5.1.5 漏洞修复数据集规模受限

由于漏洞修复过程复杂且有可能引入新的安全隐患, 软件开发者倾向于隐匿发布或者内部发布漏洞相关的数据, 例如 2021 年漏洞数据集 CVE 录入的漏洞数量仅为 20 141. 漏洞修复数据集规模受限导致漏洞修复模型训练不充分, 进而影响漏洞修复效果. 针对这一问题, Harer 等人[26]通过人工合成的方式生成大量漏洞程序, 并以此作为漏洞修复的训练数据集, 该方法虽然能够提升漏洞数据集规模, 然而漏洞数据的多样性、真实性和质量较差, 导致修复效果并不理想. Chen 等人[6]基于迁移学习在缺陷数据集上进行预训练, 然后在漏洞修复数据集上微调, 实验结果显示迁移学习对于缓解漏洞修复数据不足问题的效果有限. 综上所述, 有限的漏洞修复数据给漏洞自动修复的研究带来了另一大挑战.

5.2 研究机遇

5.2.1 基于模糊测试的漏洞利用自动生成的全栈式漏洞修复技术

现有基于动态分析的漏洞修复技术[18,24,45,51]均假设漏洞利用的存在, 然而这一假设对于现实漏洞修复场景难以成立, 一方面, 由于复现漏洞所需环境配置复杂, 另一方面, 由于人工构造漏洞利用难度大、成功率低; 这些挑战导致难以暴露现实场景中的漏洞并相应地对其根因进行定位. 为了实现符合现实场景的全栈式漏洞自动修复技术, 首先需要探索能够自动生成漏洞利用的技术 (即漏洞利用自动生成技术) 来暴露漏洞并进行定位. 在各类漏洞利用自动生成技术中, 模糊测试技术被证明是最有效检测零天 (0-day) 漏洞的技术, 基于模糊测试的漏洞检测工具如 AFL[81]已发现数万个真实软件中的漏洞. 因此, 具有应用前景的研究方向是探索基于模糊测试的漏洞利用自动生成的全栈式漏洞修复技术. 该研究方向需要重点探索如何将模糊测试报告的崩溃位置溯源至漏洞发生的根因位置, 即程序补丁位置 (以使能更高效地补丁生成). 例如 Blazytko 等人[82]提出了一种针对模糊测试的统计驱动的根因解释技术, 通过观察一组崩溃输入以及与之相似输入的行为差异进行根因定位, 这为探索基于模糊测试的漏洞利用自动生成的全栈式漏洞修复技术提供了一定的技术支撑, 该方向的研究具有可观的应用实践价值.

5.2.2 基于安全编码规范的漏洞自动修复方法

开发者通过参考安全编码规范以提高编码安全质量, 安全编码规范包含一组安全专家经验总结的编码规范, 然而现实场景中并非所有开发者严格遵循安全编码规范, 导致所开发的应用含有安全隐患. 因此, 一个可行方法是从高质量安全编码规范中抽取程序应满足的安全约束, 包括静态代码性质 (例如特定污点数据对于程序关键位置路径不可达) 或是动态程序行为约束 (例如对内存的读写操作应在缓存范围内) 等, 并相应设计补丁生成算法; 在补丁验证阶段, 仅有满足程序安全约束的候选补丁会被推荐给开发者. 该类方法需要解决一些重要问题, 例如如何保证安全编码规范本身的质量、如何从安全编码规范中抽取程序应满足的安全约束并进行代码层面的算法设计.

5.2.3 交互式漏洞修复方法

现有漏洞自动修复技术的目标是生成能够通过验证的修复补丁, 然而, 现有的技术无法保证那些能够通过验证的补丁的正确性. 此外, 对于位于业务逻辑复杂代码中的漏洞, 开发者人工修复亦十分困难, 甚至会引入新漏洞. 以 Log4j2 CVE-2021-44228 漏洞为例, 软件维护者首次提交的修复方案仅是部分修复, 且引入新的漏洞 CVE-2021-45046. 因此, 漏洞自动修复技术除了生成准确率更高的修复补丁之外, 还应关注如何辅助开发者更好地理解和接受修复补丁. 作为一种潜在的解决方案, 交互式漏洞修复技术可以综合自动修复技术的补丁生成能力以及应用开发者对于复杂业务的理解能力, 例如, Liang 等人[83]针对一般缺陷, 提出了一种两阶段的补丁过滤方法: 第 1 阶段为补丁准备阶段, 基于补丁特征 (如测试结果、执行路径) 分类程序自动修复技术生成的不同补丁; 第 2 阶段为补丁交互阶段, 开发者可以过滤补丁、选择补丁以及人工编写正确补丁. 实验结果显示在使用该方法辅助的条件下, 开发者在少花费 25.3% 时间的情况下多正确修复 62.5% 缺陷. 因此, 一个有潜力的研究方向在于针对漏洞程序, 设计自动漏洞修复技术与开发者交互的修复方法.

5.2.4 混合式漏洞修复方法

研究者们提出了若干混合式漏洞自动修复方法[15,19,22,23,47,49], 然而整体修复效果依然有限. 例如 Gao 等人[47]提出的动静态方法能有效降低静态分析的高误报率, 然而该方法使用符号执行技术使得方法整体扩展性不佳. 未来可能的优化方案在于利用静态分析的警报信息缩小符号执行覆盖的程序规模 (例如将分析范围限定在单一函数内或子模块内), 以缓解符号执行在较大规模程序上路径爆炸的问题. Yang 等人[15]提出的静态分析与历史驱动相结合的方法并不能很好解决静态分析误报率高的问题. 因此, 混合式漏洞修复方法在算法设计上依然有很大的改进空间. 混合式漏洞修复方法需要综合动态分析、静态分析、历史驱动修复以及交互式修复等各类技术的优点, 在效率与精度间取得权衡.

5.2.5 基于小样本学习的漏洞修复方法

随着深度学习的兴起, 研究者提出了基于学习的漏洞自动修复方法[6,25−27], 并取得了一定研究进展. 然而, 本文作者观察到当前基于学习的漏洞自动修复方法严重受限于可供训练数据不足的问题. 小样本学习 (few-shot learning)[84]是缓解该问题常用的学习算法, 而基于学习的漏洞自动修复任务因为数据的不足可以视为小样本学习的案例. 当前的小样本学习研究主要集中于图片、语音、自然语言等弱结构化的学习对象, 设计能够处理例如代码这类强结构化学习对象的小样本学习模型解决漏洞修复问题将会是未来的重要研究方向.

6 总 结

漏洞自动修复作为目前广泛关注的前沿技术, 对漏洞治理水平的提高有着极其重要的影响. 学术界对漏洞修复方法的研究有超过 20 年的历史, 很少有综述类文章全面总结漏洞修复方法的研究进展和成果. 本文从修复漏洞类型以及修复方法底层原理的角度梳理归纳现有漏洞修复研究, 总结了当前漏洞修复存在的关键问题和未来的发展趋势. 主要工作总结如下: (1) 归纳总结特定类型漏洞修复方法; (2) 归纳总结通用类型漏洞修复方法; (3) 分类不同漏洞修复方法的技术原理; (4) 最后指出漏洞修复领域当前面临的挑战与未来发展的方向.

References:

[1] CVE. 2022. https://www.cvedetails.com/vulnerabilities-by-types.php

[2] JNDI. 2022. https://docs.oracle.com/javase/jndi/tutorial/getStarted/overview/index.html
[3] Symatech: Symatech Internet security threat report. 2006. http://www.symantec.com
[4] Gu ZX, Barr ET, Hamilton DJ, Su ZD. Has the bug really been fixed? In: Proc. of the 32nd ACM/IEEE Int'l Conf. on Software Engineering. Cape Town: IEEE, 2010. 55–64. [doi: 10.1145/1806799.1806812]
[5] CWE. 2022. https://cwe.mitre.org/
[6] Chen ZM, Kommrusch S, Monperrus M. Neural transfer learning for repairing security vulnerabilities in C code. IEEE Trans. on Software Engineering, 2023, 49(1): 147–165. [doi: 10.1109/TSE.2022.3147265]
[7] Ye T, Zhang LM, Wang LZ, Li XD. An empirical study on detecting and fixing buffer overflow bugs. In: Proc. of the 2016 IEEE Int'l Conf. on Software Testing, Verification and Validation. Chicago: IEEE, 2016. 91–101. [doi: 10.1109/ICST.2016.21]
[8] Marchand-Melsom A, Nguyen Mai DB. Automatic repair of OWASP Top 10 security vulnerabilities: A survey. In: Proc. of the 42nd IEEE/ACM Int'l Conf. on Software Engineering Workshops. Seoul: ACM, 2020. 23–30. [doi: 10.1145/3387940.3392200]
[9] Owasp. 2022. https://owasp.org/
[10] Kechagia M, Mechtaev S, Sarro F, Harman M. Evaluating automatic program repair capabilities to repair API misuses. IEEE Trans. on Software Engineering, 2022, 48(7): 2658–2679. [doi: 10.1109/TSE.2021.3067156]
[11] Canfora G, Di Sorbo A, Forootani S, Martinez M, Visaggio CA. Patchworking: Exploring the code changes induced by vulnerability fixing activities. Information and Software Technology, 2022, 142: 106745. [doi: 10.1016/j.infsof.2021.106745]
[12] CNNVD. 2022 (in Chinese). https://www.cnnvd.org.cn/home/childHome
[13] Xuan JF, Ren ZL, Wang ZY, Xie XY, Jiang H. Progress on approaches to automatic program repair. Ruan Jian Xue Bao/Journal of Software, 2016, 27(4): 771−784 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4972.htm [doi: 10.13328/j.cnki.jos.004972]
[14] Jiang JJ, Chen JJ, Xiong YF. Survey of automatic program repair techniques. Ruan Jian Xue Bao/Journal of Software, 2021, 32(9): 2665−2690 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6274.htm [doi: 10.13328/j.cnki.jos.006274]
[15] Yang JQ, Tan L, Peyton J, Duer KA. Towards better utilizing static application security testing. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice. Montreal: IEEE, 2019. 51–60. [doi: 10.1109/ICSE-SEIP.2019.00014]
[16] Shariffdeen RS, Tan SH, Gao MY, Roychoudhury A. Automated patch transplantation. ACM Trans. on Software Engineering and Methodology, 2021, 30(1): 6. [doi: 10.1145/3412376]
[17] Avgerinos T, Cha SK, Rebert A, Schwartz EJ, Woo M, Brumley D. Automatic exploit generation. Communications of the ACM, 2014, 57(2): 74–84. [doi: 10.1145/2560217.2560219]
[18] Huang Z, Lie D, Tan G, Jaeger T. Using safety properties to generate vulnerability patches. In: Proc. of the 2019 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2019. 539–554. [doi: 10.1109/SP.2019.00071]
[19] Gao X, Wang B, Duck GJ, Ji RY, Xiong YF, Roychoudhury A. Beyond tests: Program vulnerability repair via crash constraint extraction. ACM Trans. on Software Engineering and Methodology, 2021, 30(2): 14. [doi: 10.1145/3418461]
[20] Shaw A, Doggett D, Hafiz M. Automatically fixing C buffer overflows using program transformations. In: Proc. of the 44th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks. Washington: IEEE, 2014. 124–135. [doi: 10.1109/DSN.2014.25]
[21] Long F, Sidiroglou-Douskos S, Kim D, Rinard M. Sound input filter generation for integer overflow errors. In: Proc. of the 41st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. San Diego: ACM, 2014. 439–452. [doi: 10.1145/2535838.2535888]
[22] Zhang Y, Kabir M, Xiao Y, Yao DF, Meng N. Data-driven vulnerability detection and repair in java code. arXiv:2102.06994, 2021.
[23] Sidiroglou-Douskos S, Lahtinen E, Rinard M. Automatic discovery and patching of buffer and integer overflow errors. Technical Report, MIT-CSAIL-TR-2015-018, MIT, 2015.
[24] Sidiroglou-Douskos S, Lahtinen E, Long F, Rinard M. Automatic error elimination by horizontal code transfer across multiple applications. In: Proc. of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Portland: ACM, 2015. 43–54. [doi: 10.1145/2737924.2737988]
[25] Ma SQ, Thung F, Lo D, Sun C, Deng RH. VuRLE: Automatic vulnerability detection and repair by learning from examples. In: Proc. of the 22nd European Symp. on Research in Computer Security. Oslo: Springer, 2017. 229–246. [doi: 10.1007/978-3-319-66399-9_13]
[26] Harer JA, Ozdemir O, Lazovich T, Reale CP, Russell RL, Kim LY, Chin P. Learning to repair software vulnerabilities with generative adversarial networks. In: Proc. of the 32nd Int'l Conf. on Neural Information Processing Systems. Montréal: Curran Associates Inc., 2018. 7944–7954.
[27] Chi JL, Qu Y, Liu T, Zheng QH, Yin H. SeqTrans: Automatic vulnerability fix via sequence to sequence learning. IEEE Trans. on Software Engineering, 2023, 49(2): 564–585. [doi: 10.1109/TSE.2022.3156637]

[28] Cheng X, Zhou M, Song XY, Gu M, Sun JG. Automatic fix for C integer errors by precision improvement. In: Proc. of the 40th IEEE Annual Computer Software and Applications Conf. Atlanta: IEEE, 2016. 2–11. [doi: 10.1109/COMPSAC.2016.70]

[29] Coker Z, Hafiz M. Program transformations to fix C integers. In: Proc. of the 35th Int'l Conf. on Software Engineering. San Francisco: IEEE, 2013. 792–801. [doi: 10.1109/ICSE.2013.6606625]

[30] Serebryany K, Bruening D, Potapenko A, Vyukov. AddressSanitizer: A fast address sanity checker. In: Proc. of the 2012 USENIX Conf. on Annual Technical Conf. Boston: USENIX Association, 2012. 28.

[31] Jha S, Gulwani S, Seshia SA, Tiwari A. Oracle-guided component-based program synthesis. In: Proc. of the 32nd ACM/IEEE Int'l Conf. on Software Engineering. Cape Town: IEEE, 2010. 215–224. [doi: 10.1145/1806799.1806833]

[32] Lee J, Hong S, Oh H. MemFix: Static analysis-based repair of memory deallocation errors for C. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 95–106. [doi: 10.1145/3236024.3236079]

[33] Exact Cover. Exact cover—Wikipedia, the free encyclopedia. 2018. https://en.wikipedia.org/wiki/Exact_cover

[34] Monperrus M. Automatic software repair: A bibliography. ACM Computing Surveys, 2019, 51(1): 17. [doi: 10.1145/3105906]

[35] Zhu QH, Sun ZY, Xiao YA, Zhang WJ, Yuan K, Xiong YF, Zhang L. A syntax-guided edit decoder for neural program repair. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and the Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 341–353. [doi: 10.1145/3468264.3468544]

[36] Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis. San Jose: ACM, 2014. 437–440. [doi: 10.1145/2610384.2628055]

[37] Xu ZZ, Zhang YL, Zheng LR, Xia LZ, Bao CF, Wang Z, Liu Y. Automatic hot patch generation for android kernels. In: Proc. of the 29th USENIX Conf. on Security Symp. Berkeley: USENIX Association, 2020. 135.

[38] Zhang XD, Zhu CG, Li Y, Guo JM, Liu LH, Gu HB. Precfix: Large-scale patch recommendation by mining defect-patch pairs. In: Proc. of the 42nd IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice. Seoul: IEEE, 2020. 41–50.

[39] Saha S, Saha RK, Prasad MR. Harnessing evolution for multi-hunk program repair. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 13–24. [doi: 10.1109/ICSE.2019.00020]

[40] Yuan Y, Banzhaf W. ARJA: Automated repair of java programs via multi-objective genetic programming. IEEE Trans. on Software Engineering, 2020, 46(10): 1040–1067. [doi: 10.1109/TSE.2018.2874648]

[41] Wong CP, Santiesteban P, Kästner C, Le Goues C. VarFix: Balancing edit expressiveness and search effectiveness in automated program repair. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 354–366. [doi: 10.1145/3468264.3468600]

[42] Xu TT, Chen LS, Pei Y, Zhang T, Pan MX, Furia CA. Restore: Retrospective fault localization enhancing automated program repair. IEEE Trans. on Software Engineering, 2022, 48(1): 309–326. [doi: 10.1109/TSE.2020.2987862]

[43] Jiang JJ, Xiong YF, Zhang HY, Gao Q, Chen XQ. Shaping program repair space with existing patches and similar code. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 298–309. [doi: 10.1145/3213846.3213871]

[44] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: Curran Associates Inc., 2017. 6000–6010.

[45] da Costa Meireles Barbosa JF. Automated repair of security vulnerabilities using coverage-guided fuzzing [MS. Thesis]. Porto: University of Porto, 2021.

[46] Hindle A, Barr ET, Gabel M, Su ZD, Devanbu P. On the naturalness of software. Communications of the ACM, 2016, 59(5): 122–131. [doi: 10.1145/2902362]

[47] Gao FJ, Wang Y, Wang LZ, Yang ZJ, Li XD. Automatic buffer overflow warning validation. Journal of Computer Science and Technology, 2020, 35(6): 1406–1427. [doi: 10.1007/s11390-020-0525-z]

[48] KLEE. 2022. https://klee.github.io/

[49] Mohammadi M, Chu B, Lipford HR. Automated repair of cross-site scripting vulnerabilities through unit testing. In: Proc. of the 2019 IEEE Int'l Symp. on Software Reliability Engineering Workshops. Berlin: IEEE, 2019. 370–377. [doi: 10.1109/ISSREW.2019.00098]

[50] Ma SQ, Lo D, Li T, Deng RH. CDRep: Automatic repair of cryptographic misuses in android applications. In: Proc. of the 11th ACM on Asia Conf. on Computer and Communications Security. Xi'an: ACM, 2016. 711–722. [doi: 10.1145/2897845.2897896]

[51] Wang TL, Song CY, Lee W. Diagnosis and emergency patch generation for integer overflow exploits. In: Proc. of the 2014 Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment. Egham: Springer, 2014. 255–275. [doi: 10.1007/978-3-319-08509-8_14]

[52] Muntean P, Monperrus M, Sun H, Grossklags J, Eckert C. IntRepair: Informed repairing of integer overflows. IEEE Trans. on Software Engineering, 2021, 47(10): 2225–2241. [doi: 10.1109/TSE.2019.2946148]

[53] Condit J, Harren M, McPeak S, Necula GC, Weimer W. CCured in the real world. ACM SIGPLAN Notices, 2003, 38(5): 232–244. [doi: 10.1145/780822.781157]

[54] Jim T, Morrisett JG, Grossman D, Hicks MW, Cheney J, Wang YL. Cyclone: A safe dialect of C. In: Proc. of the 2002 General Track: USENIX Annual Technical Conf. Monterey: USENIX. 2002. 275–288.

[55] 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. New Orleans: IEEE, 2000. 257–269. [doi: 10.1109/ACSAC.2000.898880]

[56] Wagner DA, Foster JS, Brewer EA, Aiken A. A first step towards automated detection of buffer overrun vulnerabilities. In: Proc. of the 2000 Network and Distributed System Security Symp. San Diego, 2000. 1–15.

[57] Evans D, Larochelle D. Improving security using extensible lightweight static analysis. IEEE Software, 2002, 19(1): 42–51. [doi: 10.1109/52.976940]

[58] Xie YC, Chou A, Engler D. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In: Proc. of the 9th European Software Engineering Conf. Held Jointly with the 11th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Helsinki: ACM, 2003. 327–336. [doi: 10.1145/940071.940115]

[59] Le W, Soffa ML. Marple: A demand-driven path-sensitive buffer overflow detector. In: Proc. of the 16th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Atlanta: ACM, 2008. 272–282. [doi: 10.1145/1453101.1453137]

[60] Infer. 2022. https://fbinfer.com/

[61] 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]

[62] Cowan C. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proc. of the 7th USENIX Security Symp. San Antonio: USENIX Association, 1998. 63–78.

[63] Jones RWM, Kelly PHJ. Backwards-compatible bounds checking for arrays and pointers in C programs. In: Proc. of the 3rd Int'l Workshop on Automated Debugging. Linköping: Linköping University Electronic Press, 1997. 13–26.

[64] Wagner D, Dean R. Intrusion detection via static analysis. In: Proc. of the 2001 IEEE Symp. on Security and Privacy. Oakland: IEEE, 2000. 156–168. [doi: 10.1109/SECPRI.2001.924296]

[65] Haugh E, Bishop M. Testing C programs for buffer overflow vulnerabilities. In: Proc. of the 2003 Network and Distributed System Security Symp. San Diego, 2003. 1–8.

[66] Xu RG, Godefroid P, Majumdar R. Testing for buffer overflows with length abstraction. In: Proc. of the 2008 Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2008. 27–38. [doi: 10.1145/1390630.1390636]

[67] Li J, Zhao BD, Zhang C. Fuzzing: A survey. Cybersecurity, 2018, 1(1): 6. [doi: 10.1186/s42400-018-0002-y]

[68] CWE-77: Improper neutralization of special elements used in a command (Command Injection). 2022. https://cwe.mitre.org/data/definitions/77.html

[69] Nguyen TD, Pham LH, Sun J. SGUARD: Towards fixing vulnerable smart contracts automatically. In: Proc. of the 2021 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2021. 1215–1229. [doi: 10.1109/SP40001.2021.00057]

[70] IBM. AppScan source. 2017. https://www.ibm.com/usen/marketplace/ibm-appscan-source

[71] UndefinedBehaviorSanitizer. 2022. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[72] Hu H, Chua ZL, Adrian S, Saxena P, Liang ZK. Automatic generation of data-oriented exploits. In: Proc. of the 24th USENIX Conf. on Security Symp. Washington: USENIX Association, 2015. 177–192.

[73] Falleri JR, Morandat F, Blanc X, Martinez M, Monperrus M. Fine-grained and accurate source code differencing. In: Proc. of the 29th ACM/IEEE Int'l Conf. on Automated Software Engineering. Vasteras: ACM, 2014. 313–324. [doi: 10.1145/2642937.2642982]

[74] Goodfellow IJ, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D. Generative adversarial nets. In: Proc. of the 27th Int'l Conf. on Neural Information Processing Systems. Montreal: MIT Press, 2014. 2672–2680.

[75] Cho K, van Merriënboer B, Bahdanau D, Bengio Y. On the properties of neural machine translation: Encoder-decoder approaches. In: Proc. of the 8th Workshop on Syntax, Semantics and Structure in Statistical Translation. Doha: ACL, 2014. 103–111. [doi: 10.3115/v1/W14-4012]

[76] Ponta SE, Plate H, Sabetta A, Bezzi M, Dangremont C. A manually-curated dataset of fixes to vulnerabilities of open-source software. In: Proc. of the 16th IEEE/ACM Int'l Conf. on Mining Software Repositories. Montreal: IEEE, 2019. 383–387. [doi: 10.1109/MSR.2019.00064]

[77] Fortify. 2022. https://www.joinfortify.com

[78] Tripp O, Guarnieri S, Pistoia M, Aravkin A. ALETHEIA: Improving the usability of static security analysis. In: Proc. of the 2014 ACM SIGSAC Conf. on Computer and Communications Security. Scottsdale: ACM, 2014. 762–774. [doi: 10.1145/2660267.2660339]

[79] Hanam Q, Tan L, Holmes R, Lam P. Finding patterns in static analysis alerts: Improving actionable alert ranking. In: Proc. of the 11th Working Conf. on Mining Software Repositories. Hyderabad: ACM, 2014. 152–161. [doi: 10.1145/2597073.2597100]

[80] Ruthruff JR, Penix J, Morgenthaler JD, Elbaum S, Rothermel G. Predicting accurate and actionable static analysis warnings: An experimental approach. In: Proc. of the 30th Int'l Conf. on Software Engineering. Leipzig: ACM, 2008. 341–350. [doi: 10.1145/1368088.1368135]

[81] Google AFL. 2022. https://github.com/google/AFL

[82] Blazytko T, Schlögel M, Aschermann C, Abbasi A, Frank J, Wörner S, Holz T. AURORA: Statistical crash analysis for automated root cause explanation. In: Proc. of the 29th USENIX Conf. on Security Symp. Berkeley: USENIX Association, 2020. 14.

[83] Liang JJ, Ji RY, Jiang JJ, Zhou SR, Lou YL, Xiong YF, Huang G. Interactive patch filtering as debugging aid. In: Proc. of the 2021 IEEE Int'l Conf. on Software Maintenance and Evolution. Luxembourg: IEEE, 2021. 239–250. [doi: 10.1109/ICSME52107.2021.00028]

[84] Wang YQ, Yao QM, Kwok JT, Ni LM. Generalizing from a few examples: A survey on few-shot learning. ACM Computing Surveys, 2021, 53(3): 63. [doi: 10.1145/3386252]

附中文参考文献:

[12] CNNVD漏洞分级规范. 2022. https://www.cnnvd.org.cn/home/childHome

[13] 玄跻峰, 任志磊, 王子元, 谢晓园, 江贺. 自动程序修复方法研究进展. 软件学报, 2016, 27(4): 771–784. http://www.jos.org.cn/1000-9825/4972.htm [doi: 10.13328/j.cnki.jos.004972]

[14] 姜佳君, 陈俊洁, 熊英飞. 软件缺陷自动修复技术综述. 软件学报, 2021, 32(9): 2665–2690. http://www.jos.org.cn/1000-9825/6274.htm [doi: 10.13328/j.cnki.jos.006274]

徐同同(1993－), 男, 博士, CCF 专业会员, 主要研究领域为程序分析与测试, 程序自动修复, 漏洞分析.

夏鑫(1986－), 男, 博士, CCF 专业会员, 主要研究领域为软件仓库挖掘, 经验软件工程.

刘逵(1988－), 男, 博士, CCF 专业会员, 主要研究领域为智能软件工程, 程序自动修复与缺陷定位, 经验软件工程.