软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.007205]

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

基于思维链的软件漏洞自动修复技术研究*

林 博, 王尚文, 毛晓光

(国防科技大学 计算机学院, 湖南 长沙 410073)

通信作者: 毛晓光, E-mail: xgmao@nudt.edu.cn

摘 要: 随着软件漏洞的类型、数量和复杂性日渐增长, 研究人员提出了诸多自动化的手段来帮助开发人员发现、检测和定位漏洞, 但研究人员仍需花费大量精力对漏洞进行修复. 近年来, 一些研究者开始关注软件漏洞自动修复技术, 然而当前的先进技术仅仅将软件漏洞修复规约为通用的文本生成问题, 没有对缺陷修复位置进行定位, 导致修复程序的生成空间较大, 使得生成的修复程序质量较低, 将其提供给开发人员反而影响漏洞修复的效率和效果. 针对上述问题, 本文提出了一种基于思维链的通用类型漏洞修复方法 CotRepair, 利用思维链技术, 模型首先对产生漏洞概率较高的位置进行预测, 而后依托预测结果, 更加准确地生成修复程序. 实验结果表明本文提出的方法在评价生成修复程序的各项指标上均显著优于基线方法, 从多个维度验证了所提方法的有效性.

关键词: 软件漏洞; 缺陷自动修复; 深度学习

中图法分类号: TP311

中文引用格式: 林博, 王尚文, 毛晓光. 基于思维链的软件漏洞自动修复技术研究. 软件学报. http://www.jos.org.cn/1000-9825/7205.htm

英文引用格式: Lin B, Wang SW, Mao XG. Automated Software Vulnerability Repair Based on Chain-of-thought. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7205.htm

Automated Software Vulnerability Repair Based on Chain-of-thought

LIN Bo, WANG Shang-Wen, MAO Xiao-Guang

(College of Computer Science, National University of Defense Technology, Changsha 410073, China)

Abstract: As software vulnerabilities grow in type, volume, and complexity, researchers have proposed various techniques to help developers discover, detect, and localize vulnerabilities. However, researchers still need to exert considerable effort to manually repair these vulnerabilities. In recent years, some researchers have focused on automated software vulnerability repair. However, such a task is merely considered a generic text generation problem by the current advanced technology, and the detects are not located. As a result, the generation space of the repair program is large, and the generated repair program is low-quality. Providing developers with such low-quality repairs affects the efficiency and effectiveness of vulnerability repair. To solve the above problems, a general type vulnerability repair approach based on chain-of-thought is proposed in this study, which is named CotRepair. By utilizing the chain-of-thought technology, the model first predicts the locations that are most likely to contain vulnerable code, and then generates the repair program more accurately based on the predicted locations. The experimental results show that CotRepair outperforms the baselines in various metrics, and the effectiveness of the proposed approach is demonstrated from multiple aspects.

Key words: software vulnerability; automated vulnerability repair; deep learning

软件漏洞 (Software Vulnerability, 以下简称为漏洞) 通常是指软件系统中可以被攻击者利用以执行未经授权的行为, 从而导致安全问题的缺陷或错误. 发现和修复漏洞对于维护应用和系统的安全至关重要, 软件厂商会定期发布补丁来修复已知的漏洞, 用户也需要定期更新补丁来防范利用这些漏洞的攻击. 传统上通常由软件开发厂商通过手工修复形成修复程序, 该方式存在较多局限性, 如时效性差和正确性难以保证等问题. 例如, 国际知名软件安全公司 Veracode 发布的 2021 年软件安全报告[1]中显示, 四分之一的高危和严重缺陷在发现之日起 290 天内仍

* 收稿时间: 2023-11-27; 修改时间: 2024-02-23; 采用时间: 2024-04-12; jos 在线出版时间: 2024-06-20

未被修复. 因此, 漏洞自动修复技术具有较大的研究价值, 在近年来得到了广泛关注[2−5].

当前, 不同漏洞产生的根因及相适应的修复机制通常存在较大区别, 为了更好地理解并修复漏洞, 研究人员依据漏洞的内在特征对其进行分类, 形成了不同的漏洞类型. 目前, 国际上主流的漏洞分类系统为 CWE(Common Weakness Enumeration)[6], 截止本文撰写时 (2023.03), 该系统收录了 933 类漏洞, 本文的漏洞分类也依托于此. 由于漏洞类型过于繁杂, 研究人员难以对每类漏洞的发生根因及相应的修复机制进行逐一分析[7], 而通用类型漏洞的修复方法不限定漏洞类型, 因而具有较高的研究价值. 当前主流的通用类型漏洞的修复方法为基于历史驱动的方法, 聚焦于如何将漏洞修复规约为通用的文本生成, 即以漏洞程序作为输入, 将其翻译为修复程序, 进而基于历史漏洞修复数据集, 利用学习算法训练模型以生成漏洞的修复程序. 该技术路径应用范围广泛, 可运用于所有类型的漏洞, 且可操作性强, 无需配置完整的运行环境, 因此除部分基于约束求解的方法[4,5]之外, 主流方法大多基于历史驱动的技术进行漏洞修复[7]. 然而, 当前的漏洞数量较少, 最大规模的漏洞修复数据集仅包含 5 365 对漏洞程序/修复程序[8], 且未经人工筛选, 质量较低, 而历史驱动的修复技术极其依赖历史漏洞数据的数量及质量, 导致当前修复程序生成的质量较低. 因此, 如何充分利用当前有限的漏洞数据生成高质量的修复程序, 成为提升通用类型漏洞修复技术有效性的关键.

Chen[3]等人提出了基于迁移学习的漏洞自动修复技术 VRepair, 该技术首先在 65 万对 C 语言缺陷程序/修复程序上进行预训练, 而后在漏洞数据集上进行微调, 以缓解漏洞数据有限的问题. 然而, VRepair 的预训练方法较为朴素, 仅仅将其作为一个机器翻译任务进行预训练, 而现有的技术证明, 通常需要多样化的预训练技术才能够更好地捕获到足够的代码语义信息[9−12]. 针对这个问题, Fu 等人[2]提出了基于 T5 预训练模型的漏洞修复技术 VulRepair, 该技术利用在 835 万个代码实例上以 4 种任务预训练而得到的 T5 模型作为基座, 在漏洞数据集上进行微调, 缓解了 VRepair 代码语义捕获能力较差的问题. 综上, 目前的先进技术都在尝试更好地捕获漏洞程序本身的语义信息以提升补丁生成的质量, 但却忽略了补丁生成时的搜索空间对补丁质量的影响. 具体而言, 当前的技术在进行漏洞修复时, 仅仅尝试将含有缺陷的代码直接翻译为修复程序; 在没有修复位置信息的情况下, 漏洞程序中的所有元素均有可能被模型识别为缺陷元素, 导致补丁生成的搜索空间较大, 进而影响补丁的生成质量. 通过对比 VulRepair 方法生成的修复程序与开发人员提供的修复程序, 我们发现, VulRepair 生成的不正确修复程序中, 84.0% 没有在正确的修复位置进行修改.

图 1(a) 展示了编号为 CVE-2017-14341 漏洞实例的官方修复补丁. 该实例属于不受控的资源消耗缺陷 (CWE-400), 该类缺陷通常是由于软件没有正确限制请求的数量或是申请的资源大小, 因而导致消耗了比预期更多的计算资源, 如 CPU 和内存等. 具体而言, 在该实例中, 初始代码没有对读入的图像实例 image 的实际大小与其在文件头声明的图像大小做校验; 因此, 当攻击者在传入的图像实例 image 中声明一个系统难以承受的大小时, 软件会以传入的图像实例所声明的大小对图像进行读取并处理, 导致系统资源被错误地耗尽. 为了避免产生不受控的资源消耗, 该补丁增加了对声明的图像大小 Rec.RecordLength 与实际图像大小 GetBlobSize(image) 的判断, 当图像声明的大小大于其实际大小, 抛出异常. 图 1(b) 展示了当前最先进技术 VulRepair 针对该漏洞所产生的 Top-1 补丁, 该漏洞的精确定位信息所提供的缺陷点位为第二行结束之后 (即 Rd_WP_DWORD(image, &Rec.RecordLength);), 新一行开始之前 (即 if (EOFBlob(image)) break;). 由于该修复方法缺少精确的修复位置信息, VulRepair 选择了修改函数 Rd_WP_DWORD 的调用内容, 而正确的缺陷点位在函数 Rd_WP_DWORD 调用结束之后, 通过增加声明的图像大小与实际图像大小 GetBlobSize(image) 的比较进行修复. 尽管 VulRepair 在修复的过程中, 于 Top-15 产生了该正确的补丁, 但由于缺少精确的修复位置信息导致修复工具的搜索空间较大, 并且对漏洞程序中的非修复位置进行了修复尝试导致生成了大量不正确的修复程序. 换而言之, 如果模型能够正确预测到缺陷的修复位置在于 Rd_WP_DWORD 的函数调用结束之后, 则不会在错误的位置 (如 Rd_WP_DWORD 的函数调用) 上生成修复程序, 因而有更大的概率能够生成正确的修复程序. 与此同时, 由于当前大量漏洞缺少复现所需的运行环境及测试用例, 因此需要研究人员复现并编写对应的测试用例或是人工对补丁正确性进行验证, 人工耗费极大, 难以大规模批量复现缺陷以验证补丁效果. 综上所述, 缩减修复工具生成修复程序的生成空间对提升漏洞自动修复技术的实用性具有较大的研究价值.

```
1.       if (i == EOF) break;
2.       Rd_WP_DWORD(image, &Rec.RecordLength);
3.  +    if (Rec.RecordLength > GetBlobSize(image))
    +          ThrowReaderException(CorruptImageError, "ImproperImageHeader");
         if (EOFBlob(image)) break;
         Header.DataOffset = TellBlob(image) + Rec.RecordLength;
```

(a) 正确补丁

```
1.       if (i == EOF) break;
2.  -    Rd_WP_DWORD(image, &Rec.RecordLength);
3.  +    Rd_WP_DWORD(0, sizeof(WPG_Record));
         if (EOFBlob(image)) break;
         Header.DataOffset = TellBlob(image) + Rec.RecordLength;
```

(b) VulRepair产生的Top-1补丁

图 1 CVE-2017-14341 漏洞的补丁

针对上述问题, 本文提出一种基于思维链的通用类型漏洞修复方法 CotRepair, 将漏洞的修复过程分解为精确修复位置推理与补丁生成, 这种模仿人类推理过程的学习方式被称为思维链, 已被证明能够增强模型对复杂问题的求解能力[13,14]. 我们将在 2.1 节对此类推理方式进行详细介绍. CotRepair 首先对概率较高的修复位置进行预测, 而后优先在这些位置对可能的修复空间进行搜索以生成修复程序. 相比于传统方法, 如 VRepair 和 VulRepair, 仅尝试依托行级别的缺陷位置标记对漏洞程序进行修复, 在细粒度修复位置预测结果的约束下, CotRepair 的修复程序搜索空间更加精确. 具体而言, 图 2 展示了 CVE-2018-879 漏洞的官方补丁以及不同标记方式. 其中, 红色方框标记了传统行级别缺陷位置标记信息, 绿色方框标记了 CotRepair 所预测并用以修复漏洞的修复位置标记. 传统行级别缺陷位置标记信息标记了整行缺陷代码; 而 CotRepair 所预测的修复位置, 标记了具体的修复位置是在 while 的条件表达式内, collen 之前. 该修复位置标记相较于行级别的缺陷位置标记更加精确地标识了代码需要更改的位置. 因此, 能够在一定程度上提高修复程序生成的准确率. 本文方法的工作流程如图 3 所示, 其总体上分为三个阶段, 分别为数据预处理阶段、模型训练阶段和漏洞修复阶段. 在数据预处理阶段, 针对历史漏洞数据进行处理, 得到漏洞程序、修复位置以及修复程序; 在模型训练阶段, 以漏洞程序作为模型的输入, 构建{修复位置+修复程序}作为预期输出, 以期模型能够学会先通过定位修复位置再生成修复程序, 达到运用思维链的方式对漏洞进行修复的目的; 当应用于漏洞修复时, CotRepair 以漏洞程序作为输入, 通过集束搜索 (beam search) 生成候选修复程序以供开发者参考. 多组对比实验验证了本文所提出的方法通用类型漏洞修复上的有效性.

```
1.  collen = 0;
2.  }
3.  -    while ( collen > 0) {
4.  +    while (indexw < width && collen > 0) {
5.       color = CVAL( in ); * out = color;
6.       out += 4;
```

行级别缺陷位置标记
修复位置标记

图 2 CVE-2018-879 漏洞补丁以及不同粒度标记方式

本文第 1 节介绍漏洞修复的相关方法和研究现状. 第 2 节介绍本文所需的背景知识. 第 3 节介绍本文构建的基于思维链的漏洞自动修复模型. 第 4 节通过对比实验验证了所提模型的有效性. 第 5 节讨论了方法的有效性与效率. 最后第 6 节对全文进行了总结.

1 漏洞修复相关工作

本文所涉及内容主要与漏洞修复有关, 下面就相关研究现状进行介绍.

当前的漏洞修复方法, 根据是否针对特定类型漏洞, 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, 特定类型的漏洞修复方法通常依据特定类型漏洞的共性根因和修复机制定制专有的漏洞定位与补丁生成算法. 通用类型方法不指定漏洞类型, 主流方法依据漏洞的历史数据, 基于学习算法对漏洞修复的过程进行学习, 以实现对任意类型漏洞的修复. 对于特定类型漏洞修复的方法, 当前研究主要针对综合影响力较大的漏洞开展, 例如缓冲区溢出和整形溢出等常见且危害性较大的漏洞. 例如, Sidiroglou 等人[15]提出了 Code Phage 技术, 该技术假设非漏洞程序中含有漏洞程序缺失的分支检查, 缺失的分支检查将导致缓冲区溢出漏洞. 依托这一假设, 该技术从健康程序中搜索能够正确处理失败用例的程序, 从中提取漏洞程序中缺失的分支检查, 后将分支条件修改移植至漏洞程序中以实现修复. 然而, 该技术存在假设过于理想, 实际难以找到合格的非漏洞程序作为供体等问题[16]. 为了提高修复程序的生成质量, Huang 等人[17]采用程序合成的方式生成漏洞修复程序. 具体而言, 其针对缓冲区溢出漏洞和整数溢出漏洞, 首先定义一组安全约束, 进而利用符号执行检测表达式是否满足安全约束, 对于不满足约束的表达式, 利用一组预先定义的修复模板生成修复程序. 此外, 除了在漏洞产生之后进行修复, 一些研究者也试图从底层预防缓冲区溢出漏洞的产生. 例如, Gao 等人[18]基于 OpenRefactory/C[19]对 C 程序进行变换, 在代码层面基于控制流、数据流及别名分析等程序分析手段确保缓冲区的安全性, 进而在不影响程序语义的前提下, 将不安全 API 和字符指针替换为安全 API 和数据结构. 一些研究工作如 CCured[20]和 Cyclone[21]也采用了类似的思路扩展了 C 语言, 实现了对缓冲区溢出漏洞的预防.

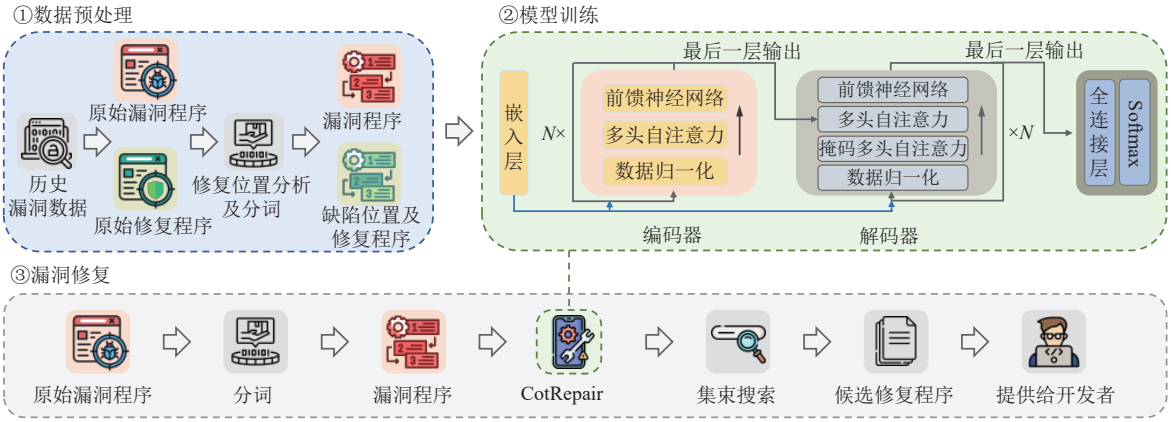

图 3 CotRepair 技术工作流程图

对于通用类型漏洞的修复方法而言, 一些研究者提出了基于约束求解的解决方案. Gao 等人[5]提出了 ExtractFix, 该方法在编译阶段插装安全约束检查, 对于违反约束的程序位置利用反向数据流分析定位可能的漏洞根因位置, 而后利用反例制导的程序合成技术[22]生成满足安全约束的修复补丁. Zhang 等人[4]提出了基于反例指导归纳推理的漏洞修复方法 VulnFix, 该方法通过反例推导可能修复的位置的不变式, 基于不变式结合预定义模板生成修复程序. 需要注意的是, ExtractFix 和 VulnFix 不指定缺陷类型, 理论上能够对约束可抽取的任意类型漏洞进行修复. 但这类方法仍然需要反例触发程序的异常状态 (如程序终止或崩溃), 以实现约束抽取及求解. 这类方法在如内存缓冲区限制不当 (CWE-119)、越界读取 (CWE-125) 等约束明晰、易抽取的漏洞类型上有较好的表现. 但对于部分漏洞类型, 如输入验证不当 (CWE-20)、敏感信息未经授权泄露 (CWE-200) 等, 由于这些类型漏洞的反例可能不会触发程序的异常状态或约束难以提取, 其性能较差. 因此, 目前通用类型漏洞修复方法主要依托待修复漏洞与历史漏洞存在的相似性, 进而从历史数据中学习到修复模板或是修复行为. Ma 等人[23]于 2017 年提出了基于抽象语法树 (Abstract Syntax Tree, AST) 级修复模板的通用类型漏洞修复方法 VuRLE, 该方法通过代码变更生成工具 GumTree[24]从历史的漏洞代码和其修复代码数据中提取 AST 级别的编辑操作 (Edit), 而后将提取到的编辑操作利用聚类算法进行聚类以获取修复模板. 最后, VuRLE 比较输入代码的 AST 和修复模板中待修复部分的 AST 之间的相似度, 标识相似度最高的修复模板用以补丁生成. 然而, 该工作存在修复模板种类较少, 可扩展性较低的问题. 为了缓解这个问题, HARER 等人[25]提出了使用对抗生成网络 (generative adversarial networks, GAN)[26]

的漏洞修复技术. 该技术主要由一个生成器 (Generative Model) 和判别器 (Discriminative Model) 组成, 生成器用以生成修复代码, 判别器用以判别输入的代码是生成代码还是实际修复代码, 优化目标为使生成器生成的代码接近实际修复代码. 然而, 由于缺少大规模的漏洞数据集, 该工作使用了包含本地合成的漏洞程序作为训练与预训练集, 难以评估该方法实际的修复效果. 为了缓解漏洞修复数据集较小的问题, Chen 等人[3]提出了基于迁移学习的漏洞修复方法 VReapir, 该工作利用 Transformer 模型, 首先在 C 语言缺陷修复数据集上进行预训练, 而后在包含 3 754 个漏洞的数据集上进行模型微调, 评估结果显示, 相比于直接在漏洞数据集上训练, 预训练之后的修复准确率从 12.58% 提升至 17.30%. 尽管如此, VReapir 仍然存在预训练任务单一, 导致代码语义捕获能力差的问题. 为了缓解上述问题, Fu 等人[2]提出了基于 T5 预训练模型[27]的漏洞修复方法, 该方法利用了在 835 万个代码实例上以 4 种任务预训练而得到的 T5 模型作为基座, 在 8 482 个漏洞数据上进行微调, 缓解了 VRepair 代码语义捕获能力较差的问题. 评估结果显示, 相比于 VReapir 方法, VulRepair 的修复准确率从 23% 提升到了 44%, 该结果揭示了利用预训练模型所蕴含的丰富信息提升修复效果的可能性. 尽管如此, 当前的通用类型漏洞修复方法仅仅试图将含有缺陷的代码直接翻译为修复程序; 在没有精确定位信息的情况下, 整个漏洞程序中的所有元素均有可能为缺陷元素, 导致修复程序生成的搜索空间较大, 进而影响修复程序的生成质量. 本文针对漏洞程序中可能的缺陷元素进行预测, 以期模型能够学会先通过定位修复位置再生成修复程序, 从而对修复程序生成的搜索空间进行精化, 以提升修复程序的生成质量.

2 背景知识

2.1 思维链

针对单纯扩大模型无法在算数推理, 符号推理等复杂任务上取得理想效果的问题, Wei 等人[14]提出了思维链 (Chain-of-Thought, CoT) 技术. 该技术将一系列中间推理步骤作为模型目标输出的一部分, 目的在于希望模型能够具备分步解决问题的推理能力. 图 4 展示了传统技术与基于思维链技术的对比. 例如, 对于数学问题而言, 传统的方法仅以一个数字作为模型的预期输出, 思维链技术将解决数学问题的中间步骤作为输出的一部分, 使得模型能够在一定程度上具备推理能力, 从而提升解决复杂问题的能力. 本文依据思维链的思想, 模拟人类对漏洞进行修复的过程, 将漏洞修复过程分解为修复位置定位和修复程序生成, 将漏洞修复分步骤解决, 对修复程序的搜索空间进行精化, 以提升修复程序生成的质量.

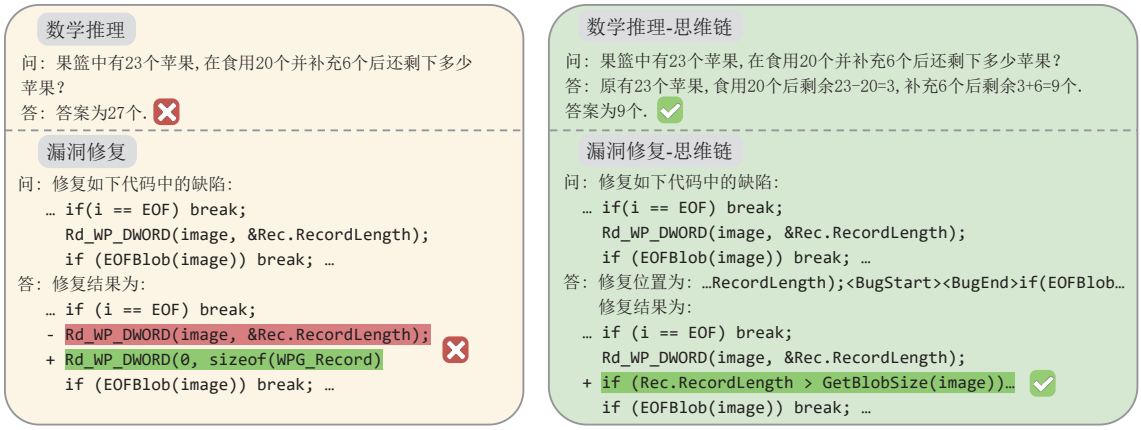

图 4 基于思维链技术的应用示例

2.2 编码器-解码器架构

编码器-解码器架构 (Encoder-Decoder Architecture) 是深度学习中最常见架构之一, 已被广泛应用于自然语言

和图像处理等任务[28−31]. 编码器通常负责将输入数据转换为一个隐藏状态向量. 隐藏状态向量包含了输入数据的高维抽象表示, 这一行为被称为“编码”. 解码器以隐藏状态向量作为输入, 输出目标数据的序列. 例如, 在机器翻译任务中, 解码器输出目标语言句子.

目前的编码器-解码器架构中使用最广泛的当属 Transformer 架构, 当前几乎所有先进的语言处理模型, 包括大语言模型, 均基于 Transformer 架构[29,30,32]. 具体而言, Transformer 由若干个编码器 (Encoder) 层和解码器 (Decoder) 层组成, 每个层均包括一个自注意力模块和一个前馈网络 (Feed-Forward Network, FFN) 模块. 在训练过程中, 模型将输入序列作为编码器的输入, 同时将目标输出序列作为解码器的输入, 之后根据目标输出序列的实际输出和预期输出之间的差异来计算损失并优化模型.

3 基于思维链的漏洞修复方法

本文提出一种基于思维链的漏洞修复方法, 该方法主要由数据预处理阶段、模型训练阶段和漏洞修复阶段组成.

数据预处理阶段, 目的在于对模型的输入输出进行构建, 包括漏洞程序、修复位置及修复程序. 本文首先针对漏洞数据进行预处理. 针对历史漏洞数据通过搜索其修复记录, 抽取出原始漏洞和修复程序, 在剔除漏洞程序和修复程序中的注释信息后, 对漏洞程序和修复程序进行分析, 标记漏洞程序的修改位置, 将修改位置及其上下文作为修复位置. 将剔除注释信息后的漏洞程序和修复程序利用 BPE[33]分词算法分别进行分词, 得到漏洞程序和修复程序. 数据预处理及表征方式于 3.1 节中进行详细介绍.

在模型训练阶段, 以漏洞程序作为模型的输入, 构建{修复位置+修复程序}的序列作为预期输出, 以减少预期输出和模型的实际输出之间的交叉熵作为优化目标.

在预测阶段, 对于给定的原始漏洞程序, 我们首先剔除其中所有的注释并利用 BPE 分词算法将其转化为漏洞程序作为模型的输入. 在得到输入后, 将其送入训练完毕的模型中, 利用集束搜索 (beam search) 技术对可能的输出进行搜索, 输出按照可能性从高到低排列, 即得到 CotRepair 产生的候选修复程序.

3.1 数据预处理与数据表示

数据预处理. 针对数据集中存在的一次漏洞修复提交记录, 通过搜索历史修复记录, 抽取出修改前与修改后代码, 即原始漏洞程序与修复程序. 而后利用 tree-sitter①去除原始漏洞程序与修复程序中的所有注释, 该操作的目的是剔除与代码无关的更新以避免在数据中引入噪声. 在去除所有注释后, 利用 tree-sitter 以剔除所有未产生更改的函数, 对产生变更的函数分别利用 BPE 分词算法进行分词, 得到漏洞程序与修复程序. 最后, 利用 difflib 工具②对两个序列进行比较并生成修复位置信息.

数据表示. 本文使用漏洞修复过程中所产生的一次提交记录作为数据集的基本构成单元. 模型的输入方面, 本文遵循先前基于历史驱动的漏洞自动修复方法[2,3], 以标记后的漏洞程序作为输入. 其中, 漏洞程序中使用两个特殊令牌<StartLoc> 和<EndLoc> 标记漏洞程序中的第一行缺陷代码. 对于出现在多行的缺陷, 也仅标记首行缺陷代码. 该操作的动机在于外部漏洞检测工具[34,35]或是安全专家通常会对部分可疑度较高的代码行进行标记以帮助开发人员对漏洞进行修复. 例如, 静态分析工具 Infer[36]仅对每个检测到的漏洞提供单个易受攻击的代码行作为缺陷位置的输出. 我们在 5.2 节中分析了该标记方式下对于多行缺陷的修复能力的影响. 需要注意的是, 标记的缺陷行并不等同于缺陷位置. 具体而言, 缺陷位置包含了漏洞程序中存在漏洞的所有代码行, 而标记的缺陷行为出现缺陷的首行代码.

由于标记的缺陷行和修复位置存在不一致性, 漏洞修复的过程可能不涉及到缺陷位置处代码的修改而在其他位置进行修改. 因此, 为了对修复位置进行表示, 本文构建了带上下文的修复位置表征方式. 图 5 展示了漏洞 CVE-

① https://tree-sitter.github.io/

② https://docs.python.org/3/library/difflib.html

2017-14341 的修复位置表示方式作为示例. 针对一次提交记录, 在对数据进行预处理后得到的漏洞程序与修复程序, 利用 diffllib 工具对两个序列进行比较, 获取漏洞程序中被更改的令牌, 而后将该令牌及其上下文, 利用预定义的令牌<BugStart><BugEnd> 进行标记并输出作为修复位置. 根据 Chen 等人的研究[3], 当上下文大小为 3 时, 足以唯一定位代码片段在代码中的位置, 因此本文对于缺陷的上下文的界定为缺陷令牌周围的 3 个令牌. 修复位置表示方式的局限性在 5.3 节中进行讨论.

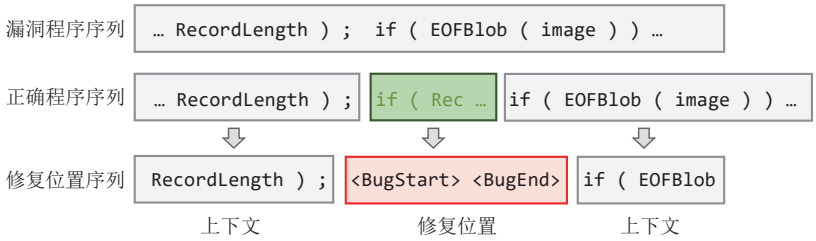

图 5 修复位置表示方式

对于修复结果的表示方面, 之前的先进通用类型漏洞修复方法中, 如 VRepair 与 VulRepair, 均选择输出被修改的代码令牌及其上下文以压缩模型的输出. 在模型能力较弱的情况下, 这种做法能够避免由于输出序列较长引起的能力下降. 然而, 预训练模型通过在大量各类代码上通过多种预训练任务已被证明能够对完整代码具有较强的输出能力[9−11,30]. 如果仍旧在预训练模型上以修改的令牌及其上下文作为预期输出进行微调, 会导致微调任务与预训练任务之间的距离过大, 这种差距会导致模型无法充分利用预训练所学习到的信息进而导致性能下降[37,38]. 因此, CotRepair 选择输出完整的程序作为修复结果的表示以充分利用预训练模型中的丰富信息. 两种方法的优劣将在 4.6 节中进行讨论.

代码分词: 对于输入的漏洞程序与修复程序, 首先采用 BPE(Byte Pair Encoding) 算法[33]对输入的漏洞程序进行分词. BPE 算法会将较长的代码令牌划分成更小但更常见的子令牌序列, 这些词根有助于构建更好的词嵌入模型和语言模型, 并已被广泛应用于语言模型中[9,27−29]. 例如, 对于代码令牌"IsValidValue"而言, BPE 会将该令牌划分为更常见的令牌序列, 即"IsValid"和"Value". 既缓解了词表外单词 (out of vocabulary, OOV) 的问题, 也使得模型能够对训练集中未出现的令牌具备理解能力. 经过该步骤得到漏洞程序与修复程序.

3.2 模型训练

本文的模型架构遵循 T5 模型, 为编码器-解码器架构. 解码器和编码器均由 12 层, 每层含 12 个注意力头的 Transformer 层组成. 该架构被应用于诸多任务并取得了最先进的效果[2,9,39−41]. 而后, 同当前的诸多工作一样[39,40,42], 我们利用 CodeT5 模型[9]的权重参数初始化 CotRepair 的参数, 目的在于赋予模型从预训练中学习到的领域知识. 在该阶段中, 模型的输入为数据预处理阶段产生的漏洞程序, 预期输出为{修复位置+修复程序}, 优化目标为减少预期输出和实际输出之间的交叉熵. 其中, 修复位置与修复程序之间以符号 [SEP] 进行连接, [SEP] 为用以分割两种数据类型的预定义符号.

嵌入层: 嵌入层由代码嵌入层和位置编码层两部分组成. 代码嵌入的目的在于将文本形式的代码映射至向量空间, 通过代码嵌入, 代码不再被视为由独立的符号组成的序列, 而是映射到稠密的, 具有实值的向量空间中[43]. 具体而言, 假定对于输入的漏洞程序 C, 经过分词得到令牌流序列 $(t_i,...,t_n)$ 后在词嵌入矩阵 E_w 中寻找第 i 个令牌对应的语义向量 x_i, 得到漏洞程序 C 的嵌入矩阵 $X=Concat(x_1,\ldots,x_n), x\in\mathbb{R}^{n\times d_x}$. 具体过程可表示为:

$$x_i = lookup(t_i, E_w)$$

对于位置编码层而言, 由于 Transformer 架构中不含有递归和卷积等能够捕获位置信息的结构, 因此需要额外构建对位置信息进行捕获的结构. 位置编码层用以捕获嵌入矩阵 X 中的任意两个元素 x_i 和 x_j 之间的相对位置关系, 其将 x_i 至 x_j 的相对位置关系编码为两个向量 $a_{ij}^K, a_{ij}^V \in \mathbb{R}^{d_a}$, 具体过程可表示为:

$$a_{ij}^{K} = w_{clip(j-i,k)}^{K}$$
$$a_{ij}^{V} = w_{clip(j-i,k)}^{V}$$
$$clip(x,k) = \max(-k, min(k,x))$$

其中, Clip 函数的作用为对距离过大的两个元素之间的相对位置进行裁剪, 即若两个元素之间相对位置大于 k, 则认为他们之间的相对位置为 k; 相对位置编码矩阵 $w^K = (w_{-k}^K, \ldots, w_k^K)$, $w^V = (w_{-k}^V, \ldots, w_k^V)$ 为可学习的权重矩阵, 其中 $w_i^K, w_i^V \in \mathbb{R}^{d_a}$, $d_a = 64$.

编码器: 编码器部分由 12 层结构相同的编码器层线性叠加组成, 用以将输入的数据编码为一个稠密的表示, 旨在提取输入数据中最重要的信息, 将其转化为一个低纬度的向量, 同时尽可能保留原始数据的某些特征. 编码器层由一个具有相对位置编码[44]的多头自注意计算模块, 和一个前馈神经网络组成. 自注意机制旨在使模型能够在处理输入序列时对序列中不同位置的信息进行加权, 从而提升模型对相关信息的关注度和提取能力. 多头自注意力机制由多个自注意头组成, 目的在于通过不同的自注意力头, 在多个不同的投影空间中捕捉不同的交互信息并进行融合. 对于单个具有相对位置编码的注意力头而言, 其接收矩阵 $X = \mathrm{Concat}(x_1, \ldots, x_n), x \in \mathbb{R}^{n \times d_x}$ 作为输入, 利用三个线性变换矩阵 $W^Q, W^K, W^V \in \mathbb{R}^{d_x \times d_z}$ 将输入分别向量映射至属于三个子空间的三个矩阵, 分别记为 Q、K 和 V, 每个子空间反应不同的隐藏特征. 其具体计算过程可表示为:

$$Attention(Q,K,V) = \mathrm{softmax}\left(\frac{Q(K+P^K)^T}{\sqrt{d_z}}\right)\left(V+P^V\right)$$

其中, Q、K 和 V、 P^K 、 P^V 的计算过程如下:

$$Q = XW^Q, K = XW^K, V = XW^V$$
$$P^K = Concat\left(p_1^K, \ldots, p_n^K\right), p_n^K = \sum_{j=1}^{n} a_{ij}^K$$
$$P^V = Concat\left(p_1^V, \ldots, p_n^V\right), p_n^V = \sum_{j=1}^{n} a_{ij}^V$$

在所有自注意力头计算完毕后, 将其结果进行连接 (concatenate) 后送入前馈神经网络. 因此, 多头自注意力机制的具体过程可表示为:

$$\mathrm{head_i} = Attention\left(QW_i^Q, KW_i^K, VW_i^V\right)$$
$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head_h})W^O$$

其中, $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_x \times d_z}$ 为第 i 个注意力头中对输入进行映射的权重矩阵; $W^O \in \mathbb{R}^{hd_z \times d_x}$ 用以对合并后的向量进行线性映射以得到输出, 其中 h=12, 为子空间的数量, 其余参数值为 $\mathrm{d_z} = d_x/h = 64$.

最后, 由一层前馈神经网络 (Feed-Forward Networks) 对输入进行处理, 其目的在于通过增加非线性变换来使模型更加灵活, 可以更好地捕获各个输入序列元素之间的关系, 从而充分利用序列内的信息. 具体过程可表示为:

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

其中, $W_1, W_2 \in \mathbb{R}^{d_x \times d_x}$ 为对输入进行变换的权重矩阵, b_1, b_2 为偏移量, 不同编码器层中的前馈神经网络的权重矩阵具有独立的参数. 编码器由 12 层结构相同的编码器层线性叠加组成, 上一层的输出为下一层的输入. 最后, 抽取最后一层编码器层的输出作为提取到的隐藏层状态 $E \in \mathbb{R}^{n \times d_x}$.

解码器: 在得到了由编码器抽取到的隐藏层状态 E 之后, 模型利用解码器对该特征进行解码. 解码器由 12 层相同的解码层的线性叠加组成, 第一层的输入为隐藏层状态 E 和经过代码嵌入层的预期输出的向量矩阵 $O \in \mathbb{R}^{n \times d_x}$. 向量矩阵 O 首先通过掩码多头自注意力层, 而后同隐藏层状态 E, 送入多头自注意力层, 经过前馈神经网络后得到第一层的输出 U_1. 具体过程可表示为:

$$U_1 = FFN(MultiHead(E,E,O'))$$
$$O' = MaskMultiHead(O,O,O)$$

在此之后, 每 i 层编码器的输入为 $i-1$ 层编码器的输出. 具体过程可表示为:

$$U_i = FFN(MultiHead(E, E, U'i-1))$$
$$U'_{i-1} = MaskMultiHead(U_{i-1}, U_{i-1}, U_{i-1})$$

最后一层的输出接入一个全连接层, 将输出投影为维度等同于词表大小的向量, 而后通过 Softmax 函数将该向量转化为在词表上的概率分布以生成最终的令牌概率分布矩阵. 具体过程可表示为:

$$Q = \text{softmax}(U_{-1}W^{vocab})$$

其中, $W^{vocab} \in \mathbb{R}^{d_x \times d_{vocab}}$, d_{vocab} 为词表大小.

损失计算: 本文使用交叉熵损失函数对模型进行训练, 损失函数计算的具体过程可表示为:

$$Loss = -\sum_{i=1}^{n} P(t_i) log(Q(t_i))$$

其中, t_i 为模型生成的令牌流中的第 i 个元素, P 为理想情况下令牌的概率分布, Q 为模型生成的令牌概率分布.

3.3 漏洞修复

当模型经过训练后, 可用以生成候选修复程序. 具体而言, 首先提取原始漏洞程序, 在剔除所有注释后, 利用 BPE 算法进行分词, 得到漏洞程序. 后将漏洞程序输入模型, 模型对该序列进行特征抽取并解码, 产生构成修复程序的令牌概率分布. 生成修复程序的过程中, 我们利用了集束搜索策略选择概率最大的 k 个候选修复程序进行输出, 即得到模型预测的候选修复程序.

4 实验分析

4.1 研究问题

为了评估 CotRepair 方法的有效性, 我们研究了以下四个问题.

- RQ1:CotRepair 能否获得比当前先进漏洞修复方法更好的结果?
- RQ2:CotRepair 能否精确地预测代码中的修复位置?
- RQ3:CotRepair 能否在给定修复位置的情况下, 有效地产生修复程序?
- RQ4: 不同的模块对 CotRepair 方法的有效性产生怎样的影响?

4.2 实验数据

本文从两个公开漏洞数据集 Big-Vul[45]和 CVEFixes[8]获取缺陷并构建实验数据集 (以下简称为数据集). Big-Vul 数据集爬取了 CVE 数据库[46], 获取了例如 CVE-ID、CVE 严重程度分数和 CVE 摘要等信息. 随后, 作者利用爬虫, 对漏洞所涉及项目的源代码仓库进行了爬取, 以获取修复漏洞的 git 提交链接进而构建数据集. Big-Vul 共包含 2002 年至 2019 年, 348 个项目, 91 种 CWE 类型, 共 3 754 个不同的漏洞.

CVEFixes 的构建方式与 Big-Vul 类似, 但更改了数据来源与检索的时间范围. 具体而言, CVEFixes 直接从美国国家数据库 (National Vulnerability Database) 爬取漏洞, 并将检索项目的时间范围由 Big-Vul 数据集的 2002 年至 2019 年扩展至 1999 年至 2021 年. CVEFixes 收集了 1 754 个项目, 180 种 CWE 类型, 共 5 365 个不同的漏洞. 而后, 本文对 Big-Vul 和 CVEFixes 两个数据集进行合并去除重复数据, 最终得到含有 6, 008 个漏洞的数据集. 表 1 给出了数据集的统计信息.

表 1 实验数据集

数据集	缺陷数量	缺陷类型
Big-Vul	3 754	91
CVEFixes	5 365	180
去重后	6 008	180

4.3 评价指标

由于本文涉及到漏洞的修复位置定位及漏洞修复, 因此, 本文的评价指标包括修复位置定位效果评价指标和

漏洞修复效果评价指标.

(1) 修复位置定位效果评价指标

在评估修复位置定位效果时, 本文采用召回率@Top-n 和 Mean First Rank(MFR) 进行度量. 召回率@Top-n 是在缺陷定位中的常用评价指标[47−50], 本文参考其定义用以评估修复位置定位效果, 以衡量标准答案在预测列表中排在前 n 个的普遍程度, 计算方式如下:

$$@Top-n = \frac{\text{出现在预测结果Top-}n\text{个中的漏洞数}}{\text{漏洞总数}}$$

其中, n 越小代表缺陷定位结果更准确, 在本文的实验中, n 值的选取为 1, 5, 10, all. 召回率@Top-n 值越大代表缺陷定位的效果越好. 其中, 召回率@Top-all 记为召回率. 需要注意的是, 由于漏洞通常涉及到多个修改点位, 因此一个漏洞可能拥有多处修复位置, 为了更加准确反应修复位置定位的效果, 本文定义多个修改点位均被定位到才为成功定位.

Mean First Rank (MFR) 用于计算预测出的可疑代码元素排名列表中第一个缺陷代码元素的平均排名值[51], 计算方式如下:

$$MFR = \frac{\sum_{i=1}^{m} Rank(i)}{m}$$

其中, m 表示缺陷总数, $Rank(i)$ 表示在第 i 个缺陷中, 缺陷元素在预测结果中的最高排名. 因此, MFR 的值越低, 代表缺陷定位的精度越高.

(2) 修复效果评价指标

通常而言, 漏洞自动修复工具生成的修复程序整体上分为以下两种类型: 无效修复程序和有效修复程序. 其中, 无效修复程序指无法通过验证模块的修复程序, 有效修复程序指能通过验证模块的修复程序. 当前, 由于数据集中漏洞众多, 缺少运行环境及测试用例, 因此难以通过测试用例对漏洞的修复效果进行评估. 因此, 本文遵循当前通用漏洞修复方法[2,3]中的评估方式, 定义漏洞自动修复工具生成的修复程序当且仅当其和标准答案完全一致的情况下为有效修复程序. 基于此, 本文采用了缺陷修复领域的通用指标召回率 (recall) 作为评价指标. 指标的定义如下:

$$\text{召回率} = \frac{\text{生成有效修复程序的漏洞数}}{\text{漏洞总数}}$$

其中, 漏洞自动修复工具对漏洞产生的所有修复程序中 (本文中针对每个漏洞产生 50 个修复程序), 存在一个及以上有效修复程序即认为工具在该漏洞上生成了有效修复程序 (即和标准答案完全一致的修复程序).

此外, 为了更加充分地衡量补丁的生成效果, 本文还采用了程序自动修复领域常用的评价指标召回率@Top-n[52−54], 其衡量的是标准答案在预测列表中排在前 n 个的普遍程度, 计算方式为:

$$\text{召回率@Top-n} = \frac{\text{有效修复程序在预测结果}Top-n\text{个中的漏洞数}}{\text{漏洞总数}}$$

其中, n 越小代表补丁生成效果更准确, 在本文的实验中, n 值的选取为 1, 5, 10. 召回率@Top-n 的值越大代表补丁生成的效果越好.

4.4 基准方法

本文选择当前最先进的两种通用类型漏洞的自动修复方法 VRepair、VulRepair 将其作为基准方法与本文的方法进行比较.

VReapir. Chen 等人于 2021 年提出了基于迁移学习技术的漏洞修复方法 VRepair. 该方法利用迁移学习技术, 首先在 C/C++缺陷修复上进行训练, 而后在漏洞数据集上进行微调, 在一定程度上减轻了由于小数据集导致模型结果的不可信及不精确问题[55].

VulRepair. Fu 等人于 2022 年提出了基于 T5 预训练模型的漏洞修复方法 VulRepair. 该技术利用预训练模型

对代码较强的理解能力, 结合 BPE 分词, 在预训练模型的基础之上, 于漏洞数据集上进行微调, 进一步提升了漏洞修复的性能.

CotRepair-w/o-CoT. 为了更加准确地研究思维链设计对工具性能的影响, 我们设计了不含思维链设计的 CotRepair 作为基准方法之一, 记为 CotRepair-w/o-CoT. 具体而言, 该方法的输入为缺陷代码, 输出为修复代码. 相比于 CotRepair, 不对修复位置进行预测及输出.

需要注意的是, 由于 VulRepair 中使用的数据集为 Big-Vul 和 CVEFixes 简单合并, 并未进行去重. 本文对两个数据集进行合并后剔除了重复漏洞, 避免了数据集划分导致的训练集和测试集中出现相同漏洞而导致的数据泄露. VRepair 分别使用了 Big-Vul 和 CVEFixes 进行评估, 但研究发现, 在这两个数据集内部也存在完全一致的漏洞, 存在数据泄露的风险. 因此, 本文报告的 VRepair 与 VulRepair 的性能与原文存在差异.

4.5 实验设置

在本小节中对本文的实验设置的两方面进行介绍, 一是本文的方法是如何实现并微调的, 二是本文评估实验的设置.

方法实现与微调: 本文的模型架构基于 T5 模型的 Pytorch 实现构建, 并利用 CodeT5[9]的权重参数进行初始化. 代码已发布于: https://doi.org/10.5281/zenodo.8115328. 本文的所有实验在一台拥有 2 张英伟达 GeForce RTX 4090 GPU 的服务器上实现. 在对模型进行微调的过程中, 超参数的设置中, 最大输入长度和最大输出长度遵循 VulRepair 的设置以避免输入输出长度不同导致的不公平性, 其余超参数的设置依托在验证集上网格搜索 (grid search) 所获取的最佳超参数. 其中, 为了公平比较, 集束搜索中的束大小 (beam size) 与基准方法保持一致, 超参数的具体设置如表 2 所示.

表 2　模型超参数取值

词嵌入维度	学习率	训练轮数	批量大小	最大输入长度	最大输出长度	束大小
768	3e-4	75	32	512	512	50

评估实验:为了回答本文提出的研究问题, 本文构建的数据集来自于 Big-Vul 和 CVEFixes 两个大规模漏洞数据集, 实验数据集的划分上, 本文采用了和基准方法一样的数据划分方式, 即随机划分 70% 为训练集、10% 为验证集, 余下的 20% 为测试集. 训练过程中, 取验证集上表现最好的保存点 (checkpoint) 作为测试模型, 在测试集上进行评估测试.

4.6 实验结果与分析

RQ1:CotRepair 能否获得比当前先进漏洞修复方法更好的结果?

为了验证该问题的结果, 我们将 CotRepair 与三种基准方法在数据集上进行了对比实验, 实验的结果如表 3 所示.

表 3　CotRepair 与基线方法的修复性能比较

方法	召回率	召回率@Top-1	召回率@Top-5	召回率@Top-10
VRepair	12.3%	7.6%	11.2%	12.3%
VulRepair	17.5%	13.5%	16.4%	17.1%
CotRepair-w/o-CoT	23.0%	19.3%	22.3%	22.8%
CotRepair	**32.2%**	**27.0%**	**30.0%**	**31.2%**

我们注意到, 与基线方法相比, CotRepair 的性能有较大提升. 具体而言, 从召回率上看, CotRepair 的召回率达到了 32.2%, 对应基线方法 VRepair、VulRepair 方法分别提升了 161.2%、84.0%. 意味着 CotRepair 相比于当前最先进工具 VulRepair 能够多产生 84.0% 的正确修复程序. 从召回率@Top-1 上来看, CotRepair 相比于基线方法 VRepair、VulRepair 的提升率分别达到了 255.3%、100.0%. 为了更加准确地评估思维链的作用, 我们对 CotRepair-w/o-CoT 进行了评估, 结果显示, 相较于没有思维链设计的情况下, CotRepair 在召回率和召回率@Top-1 上的提升

分别达到了 40.0% 和 40.6%. 这意味着思维链的加入缩减了修复程序的生成空间, 提升了生成的修复程序质量, 在一定程度上缓解了基线方法修复程序生成质量较差的问题. 我们同样计算了正确修复程序在 Top-1 的占比 (即召回率@Top-1/召回率), CotRepair 生成的正确修复程序中 83.9% 位于 Top-1, 而 VRepair、VulRepair、CotRepair-w/o-CoT 分别为 61.8%、77.1%、83.5%. 这意味着 CotRepair 生成正确修复程序相比于基线方法, 更加集中于 Top-1, 能够显著降低开发人员在修复程序正确性验证上的耗费. 本文同时还利用了 Wilcoxon 符号秩检验, 对 CotRepair 与各基线方法中**有效修复程序在预测结果中的位次**是否存在显著差异进行检验, 以验证 CotRepair 与各基线方法在修复性能上是否有显著差异. 评估结果显示, CotRepair 与各基线方法之间的 p 值均小于 0.001, 即 CotRepair 与各模型之间的性能差异十分显著.

综上所述, 针对漏洞程序, CotRepair 能够有效地对缺陷进行修复, 并显著优于基线方法.

RQ2:CotRepair 能否精确地预测代码中的修复位置?

准确修复漏洞的前提为精确地定位修复位置, 因而我们对该问题进行了探究. 具体而言, 我们对各个方法产生的输出进行分析, 对于 VRepair 和 VulRepair, 由于其输出结果为变更后的代码及其上下文, 我们对其产生变更的位置进行分析, 如果产生变更的位置与修复位置一致, 则认为该方法准确地预测了代码中的修复位置. 对于 CotRepair-w/o-CoT 和 CotRepair 而言, 由于其输出结果为完整的变更后代码, 我们将其与漏洞程序进行比较并对产生变更的位置进行分析, 如果与修复位置一致, 则认为其准确地预测了代码中的修复位置. 实验的结果如表 4 所示.

表 4 各方法对修复位置的预测效果

方法	召回率	召回率@Top-1	召回率@Top-5	召回率@Top-10	MFR
VRepair	19.7%	15.7%	18.4%	19.1%	40.3
VulRepair	41.2%	29.2%	37.2%	39.4%	30.0
CotRepair-w/o-CoT	34.6%	25.1%	31.3%	33.2%	33.1
CotRepair	**52.4%**	**36.6%**	**44.9%**	**47.8%**	**24.8**

我们注意到, CotRepair 相比于基线方法, 能够更为准确地捕获代码中缺陷的位置. 具体而言, CotRepair 对于修复位置的召回率为 52.4%, 意味着在不考虑修复程序正确性的情况下, CotRepair 生成的修复程序中, 52.4% 的修复程序在正确的修复位置进行了修改, 而 VRepair、VulRepair、CotRepair-w/o-CoT 的召回率分别为 19.7%、41.2%、34.6%. 可以推测, 在思维链技术的帮助之下, 模型对于修复位置的识别效果有了较大提升, 为漏洞修复打下了基础. 与此同时, 我们注意到, CotRepair 在 MFR 指标上相较于基线方法也有较大提升, 相较于当前表现最好技术 VulRepair, CotRepair 在 MFR 指标上下降了 17.3%, 意味着该方法能够更为准确地预测漏洞代码中出现缺陷的位置. 为了验证 CotRepair 与各基线方法在修复位置预测效果上是否有显著差异, 我们利用 Wilcoxon 符号秩检验对 CotRepair 与基线方法之间的修复位置预测效果 (以标准答案在预测列表中的位次记) 进行了检验, 结果显示 CotRepair 与各基线方法之间的 p 值均小于 0.001, 即 CotRepair 与各模型之间的修复位置预测效果差异十分显著.

综上所述, 针对漏洞程序, CotRepair 能够精确地预测漏洞程序中的修复位置, 并显著优于基线方法.

RQ3:CotRepair 能否在给定修复位置的情况下, 有效地产生修复程序?

为了探究该问题的结果, 我们探究了各方法在修复位置预测正确条件下生成正确修复程序的能力, 即在评价指标计算时, 仅考虑修改位置正确的修复程序; 若方法对一个漏洞程序产生的所有修复程序中没有修改位置正确的程序, 则在计算召回率时不考虑该漏洞程序. 实验的结果如表 5 所示.

表 5 修复位置预测正确的条件下各方法的修复效果

方法	召回率	召回率@Top-1	召回率@Top-5	召回率@Top-10	修复位置预测正确数
VRepair	57.4%	40.5%	53.2%	56.1%	237
VulRepair	41.8%	24.8%	30.1%	31.3%	495
CotRepair-w/o-CoT	**65.7%**	**55.4%**	**64.0%**	**65.4%**	417
CotRepair	61.4%	51.4%	57.3%	59.5%	**630**

可以看出, 在修复位置预测正确的情况下, 相比于不限定修复位置预测正确性, 各方法的有效性均有所提升. 从召回率的提升幅度来看, VRepair、VulRepair、CotRepair-w/o-CoT、CotRepair 分别提升了 394.8%、138.9%、185.7%、90.7%. 其中, VRepair 的提升幅度最大, CotRepair-w/o-CoT 次之, CotRepair 的提升幅度最小. VRepair 和 CotRepair-w/o-CoT 在修复位置预测正确的情况下, 在召回率指标上表现较好, 我们分析是由于 VRepair 与 CotRepair-w/o-CoT 本身对修复位置预测的能力较弱, 仅分别在 237、417 个漏洞上正确预测了修复位置, 这些漏洞多为较容易修复的漏洞, 因此能够在修复位置预测正确的情况下达到较高的召回率. CotRepair-w/o-CoT 在失去思维链设计的情况下, 修复程序的搜索空间较大, 产生了较多的错误修复程序, 因而在修复位置预测正确的条件约束下, 有效性提升明显, 本文在 5.5 节中对这一现象产生的原因进行了详细分析;CotRepair 在含有思维链设计的条件下, 本身能够较好地对修复位置进行预测, 因此修复位置预测正确的条件约束对 CotRepair 产生的影响较小, 因此提升幅度最小; 基于分析可以得出推论, 思维链设计能够显著缩小修复程序的搜索空间. 我们同样注意到, VulRepair 有较大的性能提升, 意味着这些方法本身对于修复位置预测的能力较弱, 因此在修复位置预测正确条件下能够获得较大的性能提升.

为了更加充分地探究 CotRepair 能否在给定修复位置的情况下有效地产生修复程序, 本文在另一实验设置下对该问题开展了探究: 对输入的漏洞程序中缺陷的修复位置进行精确标记, 作为提示模板的一个部分, 模型将依托给定的修复位置信息输出修复程序. 图 6 展示了对漏洞程序中缺陷修复位置进行精确标记后的输入. 精确标记后的漏洞程序序列由漏洞程序序列和修复位置序列组成, 二者之间以符号 [SEP] 进行连接, [SEP] 为用以分割两种数据类型的预定义符号. 实验结果如表 6 所示.

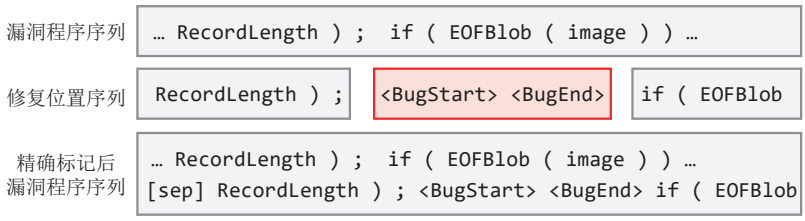

图 6 漏洞程序中缺陷修复位置精确标记方式

表 6 提供修复位置的条件下各方法的修复效果

方法	召回率	召回率@Top-1	召回率@Top-5	召回率@Top-10
VRepair	15.9%	11.1%	14.6%	15.1%
VulRepair	35.9%	25.1%	34.4%	35.6%
CotRepair-w/o-CoT	37.8%	32.5%	35.5%	36.9%
CotRepair	43.3%	36.2%	38.8%	40.5%

评估结果显示, 相比于未对输出的漏洞程序进行标记的情况下 (即 RQ1 的设置), VRepair、VulRepair、CotRepair-w/o-CoT、CotRepair 在召回率上的提升分别达到了 29.3%、105.1%、64.3% 和 34.5%, 其中 VulRepair 的提升幅度最大, 意味着 VulRepair 在未对输出的漏洞程序进行标记的情况下, 对漏洞中缺陷的修复位置捕获能力较弱, 导致其性能受到限制; VRepair 的提升幅度较小, 我们推测是由于 VRepair 模型规模较小, 且缺少预训练模型所提供的领域知识, 导致其无法准确捕获输入中标记的缺陷修复位置; CotRepair-w/o-CoT 的提升幅度显著大于 CotRepair, 印证了 CotRepair 借助思维链设计, 本身对缺陷修复位置的推测能力较强.

综上所述, 在给定修复位置的情况下, 各方法的有效性均有较大程度的提升. 这意味着当前限制各方法修复能力的主要因素为其对修复位置的预测能力. 思维链设计能够在很大程度上帮助方法对修复缺陷位置进行预测. 因此, 探究如何提升模型对修复位置的预测能力以提升修复能力颇有前景.

RQ4: 不同的模块对 CotRepair 方法的有效性产生怎样的影响?

为了研究本文方法中不同的模块对模型有效性带来的影响, 我们进行了消融实验. 具体而言, 我们关注于

CotRepair 中的三个设计: (1) 利用思维链进行修复程序生成; (2) 基于预训练模型进行微调; (3) 将完整的修复代码作为模型输出. 针对第一个设计, 我们移除思维链, 即在训练过程中移除所有的缺陷定位信息, 该方法的输入为缺陷代码, 输出为修复代码. 需要注意的是, 该方法即为 4.4 节基准方法中所设计的 CotRepair-w/o-CoT. 相比于 CotRepair, 该方法不对修复位置进行预测及输出, 仅对修复后的代码进行输出; 针对第二个设计, 我们不使用预训练模型, 即将预训练模型中的权重矩阵进行随机初始化; 针对第三个设计, 我们将模型的输出修改基线方法中的输出, 即在修复过程中仅输出变更的代码令牌及其上下文. 实验结果如表 7 所示.

表 7　消融实验结果

模型设计	召回率	召回率@Top-1	召回率@Top-5	召回率@Top-10
-思维链	23.0%	19.3%	22.3%	22.8%
-预训练模型	23.3%	19.9%	22.5%	23.0%
-完整修复代码	24.3%	20.4%	23.0%	23.3%
CotRepair	**32.2%**	**27.0%**	**30.0%**	**31.2%**

我们注意到, 在剔除思维链设计后, 方法的召回率下降幅度达到了 28.6%, 表明在缺少思维链的情况下, 可能的修复空间较大, 导致模型难以在限定的输出长度下搜索到正确的修复程序. 在缺少预训练模型的情况下, 方法的召回率下降了 27.6%, 意味着预训练模型所提供的领域知识能够极大帮助方法在微调数据集较小的情况下获得修复程序的生成能力, 在一定程度上减轻了由于部分漏洞类别数量较少导致难以学习到修复规律的问题. 在将方法的输出结果由完整修复代码更改为变更的代码令牌及其上下文后, 方法的召回率下降了 24.5%, 意味着直接输出完整修复代码, 能够更加充分地利用预训练模型中的捕获的信息, 避免了微调任务与预训练任务之间的距离过大所导致的性能下降. 从整体趋势而言, 以召回率、召回率@Top-1、召回率@Top-5 和召回率@Top-10 作为评估指标时, 思维链设计仍然对模型的性能影响最大; 输出完整修复代码的设计对模型性能的影响小于预训练模型, 意味着直接输出变更的代码令牌及其上下文的情况下, 模型仍然能够利用部分预训练模型中捕获的领域知识, 因而在此情况下性能仍好于没有预训练模型的情况.

综上所述, CotRepair 方法中的各个设计共同作用使该方法的有效性达到最优, 其中思维链设计对方法有效性的提升最明显, 其次是预训练模型.

5　分析与讨论

5.1　缺陷类型对修复有效性影响分析

表 8 展示了 CotRepair 在出现频率最高的 10 类漏洞上的表现. 可以看出, CotRepair 在 CWE-190 漏洞上表现最佳, 召回率达到了 42.1%, 修复了 38 个漏洞中的 16 个. 在 CWE-264, CWE-399 和 CWE-416 上分别修复了 32.0%, 29.0% 和 28.6% 的漏洞. 在 CWE-476 和 CWE-20 这两类漏洞上的表现最差, 仅能修复 20.5% 和 24.8% 的漏洞. 经过分析, 发现在这两类漏洞上, 漏洞程序长度的中位数分别为 536.5、593, 而表现较好的 CWE-190 漏洞上的中位数仅为 286. 由于 CotRepair 的性能随着漏洞程序长度的增加而下降 (5.2 节中进行详细讨论), 因此, 在这两类漏洞上的表现较差. 就出现频率最高的 10 类漏洞上的总体表现而言, 召回率为 31.6%, 修复了 805 个漏洞中的 254 个; 而 CotRepair 在完整数据集上的召回率为 32.2%, 与出现频率最高的 10 类漏洞上的召回率差距较小, 即 CotRepair 在出现频率较低的漏洞类别上的表现与出现频率最高的漏洞类别上接近, 意味着 CotRepair 不需要大量的漏洞数据即可学习到漏洞的修复方式.

5.2　漏洞程序长度对修复有效性影响分析

表 9 展示了 VRepair、VulRepair 和 CotRepair 在不同长度漏洞程序上的表现. 我们发现, 上述技术在不同长度漏洞程序上的表现整体上随着漏洞程序长度和缺陷行数的增加而下降. 在长度小于 100 的漏洞程序之中, VulRepair 和 CotRepair 分别修复了其中 44.6%、60.5% 的漏洞程序, 在长度为 300 至 400 的漏洞程序上分别修复

了 16.9% 和 39.6% 的漏洞程序. CotRepair 相比于 VRepair 和 VulRepair, 在长度较长的漏洞程序上性能提升较为明显. 具体而言, CotRepair 相比于 VRepair 和 VulRepair 在长度为 0-100 的漏洞程序上提升分别为 87.3% 和 35.7%, 而在长度为 401-500 的漏洞程序上提升分别为 171.6% 和 89.6%. 我们推测是由于 CotRepair 能够更为准确地定位修复位置, 缩减修复程序的生成空间, 因而在漏洞程序较长的情况下也能够较好地对漏洞程序进行修复. 此外, 本文通过分析在不同长度的漏洞程序中的平均缺陷行数对漏洞修复的复杂性进行探究, 研究发现, 在长度为 0-100 的漏洞程序中, 漏洞涉及到的平均行数仅为 1.43, 而长度为 300 至 400 的漏洞程序上漏洞涉及到的平均行数为 2.59, 这意味着随着漏洞平均长度的增加, 平均缺陷行数也显著增加, 漏洞修复的难度的随之增加. 因此, VRepair、VulRepair 和 CotRepair 的性能表现均随着平均缺陷行数的增加而下降. 为了探究 CotRepair 对于多行缺陷的修复能力, 我们对 CotRepair 在缺陷行数大于 1 的漏洞上的性能进行了评估, 评估结果显示, CotRepair 在此情况下的召回率为 19.5%, 意味着其在多行缺陷的情况下, 仍然能够保持对漏洞修复位置的定位及修复能力. 此外, 由于本文基于 T5 模型, 而 T5 模型的最大输入长度为 512, 即输入的漏洞程序长度超过 512 个令牌的部分会被截断丢弃, 导致 CotRepair 在长度大于 500 的漏洞程序上的召回率上发生较大幅度的下降. 而在数据集中, 51.2% 的漏洞长度超过了 500, 导致了 CotRepair 最终的召回率较低. 其余长度的漏洞分布较为均匀, 其中长度为 0-100 的漏洞占比最低, 为 7.2%; 长度为 101-200 的漏洞占比为 12.6%. 利用最大输入长度较大的预训练模型, 如 CodeGen[32], 可以在一定程度上缓解由于模型对超长输入进行截断而导致的性能大幅下降的问题. 由于该类模型规模较大, 训练所耗费的时间与资源较多, 本文将其作为未来工作.

表 8　CotRepair 在出现频率最高的 10 类漏洞上的表现

序号	CWE类型	CWE描述	召回率	比例
1	CWE-119	内存缓冲区限制不当	32.7%	101/308
2	CWE-125	越界读取	26.8%	30/112
3	CWE-20	输入验证不当	24.8%	26/105
4	CWE-264	权限、特权和访问控制不当	46.0%	23/50
5	CWE-476	空指针解引用	20.5%	9/44
6	CWE-200	敏感信息未经授权泄露	32.6%	14/43
7	CWE-416	释放后使用	35.7%	15/42
8	CWE-190	整数溢出或越界折返	44.7%	17/38
9	CWE-787	内存越界写入	34.4%	11/32
10	CWE-399	资源管理错误	35.5%	11/31
	合计		31.6%	254/805

表 9　各方法在不同长度漏洞程序上的表现

漏洞程序长度	平均缺陷行数	占比	VRepair召回率	VulRepair召回率	CotRepair召回率
0-100	1.43	7.2%	32.3%	44.6%	60.5%
101-200	1.93	12.6%	15.6%	31.2%	47.3%
201-300	2.35	11.0%	14.1%	27.6%	42.4%
301-400	2.59	9.7%	11.0%	16.9%	39.6%
401-500	2.47	8.5%	14.8%	21.2%	40.2%
>501	3.52	51.2%	9.7%	9.8%	19.5%

综上所述, VRepair、VulRepair 和 CotRepair 的性能在很大程度上取决于输入的漏洞程序长度和漏洞程序的缺陷行数, 整体上随着漏洞程序长度和缺陷行数的增加而下降, 在较短的漏洞程序和缺陷行数较少的漏洞程序上能够取得较高的召回率.

5.3　数据表示方式的局限性分析

本文在 3.1 节中对本文构建的带上下文的修复位置表示方式进行了介绍, 即通过将缺陷修复位置一定范围内

的上下文连同缺陷代码一同输出作为修复位置的表示, CotRepair 通过该表示定位缺陷代码中需要被更改的代码. 该表示方式中上下文范围的选择对表示方式的合理性有较大的影响; 当上下文范围过小时, 可能存在无法在缺陷代码中准确定位待修复位置的问题; 当上下文范围过大时, 过长的修复位置表示挤占大量的输出空间引起修复性能的下降. 因此, 在满足大多数情况下无歧义的前提下, 上下文大小的选择应尽可能小. 图 7 展示了漏洞 CVE-2017-14341 在不同上下文大小对应的修复位置表示方式, 其中红框所对应的位置表示上下文大小为 2 时, 修复位置表示在缺陷代码中的匹配点, 绿框为上下文大小为 3 时的匹配点. 可以看出, 在上下文大小为 2 时, 缺陷代码中存在多个匹配点, 导致了混淆. 本文定义仅能在缺陷代码中匹配一次的修复位置表示方式为无歧义的修复表示方式. 本文通过对不同上下文大小时修复位置表示的无歧义率进行分析得到, 在上下文大小为 2、3 和 4 时, 无歧义率分别为 77.3%、88.6% 和 91.3%. 基于本文的分析及 Chen 等人的研究[3], 当上下文大小为 3 时, 足以在大多数情况下唯一定位代码片段在代码中的位置, 因此本文对于缺陷的上下文的界定为缺陷令牌周围的 3 个令牌.

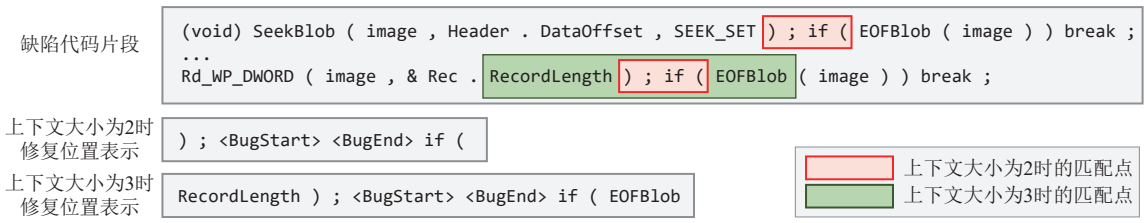

图 7 不同上下文大小对应的修复位置表示方式

5.4 与当前大语言模型的对比

为了探究大语言模型在漏洞自动修复上的有效性, 本文选择了在当前最受欢迎的大语言模型对话机器人 ChatGPT 上开展实验. 我们在两种设置下开展了实验:

设置一: 输入为漏洞程序, 输出为修复程序, 即不含有思维链设计. 输入提示语为“You are an automated software vulnerability repair tool. Please assist me in repairing the following vulnerable code …”, 提示语中第一句是为了准备 ChatGPT 进行漏洞自动修复任务, 第二句描述了详细的任务要求, 即要求 ChatGPT 直接输出修复后的缺陷代码.

设置二: 输入为漏洞程序, 输出为缺陷位置及修复代码, 即含有思维链设计. 输入提示语为“You are an automated software vulnerability repair tool. Please provide the defective code snippet first and then repair the following vulnerable code based on the defective code snippet …”. 提示语中第一句是为了准备 ChatGPT 进行漏洞自动修复任务, 第二句描述了详细的任务要求, 即要求 ChatGPT 先进行缺陷定位, 而后依托缺陷定位结果, 输出修复后的缺陷代码.

该实验利用了 ChatGPT API(访问于 2023 年 6 月 20 日), 两种设置下温度参数均设置为 0, 意味着 ChatGPT 将始终返回具有最高概率的代码. 结果显示, ChatGPT 在不含思维链设计的情况下返回的修复程序召回率为 4.6%, 在含有思维链设计的情况下为 6.7%. 显著低于 CotRepair 及基线方法. 结果表明, 在含有思维链设计的情况下, 相比于不含思维链设计在召回率指标上有所提升, 意味着思维链设计对大语言模型也有显著帮助. 此外, 尽管以 ChatGPT 为代码的大语言模型在许多任务上已显著超过当前的最先进方法, 但在细分领域上, 距离当前的先进方法还有较大差距. 一个合理的推测为当前的先进方法已经在任务相关的数据集上进行了充分微调, 而 ChatGPT 是为人工通用智能 (artificial general intelligence, AGI) 而优化, 而不是专门为漏洞修复优化.

5.5 案例分析

在 RQ3 中, 我们探究了各方法在修复位置预测正确条件下生成正确修复程序的能力, 发现 CotRepair-w/o-CoT 在该条件下召回率高于 CotRepair, 我们推测是由于 CotRepair-w/o-CoT 在较容易修复的漏洞上能够较好地预测修复位置. 为了更好地阐述该情况, 本节中对两个案例进行分析. 图 8 展示了编号为 CVE-2018-8797 漏洞实例

的官方修复补丁, 该漏洞属于内存越界写入 (CWE-787). 通过对 while 语句的判定条件进行修改达到了约束内存写入的范围的目的, 实现了对该漏洞的修复. 该漏洞的修复模式较简单, 仅对表达式进行更改, 是自动修复领域常见的修复模式之一, 已被诸多基于模板的自动修复工具所使用[52,56−60]. 因此, 该漏洞能够被 VulRepair、CotRepair-w/o-CoT 和 CotRepair 成功定位并修复. 该案例表明, 对于修复模式较为简单的漏洞, 漏洞修复工具能够较为准确地定位缺陷产生的位置. 相较而言, 图 9 展示了编号为 CVE-2017-15649 漏洞实例的官方修复补丁, 该漏洞属于共享资源并发执行时同步不当 (CWE-362), 该漏洞的修复过程涉及到较复杂的逻辑更改, 尚无匹配的修复模式. 具体而言, 该漏洞的修复通过对变量 po->bind_lock 增加自旋锁 (spin lock), 并在条件判断语句中增加了对 po->running 状态的判断, 以避免对不同线程之间对共享资源的竞争. 在该漏洞上, 仅 CotRepair 成功定位了修复位置, 但并未修复成功. 综上, 由于 CotRepair 能够定位此类修复较为复杂的漏洞, 但修复能力上还不足以支持此类漏洞的修复, 因此导致了召回率略低于 CotRepair-w/o-CoT.

```
 collen = 0;
 }
-    while (collen > 0) {
+    while (indexw < width && collen > 0) {
     color = CVAL( in ); * out = color;
     out += 4;
```

图 8 CVE-2018-8 797 漏洞的补丁

```
     }
     err = -EINVAL;
-    if (match->type == type &&
+    spin_lock(&po->bind_lock);
+    if (po->running && match->type == type &&
         match->prot_hook.type == po->prot_hook.type &&
         match->prot_hook.dev == po->prot_hook.dev) {
```

图 9 CVE-2017-15 649 漏洞的补丁

5.6 独特性分析

为了更好地研究 CotRepair 与当前通用类型漏洞自动修复技术之间的区别, 本文对 CotRepair 所能修复的漏洞的独特性进行了分析. 我们首先对 CotRepair 与基线方法之间所能修复的漏洞之间的重叠情况进行了分析, 分析结果如图 10 所示. CotRepair 能够修复其他基线方法无法修复的 118 个漏洞, 而 CotRepair-w/o-CoT、VRepair、VulRepair 能唯一修复的漏洞分别为 14、4 和 3 个. 此外, 我们注意到, CotRepair 能够修复 141 个 CotRepair-w/o-CoT 无法修复的漏洞, 展示了思维链设计所带来的修复性能提升.

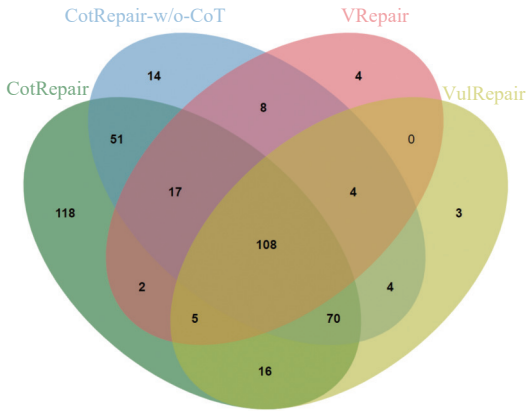

图 10 CotRepair 与基线方法之间修复漏洞的重叠情况

为了更加充分地分析 CotRepair 的设计所带来的优势, 本文探究了 CotRepair 与基线方法能共同修复的漏洞、和 CotRepair 能唯一修复的漏洞在出现频率最高的 10 类漏洞类型上的分布, 结果如表 10 和表 11 所示. 其中, 唯一修复率定义为 CotRepair 在该漏洞类型下唯一修复的漏洞数量与 CotRepair 在该类型下修复总数之比. 该指标用以衡量 CotRepair 在特定类型漏洞下修复的漏洞中有多少漏洞仅 CotRepair 能够修复. 评估结果表明, CotRepair 在其他方法修复效果较好的漏洞类型, 如 CWE-119, 能够独有地修复较多数量的漏洞; 在其他方法修复效果较差、CotRepair 召回率也较低的漏洞, 如 CWE-125、CWE-20 和 CWE-476, 其唯一修复率达到较高程度, 分别为 56.7%、42.3% 和 44.4%. 这意味着, 对于基线方法表现较好的漏洞类型, CotRepair 同样表现较好; 对于修复难度较大 (即召回率较低) 的漏洞类型, CotRepair 所能修复的漏洞中的大量漏洞其他基线方法无法修复. 表明 CotRepair 融合了修复位置预测的设计, 能够在修复难度较大的漏洞上, 弥补修复基线方法的不足.

表 10 CotRepair 与基线方法能共同修复的漏洞在出现频率最高的 10 类漏洞类型上的分布

序号	CWE类型	CWE描述	共同修复数
1	CWE-119	内存缓冲区限制不当	20
2	CWE-125	越界读取	7
3	CWE-20	输入验证不当	6
4	CWE-264	权限、特权和访问控制不当	4
5	CWE-476	空指针解引用	0
6	CWE-200	敏感信息未经授权泄露	3
7	CWE-416	释放后使用	7
8	CWE-190	整数溢出或越界折返	4
9	CWE-787	内存越界写入	2
10	CWE-399	资源管理错误	7

表 11 CotRepair 能唯一修复的漏洞在出现频率最高的 10 类漏洞类型上的分布

序号	CWE类型	CWE描述	修复总数	唯一修复数	唯一修复率*	召回率
1	CWE-119	内存缓冲区限制不当	101	28	27.7%	32.7%
2	CWE-125	越界读取	30	17	56.7%	26.8%
3	CWE-20	输入验证不当	26	11	42.3%	24.8%
4	CWE-264	权限、特权和访问控制不当	23	5	21.7%	46.0%
5	CWE-476	空指针解引用	9	4	44.4%	20.5%
6	CWE-200	敏感信息未经授权泄露	14	5	35.7%	32.6%
7	CWE-416	释放后使用	15	3	20.0%	35.7%
8	CWE-190	整数溢出或越界折返	17	6	35.3%	44.7%
9	CWE-787	内存越界写入	11	3	27.3%	34.4%
10	CWE-399	资源管理错误	11	1	9.1%	35.5%

*唯一修复率表示CotRepair在该漏洞类型下唯一修复的漏洞数量与CotRepair在该类型下修复总数之比

5.7 修复效率

精确的修复位置有利于获得精确的搜索空间, 理论上能够以更小的搜索代价获得高质量的补丁, 从而减少无用补丁的数量并减少花费在补丁生成上的时间以提升工具的效率. 然而在本文的实验中, 为了实现 CotRepair 和基线方法的公平比较, 我们统一了所有方法生成的补丁数量 (在 4.5 节中阐明). 补丁生成时间效率方面, 由于 CotRepair、CotRepair-w/o-CoT 和 VulRepair 均基于 CodeT5 模型, 模型参数量相似, 单个补丁生成时间方面无较大差别. 因此, 本文主要关注于在同样的计算耗费下 (即限制补丁的生成数量相同), 不同方法生成有效补丁的能力.

6 总 结

漏洞自动修复技术旨在自动地对漏洞进行修复, 从而降低漏洞对现代软件及数据的完整性、安全性和可靠性

带来的威胁, 漏洞自动修复技术的发展深切地影响漏洞的治理水平. 本文提出了一种基于思维链的通用类型漏洞修复方法. 针对当前通用类型漏洞缺陷自动修复方法中存在的修复程序生成质量较低的这一问题, 本文提出了利用思维链技术, 帮助模型生成更加准确地生成修复程序. 本文基于 T5 预训练模型, 通过以漏洞程序为输入, 在训练过程中辅以修复位置信息进行学习, 生成候选修复程序帮助开发人员更快更精确地对漏洞程序进行修复. 在本文构建的数据集上的实验表明, 本文提出的漏洞修复方法不仅可以较为准确地产生修复程序, 而且显著优于当前的先进方法, 具有较好前景.

References:

[1] Veracode. State of software security. https://info.veracode.com/report-state-of-software-security-volume-12.html

[2] Fu M, Tantithamthavorn C, Le T, Nguyen V, Phung D. VulRepair: A T5-based automated software vulnerability repair. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Singapore: ACM, 2022. 935–947. [doi: 10.1145/3540250.3549098]

[3] Chen ZM, Kommrusch S, Monperrus M. Neural transfer learning for repairing security vulnerabilities in C code. IEEE Transactions on Software Engineering, 2023, 49(1): 147–165. [doi: 10.1109/tse.2022.3147265]

[4] Zhang YT, Gao X, Duck GJ, Roychoudhury A. Program vulnerability repair via inductive inference. In: Proc. of the 31st ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2022. 691–702. [doi: 10.1145/ 3533767.3534387]

[5] Gao X, Wang B, Duck GJ, Ji RY, Xiong YF, Roychoudhury A. Beyond tests: Program vulnerability repair via crash constraint extraction. ACM Transactions on Software Engineering and Methodology, 2021, 30(2): 14. [doi: 10.1145/3418461]

[6] CWE. https://cwe.mitre.org, 2023.

[7] Xu TT, Liu K, Xia X. Survey on automated vulnerability repair. Journal of Software, 2024, 35(1): 136–158 (in Chinese with English abstract). [doi: 10.13328/j.cnki.jos.006828]

[8] Bhandari G, Naseer A, Moonen L. CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software. In: Proc. of the 17th Int'l Conf. on Predictive Models and Data Analytics in Software Engineering. Athens: ACM, 2021. 30–39. [doi: 10.1145/ 3475960.3475985]

[9] Wang Y, Wang WS, Joty S, Hoi SCH. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proc. of 2021 Conf. on Empirical Methods in Natural Language Processing. Punta Cana: Association for Computational Linguistics, 2021. 8696–8708. [doi: 10.18653/v1/2021.emnlp-main.685]

[10] Feng ZY, Guo DY, Tang DY, Duan N, Feng XC, Gong M, Shou LJ, Qin B, Liu T, Jiang DX, Zhou M. CodeBERT: A pre-trained model for programming and natural languages. In: Proc. of Findings of the Association for Computational Linguistics. Association for Computational Linguistics, 2020. 1536–1547. doi: 10.18653/v1/2020.findings-emnlp. 139]

[11] Guo DY, Ren S, Lu S, Feng ZY, Tang DY, Liu SJ, Zhou L, Duan N, Svyatkovskiy A, Fu SY, Tufano M, Deng SK, Clement CB, Drain D, Sundaresan N, Yin J, Jiang DX, Zhou M. GraphCodeBERT: Pre-training code representations with data flow. In: Proc. of the 9th Int'l Conf. on Learning Representations. OpenReview.net, 2021.

[12] Ahmad W, Chakraborty S, Ray B, Chang KW. Unified pre-training for program understanding and generation. In: Proc. of 2021 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Association for Computational Linguistics, 2021. 2655–2668. [doi: 10.18653/v1/2021.naacl-main.211]

[13] Lu P, Mishra S, Xia T, Chang KW, Zhu SC, Tafjord O, Clark P, Kalyan A. Learn to explain: Multimodal reasoning via thought chains for science question answering. In: Proc. of the 36th Conf. on Neural Information Processing Systems. New Orleans: NeurIPS, 2022. 2507–2521.

[14] Wei J, Wang XZ, Schuurmans D, Bosma M, Ichter B, Xia F, Chi E, Le O, Zhou D. Chain-of-thought prompting elicits reasoning in large language models. arXiv preprint arXiv:2201.11903, 2023. [doi: 10.48550/arXiv.2201.11903]

[15] 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]

[16] Zhang Y, Kabir M, Xiao Y, Yao DF, Meng N. Data-driven vulnerability detection and repair in java code. arXiv preprint arXiv: 2102.06994, 2021. [doi: 10.48550/arXiv.2102.06994]

[17] Huang Z, Lie D, Tan G, Jaeger T. Using safety properties to generate vulnerability patches. In: Proc. of 2019 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2019. 539–554. [doi: 10.1109/SP.2019.00071]

[18] Shaw A, Doggett D, Hafiz M. Automatically fixing C buffer overflows using program transformations. In: Proc. of the 2014 44th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks. Atlanta: IEEE, 2014. 124–135. [doi: 10.1109/DSN.2014.25]

[19] Hafiz M, Overbey J, Behrang F, Hall J. OpenRefactory/C: An infrastructure for building correct and complex C transformations. In: Proc. of 2013 ACM Workshop on Workshop on Refactoring Tools. Indianapolis: ACM, 2013. 1–4. [doi: 10.1145/2541348.2541349]

[20] Condit J, Harren M, McPeak S, Necula GC, Weimer W. CCured in the real world. In: Proc. of ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation. San Diego: ACM, 2003. 232–244. [doi: 10.1145/781131.781157]

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

[22] 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: ACM, 2010. 215–224. [doi: 10.1145/1806799.1806833]

[23] 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]

[24] 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]

[25] 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 Conf. on Neural Information Processing Systems. Montréal: NeurIPS, 2018. 31.

[26] Creswell A, White T, Dumoulin V, Arulkumaran K, Sengupta B, Bharath AA. Generative adversarial networks: An overview. IEEE Signal Processing Magazine, 2018, 35(1): 53–65. [doi: 10.1109/MSP.2017.2765202]

[27] Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou YQ, Li W, Liu PJ. Exploring the limits of transfer learning with a unified text-to-text transformer. The Journal of Machine Learning Research, 2020, 21(1): 140. [doi: 10.5555/3455716.3455856]

[28] Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I. Language models are unsupervised multitask learners. OpenAI Blog, 2019, 1(8): 9.

[29] Brown TB, Mann B, Ryder N, Subbiah M, Kaplan J, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, Agarwal S, Herbert-Voss A, Krueger G, Henighan T, Child R, Ramesh A, Ziegler DM, Wu J, Winter C, Hesse C, Chen M, Sigler E, Litwin M, Gray S, Chess B, Clark J, Berner C, McCandlish S, Radford A, Sutskever I, Amodei D. Language models are few-shot learners. In: Proc. of the 34th Conf. on Neural Information Processing Systems. Vancouver: NeurIPS, 2020. 1877–1901.

[30] Nijkamp E, Pang B, Hayashi H, Tu LF, Wang H, Zhou YB, Savarese S, Xiong CM. CodeGen: An open large language model for code with multi-turn program synthesis. In: Proc. of ICLR. 2020.

[31] Wang Y, Le H, Gotmare AD, Bui NDQ, Li JN, Hoi SCH. CodeT5+: Open code large language models for code understanding and generation. In: Proc. of 2023 Conf. on Empirical Methods in Natural Language Processing. Singapore: Association for Computational Linguistics, 2023. 1069–1088. [doi: 10.18653/v1/2023.emnlp-main.68]

[32] Nijkamp E, Hayashi H, Xiong CM, Savarese S, Zhou YB. CodeGen2: Lessons for training LLMs on programming and natural languages. arXiv preprint arXiv:2305.02309, 2023. [doi: 10.48550/arXiv.2305.02309]

[33] Sennrich R, Haddow B, Birch A. Neural machine translation of rare words with subword units. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics. Berlin: Association for Computational Linguistics, 2016. 1715–1725. [doi: 10.18653/v1/p16-1162]

[34] Lin GJ, Wen S, Han QL, Zhang J, Xiang Y. Software vulnerability detection using deep neural networks: A survey. Proceedings of the IEEE, 2020, 108(10): 1825–1848. [doi: 10.1109/jproc.2020.2993293]

[35] 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 2018 17th IEEE Int'l Conf. on Machine Learning and Applications. Orlando: IEEE, 2018. 757–762. [doi: 10.1109/ICMLA.2018.00120]

[36] Facebook. Infer. https://fbinfer.com/

[37] Wang CZ, Yang YH, Gao CY, Peng Y, Zhang HY, Lyu MR. No more fine-tuning? An experimental evaluation of prompt tuning in code intelligence. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Singapore: ACM, 2022. 382–394. [doi: 10.1145/3540250.3549113]

[38] Liu PF, Yuan WZ, Fu JL, Jiang ZB, Hayashi H, Neubig G. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Computing Surveys, 2023, 55(9): 195. [doi: 10.1145/3560815]

[39] Lin B, Wang SW, Liu ZX, Liu YP, Xia X, Mao XG. CCT5: A code-change-oriented pre-trained model. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 1509–1521. [doi: 10.1145/3611643.3616339]

[40] Li ZY, Lu S, Guo DY, Duan N, Jannu S, Jenks G, Majumder D, Green J, Svyatkovskiy A, Fu SY, Sundaresan N. Automating code review activities by large-scale pre-training. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Singapore: ACM, 2022. 1035–1047. [doi: 10.1145/3540250.3549081]

[41] Tufano R, Masiero S, Mastropaolo A, Pascarella L, Poshyvanyk D, Bavota G. Using pre-trained models to boost code review automation. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 2291–2302. [doi: 10.1145/3510003.3510621]

[42] Zhou X, Kim K, Xu BW, Han D, He J, Lo D. Generation-based code review automation: How far are we? arXiv preprint arXiv: 2303.07221, 2023. [doi: 10.48550/arXiv.2303.07221]

[43] Mikolov T, Sutskever I, Chen K, Corrado G, Dean J. Distributed representations of words and phrases and their compositionality. In: Proc. of the 26th Int'l Conf. on Neural Information Processing Systems. Lake Tahoe: ACM, 2013. 3111–3119. [doi: 10.5555/2999792.2999959]

[44] Shaw P, Uszkoreit J, Vaswani A. Self-attention with relative position representations. In: Proc. of 2018 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. New Orleans: Association for Computational Linguistics, 2018. 464–468. [doi: 10.18653/v1/n18-2074]

[45] Fan JH, Li Y, Wang SH, Nguyen TN. A C/C++ code vulnerability dataset with code changes and CVE summaries. In: Proc. of the 17th Int'l Conf. on Mining Software Repositories. Seoul Republic of Korea: ACM, 2020. 508–512. [doi: 10.1145/3379597.3387501]

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

[47] Zou DM, Liang JJ, Xiong YF, Ernst MD, Zhang L. An empirical study of fault localization families and their combinations. IEEE Transactions on Software Engineering, 2021, 47(2): 332–347. [doi: 10.1109/tse.2019.2892102]

[48] Li X, Li W, Zhang YQ, Zhang LM. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 169–180. [doi: 10.1145/3293882.3330574]

[49] Zhang MS, Li X, Zhang LM, Khurshid S. Boosting spectrum-based fault localization using pagerank. In: Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Santa Barbara: ACM, 2017. 261–272. [doi: 10.1145/3092703.3092731]

[50] Sohn J, Yoo S. FLUCCS: Using code and change metrics to improve fault localization. In: Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Santa Barbara: ACM, 2017. 273–283. [doi: 10.1145/3092703.3092717]

[51] Lou YL, Ghanbari A, Li X, Zhang LM, Zhang HT, Hao D, Zhang L. Can automated program repair refine fault localization? A unified debugging approach. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2020. 75–87. [doi: 10.1145/3395363.3397351]

[52] Saha RK, Lyu YJ, Yoshida H, Prasad MR. Elixir: Effective object-oriented program repair. In: Proc. of the 2017 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering. Urbana: ACM, 2017. 648–659. [doi: 10.1109/ase.2017.8115675]

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

[54] Ren ZL, Jiang H, Xuan JF, Yang ZJ. Automated localization for unreproducible builds. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM, 2018. 71–81. [doi: 10.1145/3180155.3180224]

[55] Li DC, Wu CS, Tsai TI, Lina YS. Using mega-trend-diffusion and artificial samples in small data set learning for early flexible manufacturing system scheduling knowledge. Computers & Operations Research, 2007, 34(4): 966–982. [doi: 10.1016/j.cor.2005.05.019]

[56] Kim D, Nam J, Song J, Kim S. Automatic patch generation learned from human-written patches. In: Proc. of the 2013 35th Int'l Conf. on Software Engineering. San Francisco: IEEE, 2013. 802–811. [doi: 10.1109/icse.2013.6606626]

[57] Xin Q, Reiss SP. Leveraging syntax-related code for automated program repair. In: Proc. of the 2017 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering. Urbana: ACM, 2017. 660–670. [doi: 10.1109/ase.2017.8115676]

[58] Le XBD, Chu DH, Lo D, Le Goues C, Visser W. S3: Syntax-and semantic-guided repair synthesis via programming by examples. In: Proc. of the 2017 11th Joint Meeting on Foundations of Software Engineering. Paderborn: ACM, 2017. 593–604. [doi: 10.1145/3106237.3106309]

[59] Le XBD, Lo D, Le Goues C. History driven program repair. In: Proc. of the 2016 IEEE 23rd Int'l Conf. on Software Analysis, Evolution, and Reengineering. Osaka: ACM, 2016. 213–224. [doi: 10.1109/saner.2016.76]

[60] Hua JR, Zhang MS, Wang KY, Khurshid S. Towards practical program repair with on-demand candidate generation. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM, 2018. 12–23. [doi: 10.1145/3180155.3180245]

附中文参考文献:

[7] 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. [doi: 10.13328/j.cnki.jos.006828]

林博(1996－), 男, 博士生, 主要研究领域为智能化软件工程, 软件维护与演化.

毛晓光(1970－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为高可信软件技术.

王尚文(1994－), 男, 博士, 助理研究员, 主要研究领域为智能化软件工程, 软件维护与演化.