计算机研究与发展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(4): 868−883, 2026

融合静态分析与大语言模型的非连续代码重构方法

嵇友晴[1] 张迎周[1,2] 苏玉鹏[1] 王 刚[1] 张文智[1] 谢金言[1]

[1](南京邮电大学计算机学院、软件学院、网络空间安全学院 南京 210023)

[2](藏语智能全国重点实验室(南京邮电大学) 南京 210023)

(1224045801@njupt.edu.cn)

Non-Contiguous Code Refactoring: A Hybrid Approach of Static Analysis and Large Language Model

Ji Youqing[1], Zhang Yingzhou[1,2], Su Yupeng[1], Wang Gang[1], Zhang Wenzhi[1], and Xie Jinyan[1]

[1](*School of Computer Science*, *School of Software*, *School of Cyberspace Security*, *Nanjing University of Posts and Telecommunications*, *Nanjing* 210023)

[2](*State Key Laboratory of Tibetan Language Intelligence* (*Nanjing University of Posts and Telecommunications*), *Nanjing* 210023)

Abstract The widespread adoption of large language model (LLM) in software engineering has made automated code refactoring, leveraging their powerful code comprehension and generation capabilities, a crucial direction for enhancing software quality and development efficiency. However, when refactoring non-contiguous code clones which arise from statement interleaving, reordering, and similar transformations, LLM faces core challenges: dispersed semantic context, difficulty in capturing critical dependencies, and susceptibility to "hallucination" errors. To address these challenges, we propose a novel method for non-contiguous code clone refactoring that integrates static analysis with LLM. Our method first efficiently and accurately identifies non-contiguous clones by combining program slicing with an algebraic classifier. Next, a context-aware refactoring opportunity identification algorithm determines the optimal refactoring targets for the LLM. Finally, a chain-of-thought few-shot prompting strategy guides the LLM to generate high-quality "extract method" refactoring suggestions, and a verification mechanism, inspired by metamorphic relations, validates the semantic and structural consistency of the generated results. Experiments on the open-source datasets Google Code Jam and BigCloneBench demonstrate that our proposed refactoring method reduces clone code by 66% to 71% in real-world projects like Junit. Furthermore, our detection method achieves an *F*1-score 2% to 18% higher than existing mainstream tools. On the Community Corpus-A refactoring opportunity identification benchmark, it reaches an *F*1-score of 0.415, surpassing the state-of-the-art tool GEMS by 7.5%, enhancing software quality.

Key words large language models; static analysis; non-contiguous code clone; code refactoring; prompt engineering

摘 要 随着大语言模型(LLM)在软件工程领域的广泛应用,通过其强大的代码理解与生成能力进行自动化代码重构,已成为提升软件质量与开发效率的关键方向。然而,对于由语句交错、重排等导致的非连续代码克隆,LLM在重构时面临着语义上下文分散、关键依赖捕捉困难以及易产生"幻觉"错误等核心挑战。为应对这些挑战,提出了一种融合静态分析与LLM的非连续代码克隆重构方法。该方法首先通过结合程序切片与代数分类器,高效精准地识别非连续克隆;然后,通过一种基于上下文信息的重构机会识别算法,为LLM确定最佳重构目标;最后,利用思维链少样本提示策略引导LLM生成高质量的"提取

收稿日期: 2025-09-21; **修回日期**: 2025-12-31

通信作者: 张迎周(zhangyz@njupt.edu.cn)

函数”重构建议，并利用蜕变关系验证机制，对生成结果进行语义和结构一致性验证。所提出的重构方法在 Junit 等真实项目中减少了 66%~71% 的克隆代码。此外，在开源数据集 Google Code Jam 和 BigCloneBench 上的实验表明，所提出的检测方法 *F*1 值较现有主流工具提升了 2%~18%，在 Community Corpus-A 重构机会识别基准上，*F*1 值达到了 0.415，超越先进工具 GEMS 7.5%，提升了软件质量。

关键词 大语言模型；静态分析；非连续代码克隆；代码重构；提示工程

中图法分类号 TP311

DOI: 10.7544/issn1000-1239.202550688 **CSTR:** 32373.14.issn1000-1239.202550688

在软件开发过程中，由于设计不规范或者编码实践不当往往会引入低质量的代码，即“代码异味”[1]。这类代码往往会增加软件开发的技术债务，其中“代码低内聚”是最典型的异味类型之一。代码低内聚的后果是增加代码重构的风险，引发不易察觉的错误，降低代码的健壮性和软件的可读性，提高维护成本。因此，学界有“提取方法”[2]“抽取类”[3]“死代码消除”[4]等方法来进行代码重构，提高代码的内聚性。

近年来随着大语言模型(large language model, LLM)的兴起，学者们也开始基于 LLM 进行代码重构，利用 LLM 的代码理解能力，为实现高度自动化开发提供了可能。然而，在复杂软件系统中，大量的克隆并非是简单的连续代码块，而是因功能演进、并发逻辑或者开发者修改而形成的非连续代码克隆[5]。这类克隆在功能上高度内聚，但在代码文本上却被不相关的语句分隔，这对自动化分析工具构成了严峻挑战[6]。相关实践表明，直接将 LLM 应用于此类复杂的结构化重构任务，暴露了当前人工智能驱动代码重构研究的局限性。LLM 缺乏对代码逻辑的深度理解，无法准确捕捉上下文依赖，并且 LLM 存在幻觉副作用，会生成不准确或者不一致的代码[7-9]。为阐述 LLM 在处理非连续代码克隆重构时面临的挑战，下面给出一个用户行为埋点上报场景下的实例，如图 1 所示。

图 1(a)中 trackPageView 和 trackButtonClick 方法里展示了非连续代码克隆的典型困境，本应连贯核心埋点逻辑(如上报 URL 与上报操作内容)，被日志记录、权限校验等这类无关代码分隔。2 段核心逻辑分布在不连续的位置，从而形成了非连续代码克隆块。理想的自动化重构工具应能精准识别非连续核心代码逻辑，洞察到这 2 处分散的埋点逻辑本质上是可复用的单元，进而将它们提取封装为图 1(b)的 logTrackingEvent 函数，把零散的克隆片段重新整合，让代码回归简洁且高内聚的结构。尽管 LLM 提供了理解代码业务逻辑的可能性，但是直接将 LLM

```java
public class TrackingService {
    // 记录页面访问行为
    public void trackPageView(String pageUrl, String userAction) {
        // 1. 前置校验，防止无效埋点
        validateUrl(pageUrl);
        // 克隆片段 1，上报 URL
        System.out.println("上报埋点：URL=" + pageUrl);
        // 无关代码，模拟业务侧插入的日志、权限判断
        logDebug( eventType: "pageView", pageUrl);
        checkPermission(userAction);
        // 克隆片段 2，上报操作内容
        System.out.println("上报埋点：操作=" + userAction);
        // 2. 收尾操作，标记埋点已处理
        markAsTracked(pageUrl, userAction);
    }
    // 记录按钮点击行为
    public void trackButtonClick(String pageUrl, String buttonId) {
        // 1. 前置校验，防止无效埋点
        validateUrl(pageUrl);
        // 克隆片段 1，上报 URL
        System.out.println("上报埋点：URL=" + pageUrl);
        // 无关代码，模拟业务侧插入的日志、权限判断
        logDebug( eventType: "buttonClick", pageUrl);
        checkPermission(buttonId);
        // 克隆片段 2，上报操作内容
        System.out.println("上报埋点：按钮ID=" + buttonId);
        // 2. 收尾操作，标记埋点已处理
        markAsTracked(pageUrl, buttonId);
    }
    // 以下为辅助方法（省略具体实现）
    private void validateUrl(String url) { /*... */ }
    private void logDebug(String eventType, String url) { /*... */ }
    private void checkPermission(String action) { /*... */ }
    private void markAsTracked(String url, String action) { /*... */ }
}
```

(a) 原代码

```java
public class RefactoredTrackingService {
    public void trackPageView(String pageUrl, String userAction) {
        validateUrl(pageUrl);
        // 调用提取的通用方法
        logTrackingEvent(pageUrl, userAction);
        markAsTracked(pageUrl, userAction);
    }
    public void trackButtonClick(String pageUrl, String buttonId) {
        validateUrl(pageUrl);
        // 调用提取的通用方法
        logTrackingEvent(pageUrl, buttonId);
        markAsTracked(pageUrl, buttonId);
    }
    // 提取的通用埋点方法，聚合原本分散的核心逻辑
    private void logTrackingEvent(String pageUrl, String action) {
        System.out.println("上报埋点：URL=" + pageUrl);
        // 模拟无关代码插入场景
        logDebug( eventType: "trackEvent", pageUrl);
        checkPermission(action);
        System.out.println("上报埋点：操作=" + action);
    }
    // 辅助方法保持不变（省略重复代码）
    private void validateUrl(String url) { /*... */ }
    private void logDebug(String eventType, String url) { /*... */ }
    private void checkPermission(String action) { /*... */ }
    private void markAsTracked(String url, String action) { /*... */ }
}
```

(b) 理想的重构代码

Fig. 1 An example of non-contiguous code clone refactoring

图 1 非连续代码克隆重构示例

应用到代码重构问题上,还存在挑战:

1)碎片化语义上下文理解失效问题。非连续代码克隆导致核心语义分布在不连续的位置。由于LLM在处理长距离依赖时存在衰减效应,这导致模型很难跨越那些不相关的代码。这种特点使得LLM难以有效地将克隆对的内在逻辑关联起来,并将其视为统一可提取的整体。从某种程度上而言,这形成了结构化分析的盲区。

2)结构化分析缺失问题。LLM缺乏对程序控制流和数据流等结构化信息的显式表征和推理能力。这表明,LLM难以精确判断克隆对是否构成可提取的“高内聚、低耦合”单元。在LLM缺乏这种能力的情况下,开发者不得不手动指定重构范围,这无疑限制了重构的自动化程度。

3)生成幻觉与语义保持性缺失问题。LLM的幻觉副作用可能导致生成的代码语法正确,但语义存在错误(如改变原代码业务逻辑)。其破坏了重构任务的语义保持核心原则。

针对上述挑战,本文的核心思想是融合静态分析与LLM。由于观察到业务逻辑相近的代码一定是程序切片的子集,因此采用程序切片引导的思维链(chain of thought, CoT)技术,指导LLM识别完整的业务逻辑。此外,还对LLM生成的重构代码进行蜕变关系验证,提出语义与结构一致性的蜕变关系,由此缓解LLM的幻觉问题。

为此,本文提出一种融合静态分析与LLM的非连续代码克隆重构方法,通过程序切片技术穿透文本表象、聚合语义相关的代码,再以CoT提示和蜕变关系引导并验证LLM的生成过程以解决上述挑战。本文以Java为目标语言实现了原型工具,并在多个公开基准数据集与真实开源项目上进行实验评估。实验结果表明,本文方法在检测效率、重构机会识别准确率以及最终的软件质量改进效果上均有所提升。在重构机会识别基准上,$F1$值达到0.415,超越先进工具GEMS 7.5%。本文的贡献有3点:

1)提出一种高效精确的非连续代码克隆检测方法。结合代数分类器对功能性代码片段进行分类,将克隆检测转化为高效的分类任务,提升检测效率。

2)设计基于多维上下文信息的重构机会识别算法。综合多维上下文信息判断克隆对是否合适“提取函数”重构。

3)提出基于蜕变关系的自动化验证机制。将传统蜕变测试思想用于代码重构验证领域,通过双阈值机制验证,过滤错误重构以确保重构的可靠性。

1 相关工作

1.1 代码克隆重构技术

代码重构是软件维护领域的经典研究方向。其核心目标是在不改变其外部行为的前提下,改进现有代码内部结构,目的是为了识别和修改软件代码中重复或相似的代码片段,以提高代码质量和可维护性。

现有许多学者在代码重构领域做出了许多重要贡献。Tsantalis等人[10]通过分析克隆代码的程序依赖图,来确定给定克隆代码能否被安全重构。他们通过检测并参数化克隆代码的差异点来完成重构,取得了较好的效果。Barbosa等人[11]使用4种规则重构克隆代码,并在开源软件JHotDraw上进行了实验,结果表明,基于规则的重构方法也能有效地移除软件中的克隆代码。而这类方法通常会定义一组重构规则,并在克隆代码满足特定条件时,应用相应的规则进行重构。Zibran等人[12]在克隆重构的研究中,回答了7个相关问题,他们发现克隆规模对克隆重构并没有重要的影响,同时在软件早期的版本中重构会更为频繁地发生,需要重构的克隆块在重构之前往往是较为稳定的代码。传统的“提取函数”重构工具,例如JDeodorant[13],同样利用程序切片技术来识别可提取的代码片段。它能够处理部分不连续的语句,为自动化重构提供了重要支持。然而,当克隆片段间存在高耦合、功能交织时,JDeodorant在进行代码块映射时容易出错,导致重构建议不准确或不完整。

最近,也有一些研究开始探索新的方向。肖泉彬等人[14]提出了一种基于代码克隆差异分析的函数模板挖掘和检索方法,旨在为开发者提供“检索与推荐”,而非直接对现有代码进行自动化重构与优化。他们通过拼接片段级克隆与差异分析来构建高质量函数模板,解决了传统方法模板碎片化问题。但与本文工作不同的是,该方法未引入LLM,因此在处理Type-3等存在复杂语义差异的克隆片段时能力受限。Xiao等人[15]提出了一种基于漏洞签名与补丁签名结合的克隆检测方法,用于识别代码中的重复性漏洞。尽管这种方法通过程序切片和语义分析来减少误报与漏报,但是对代码修改的敏感性存在不足,尤其是在检测经过内部修改的代码或语义等效但语法差异较大的代码时效果不佳。此外,Li等人[16]提出一种

基于细粒度补丁增强的抽象语法树级签名的克隆检测方法。该方法主要用于识别未修补的第三方 API，它通过分析漏洞修复补丁前后的 AST 差异，提取关键漏洞相关元素，用以生成细粒度的签名。虽然其抽象语法树级的分析比语句级更精细，但也可能忽略某些语义层面的相似性，而且对于经过复杂重构或者混淆的代码，AST 结构可能发生较大变化，导致检测失效。同时，由于其需要进行 AST 差异提取和漏洞追踪，该方法的计算复杂度也相对较高。Song 等人[17]提出一种基于程序切片的漏洞代码克隆检测方法，其优势在于通过结合程序切片与哈希匹配，实现了细粒度的 Type-3 克隆检测。但是其检测范围仅限于对于函数级别的 Type-1/2/3 的克隆，缺乏对 Type-4 克隆的检测能力。

综上所述，传统方法在处理非连续代码克隆时，常常需要在效率与精度之间做权衡。虽然已有研究探索了片段组合的可能性，但尚未形成一个既能高效检测又能对复杂语义差异进行自动化重构的系统性解决方案。

1.2 LLM 在软件工程中的应用

LLM 在软件工程领域正扮演着越来越重要的重要角色。其中，自动代码生成是 LLM 最常见的应用之一[18-20]。例如，Wermelinger[21]利用 Copilot 根据上下文提示解决编程问题，极大地提升了开发效率。在程序修复方面，相关学者也发现 LLM 能够根据错误信息自动生成补丁[22]，同时也有研究利用 LLM 进行代码重构任务。徐子懋等人[23]提出一种名为 Lsplitter 的方法，旨在解决“长方法”代码异味。他们首次结合启发式规则与 LLM 来解决长方法分解难题，并且还提出基于位置的相似方法合并算法，以优化 LLM 输出。这充分验证了 LLM 在代码理解和分解任务上的有效性，但也指出了 LLM 在进行长方法分解时会拆分出相似的方法，需要进一步合并冗余。Siddeeq 等人[24]则通过基于 LLM 的多智能体系统，实现了对 Haskell 代码的自动化重构。其研究表明，这种方法能让程序在代码质量和性能效率方面均有所提高，进一步证实了 LLM 在重构任务方面的有效性与可行性。

然而，现有利用 LLM 进行代码重构的研究主要集中在连续代码克隆。如前文所述，语义的碎片化和非局部依赖使得 LLM 难以直接识别和处理。受到 Lsplitter“先 LLM 生成，后处理修正”的模式启发，单纯依赖 LLM 进行端到端的重构是不可靠的，必须设计相应的协同框架。本文则采用“静态分析预处理，LLM 引导生成”的框架，通过静态分析来利用程序切片和上下文信息重构机会识别，先行解决 LLM 结构识别问题，为 LLM 明确重构任务，进而从根源上减少其产生不合理拆分或错误重构的可能性。

1.3 自动化重构的验证与评估

确保重构操作的正确性，保持程序外部行为不变是自动化重构的前提。目前对于重构验证与评估的方法大致能分为 3 类：基于约束、基于测试与基于深度学习的方法。

基于程序的形式化语义（如公理语义、操作语义）通过逻辑推理证明重构前后程序的行为等价性。典型方法包括利用定理证明器（如 Coq[25]、Isabelle[26]）构建重构前后代码的语义模型以证明两者在所有可能输入下的输出一致，验证重构后的模型是否满足与原模型相同的属性。其优势是能提供严格的正确性保证，适用于对可靠性要求极高的系统，但局限性在于对复杂程序的建模难度大、自动化程度低，且计算成本高，难以应用于大规模工业级项目。

基于约束的重构验证方法，例如 KLEE[27]。该方法通过监控程序在实际运行中的动态行为，对比重构前后的执行轨迹以判断一致性。常见技术包括符号执行和跟踪等价性检查。符号执行通过符号化输入遍历程序的所有可能执行路径，并记录重构前后路径上的约束条件以验证约束是否等价。而跟踪等价性检查则是在相同输入下对比重构前后程序的执行轨迹，若轨迹一致则认为重构正确。这类方法的优点在于能捕捉程序运行时的动态依赖，弥补静态分析对复杂上下文处理的不足。但局限性也十分明显，比如存在路径爆炸问题，使其难以覆盖所有可能的执行路径，其次对输入的选择依赖性也很强。

随着深度学习在软件工程中的应用，基于预训练模型的代码表示学习方法也用于重构验证[28]。典型的模型有 CodeBERT（捕捉语义信息）、TreeBERT（捕捉语法结构信息）等。其优势是自动化程度高，能快速处理大规模代码，特别适合与 LLM 结合的重构场景，但它也有一定局限性，即依赖预训练模型的表示能力。向量相似度高并不代表逻辑完全等价，因为可能存在“假阳性”结果。

此外，验证重构任务还能通过执行现有测试套件，评估重构后程序的行为一致性。其核心思想是：若重构后的代码能通过所有原测试用例，那么就认为重构未破坏程序功能[29]。这种方法优势在于简单易行，可直接利用软件项目中已有的测试资源，适用于各类重构场景。不过，这种方法也存在明显不足：

它的有效性完全依赖测试用例的覆盖率。若测试不充分,就可能无法发现重构引入的潜在错误。

然而,为复杂的代码重构提供形式化的正确性证明是极其困难的,这便引出了软件测试中的“测试预言机问题”(test oracle problem, TOP)。不过,蜕变测试(metamorphic testing, MT)可以有效缓解预言机问题。它不依赖于单次执行的预期输出,而是通过检查程序执行的输入与输出之间是否满足某种预定义的蜕变关系(metamorphic relation, MR)来判断程序是否正确[30]。在代码重构领域,蜕变关系为验证重构的语义与结构保持性提供了强有力的理论框架。近期,Xue 等人[31]提出了一种名为 MT-LAPR 的蜕变测试框架用于评估和改进自动程序修复技术的稳定性。他们证实了蜕变测试能够有效提升基于 LLM 的自动程序修复的健壮性。同时,孙昌爱等人[32]对基于路径分析的蜕变测试生成技术进行了深入研究,并指出蜕变关系的定义和测试用例的生成对蜕变测试的有效性至关重要。

传统的蜕变关系通常基于程序的输入输出,而对于代码本身,其内在的语义和结构信息也可以作为蜕变关系的来源。因此,基于代码表示的蜕变关系为缓解 LLM 的幻觉问题提供了可行思路,能将其引入自动化重构领域。

2 非连续代码克隆的高效检测方法

本文首先详细阐述作如何高效地检测非连续代码克隆。该阶段的成功是后续所有步骤的基础,其核心目标是解决传统方法在面对语义复杂性和语句交错时遇到的难题。

传统的图匹配方法虽然可以识别非连续代码克隆,但是其时空复杂度高、计算效率低,难以应用于大规模软件项目中。为解决这一问题,本文提出一种非连续代码克隆的高效检测方法。该方法的核心在于,通过程序切片将复杂的图匹配问题转换为功能性代码片段相似性比对,以便利用代数分类器(algebraic classifiers)[33]从功能性代码片段提取到特征向量,进而进行高效的机器学习分类任务,提升非连续代码克隆的检测效率。

2.1 基于程序切片的功能性代码片段提取

程序切片是一种强大的程序分析技术,它能够从程序中提取出所有对某个特定计算点(也称切片标准)有影响的语句集合。由于切片是基于控制流和数据流依赖进行的,所以无论它们在物理上是否连续,也能够天然地聚合功能相关的语句,这使其成为提取非连续功能性代码片段的理想工具。

为了精确提取与功能相关的代码语句,本文采用了基于程序切片的策略,选择 WALA(Watson libraries for analysis)作为静态分析引擎。这主要是因为它提供了强大的数据流分析能力,并且还能生成一种结构化的中间表示(intermediate representation, IR),便于定义和实现切片规则。本文以 Java 程序为例,通过利用 WALA 静态分析框架将 Java 字节码转换为一种静态单赋值(static single assignment, SSA)形式的中间表示,并基于 WALA IR 设计了一套规则。利用前向数据流分析自动核心变量,并以此形成核心变量集。如表 1 所示,在 WALA IR 中,主要关注 5 类对内存和控制流有关键影响的语句,分别是 returnStmt(返回语句)、printStmt(打印语句)、getStaticStmt(全局变量获取语句)、invokeStmt(函数调用语句)、assignStmt(赋值语句)。

Table 1 WALA IR Instructions Abstract

表 1 WALA IR 指令抽象

语句类型	IR 指令抽象
returnStmt	return <%v>
printStmt	getstatic <system, out, io> %v
getStaticStmt	%v = getstatic <global, type>
invokeStmt	Invoke <class, func(I) V> %v, %x
assignStmt	%v =op %x, %y

具体地,对于不同的语句类型,将相应 IR 指令抽象中的目标值加入到当前上下文的核心变量集合之中。例如,对于全局变量获取语句,需要根据其 IR 指令将目标全局变量及其位置加入到全局变量集合;而对于赋值语句,由于 WALA IR 的单静态赋值特点导致全局变量会赋值给普通变量,所以需要更新替换涉及全局变量赋值或计算的相应核心变量集合;对于返回语句、打印语句与函数调用语句,同理,通过不断更新其上下文相关的核心变量集,为检测功能性片段提供依据。

为了后续检测非连续代码克隆对,提出了非连续克隆功能性代码片段提取算法,如算法 1 所示。针对程序生成其逆拓扑函数调用图,通过遍历函数调用图,根据所设计的数据流方程获取程序核心变量集,并且进一步获取过程内切片。最后,收集所有切片形成功能性代码切片集合。该集合将作为后续进行克隆对检测的前提。其中,核心变量集 OTV 指的是对函数外部状态产生直接影响的变量集合,一

个功能性代码片段OVF是一个与某个核心变量的计算直接相关的语句集合。

算法 1. 非连续克隆功能性代码片段提取算法。

输入：项目P；

输出：功能性代码片段集合OVF。

① 初始化集合OVF；

② 利用 WALA 将程序解析为IR，并构建程序控制流图CFG，函数调用图CG及其逆拓扑排序函数集$FunctionSet$；

③ while 函数集$FunctionSet$不为空 do

④ 基于前向数据流分析获取核心变量集 OTV；

⑤ while 核心变量集OTV不为空 do

⑥ 获取过程内切片$TSlice_{\text{var}}$；

⑦ 更新切片准则$TSlice_{\text{var}}$直至无函数调用语句；

⑧ end while

⑨ $OVF \leftarrow TSlice_{\text{var}} \cup OVF$；

⑩ end while

2.2 功能性代码片段的特征提取

本文所提出的方法利用程序切片将功能上内聚的语句聚合在一起，从而形成了功能性代码片段。然而，直接对这些代码片段进行文本或语法树层面的比对，对于诸如变量重命名、微小的语句调整等语义无关的变动会十分敏感，容易导致漏报。为了实现一种更具鲁棒性、更能抵抗语法噪音的相似性度量方法，本研究并未直接比较代码本身。反而是为其设计了能刻画内在规模、复杂度与结构特性的多维度特征向量。这种从代码到特征的抽象，正是实现高精度克隆识别的关键一步。

由于相似性匹配易受到代码片段的命名、结构等变化影响，故算法 1 提取到的语句集合功能性代码片段并不适合作为相似性匹配的对象。为此需要提取功能性代码片段的特征作为克隆相似度的匹配对象，以提高克隆检测的准确性。语义特征构建过程的前提是构建抽象语法树（abstract syntax tree, AST），并在其基础上进行相应信息获取，而 JavaParser 库能够将 Java 源码解析为 AST，进而可以利用 AST 来追踪引用代码片段对应的声明。所以，本文在功能性代码片段的 AST 上进行度量指标的收集。

本文借鉴开源软件质量观察站的度量研究[34]，从功能性代码片段提取了 24 个维度特征，涵盖了规模、复杂度和结构 3 个方面，这些特征包括：

1）规模度量（10 个）：代码行数、声明变量数量、被使用的变量数量、参数数量、操作符数量、操作数数量、表达式数量、字符串常量个数、数值常量个数、赋值语句数量。

2）复杂度度量（10 个）：圈复杂度、Halstead 困难度、Halstead 努力度、Halstead 困难度+努力度、return 语句个数、数值运算符数量、最大嵌套块数、循环个数（for/while）、比较运算符数量、try/catch 数量。

3）结构度量（4 个）：本地方法被调用数量、外部方法被调用数量、类使用数量、类转换数量。

对于任意待比较代码片段(OVF_1, OVF_2)的度量特征，用 2 个 24 维特征向量拼接成 48 维的向量$\boldsymbol{V} = [\boldsymbol{V}_1, \boldsymbol{V}_2]$，归一化后作为后续分类器的输入。

2.3 基于代数分类器的非连续代码克隆检测

在将每个潜在的克隆对转化为定量的 48 维特征向量后，原始复杂的代码克隆检测问题便在形式上被转化为一个经典的机器学习二分类任务，即判断该向量所代表的代码对是否构成克隆。考虑到工业级项目中需要处理海量的代码对组合，分类器的训练和推理效率至关重要。为此，本文的检测采用了代数分类器[33]。代数分类器是传统机器学习分类器模型在数学结构上的更改，其核心优势在于，它为机器学习模型赋予了群同态的代数结构。这使得模型的训练过程可以通过批量的群运算来高效完成。更重要的是，相比于底层学习器（如朴素贝叶斯、决策树等）采用传统k折交叉验证的实现，代数分类器通过群同态的代数结构，可以在不牺牲底层学习器固有分类精度的前提下，将验证时间复杂度从传统的$O(k \times n)$优化到$O(k+n)$，其中n是样本数。这种效率上的巨大提升，使得在大规模数据集上也能进行高效的训练和模型验证。

算法 2. 非连续克隆代码检测算法。

输入：项目；

输出：非连续克隆对集合$Pairs$。

① 初始化特征向量$\boldsymbol{V}$以及非连续克隆对集合$Pairs$；

② 提取功能性代码片段集合OVF；

③ for $(OVF_i, OVF_j) \wedge i \neq j$ in OVF do

④ 提取不同功能性代码片段的特征向量$\boldsymbol{V}$；

⑤ end for

⑥ for $(\boldsymbol{V}_1, \boldsymbol{V}_2)$ in $\boldsymbol{V}$ do

⑦ 获取 HLearn 代数分类器分类结果$isClone$；

⑧ if $isClone$ = True do

⑨ 更新非连续克隆对集合$Pairs$；

⑩ end if

⑪ end for

如算法 2 所示，行①通过算法 1 获取到功能性代码片段，然后对所有功能性片段提取特征向量，并利用 HLearn 代数分类器[35]进行代码克隆检测的二分类，再根据分类结果获取程序的所有克隆对。其中，非连续克隆对 $Pairs=\{(C_i,C_j)\}$，C_i和C_j分别为源代码L中 2 个或多个代码片段CF的集合。具体地，$C_i=\underset{i\in k}{\cup}CF_i$，$C_j=\underset{j\in k}{\cup}CF_j$。它们是程序逻辑功能与结构相似的非连续代码片段，其作为潜在的克隆对在实现细节上却存在差异。因此，高效且准确地识别出潜在克隆对是必要的。

通过上述检测框架，已然能够从大量源码中高效、精准地识别出潜在的非连续克隆对。然而，基于该框架检测出的克隆并非都具备同等的重构价值或可行性。一个在技术上完全可提取的片段，在软件工程实践中，可能因破坏封装性或引入不必要的复杂接口而导致损失。因此，关键问题在于如何从这些候选者中筛选出真正值得重构的机会。

3 非连续代码克隆的自动化重构方法

在精准识别出非连续克隆对的基础上，利用 LLM 程序进行自动化“提取函数”重构。本文提出方法的核心在于充分利用而非盲从 LLM。该框架通过结合静态分析预处理与引导 LLM 生成完成非连续代码克隆的自动化重构。

本文提出的总体框架图如图 2 所示。通过非连续代码克隆的高效检测方法获取非连续代码克隆对，基于上下文信息的重构机会识别，收集到重构对集合。然后根据重构示例，结合 CoT 的少样本提示引导 LLM 进行提取函数建议生成，并且验证生成的重构建议是否违背语义与结构一致性蜕变关系，若不违背蜕变关系则执行重构，输出重构代码；反之重新根据引导 LLM 得到的重构建议进行蜕变关系验证，直至验证通过。该框架通过结合静态分析预处理与引导 LLM 生成完成非连续代码克隆的自动化重构。

3.1 基于上下文信息的重构机会识别

在重构任务中，并非所有检测到的代码克隆都值得进行重构。传统方法常常仅依赖内聚、耦合等代码内部度量来判断，而忽略了克隆在软件整体架构中的位置，导致推荐结果脱离实际。为解决 LLM 无法主动、精准判断重构机会的挑战，本文构建了重构机会识别算法，该方法从 3 个维度综合评估一个克隆对的重构价值。

1)继承上下文(relation context, RC)

在面向对象系统中，克隆片段所在的类之间的继承关系直接决定了重构后提取出的新函数的放置位置和可见性。因此，根据克隆对(C_i,C_j)所在类的关系，将其划分为“共同类方法”“共同层级”“共同接

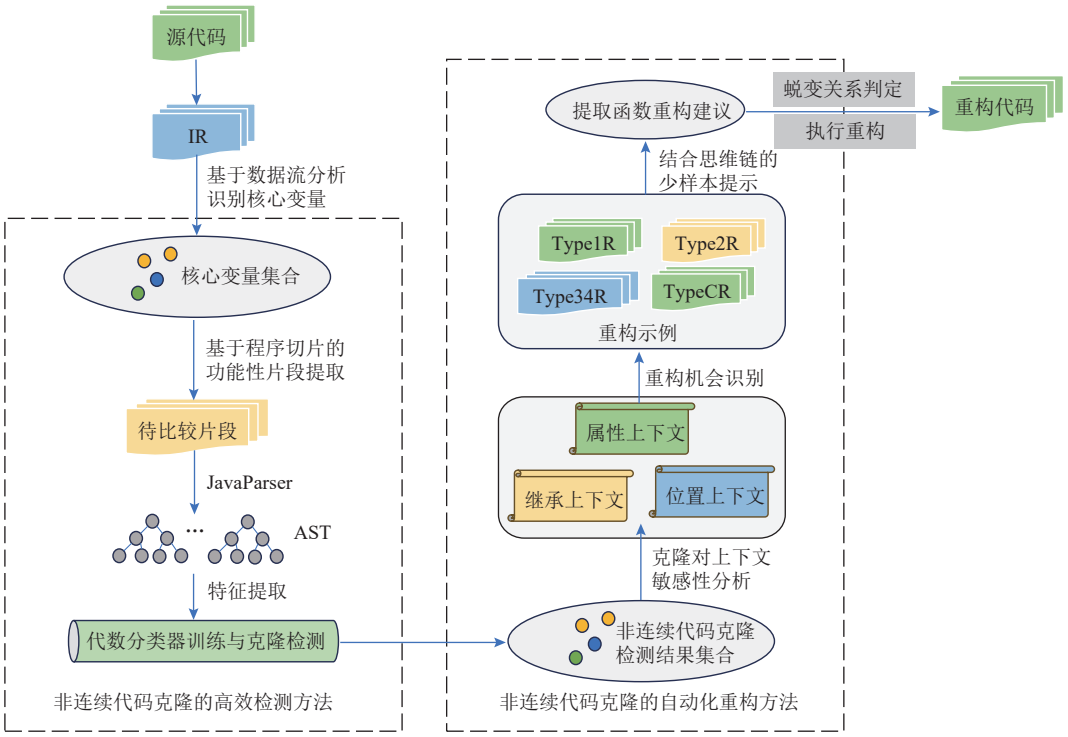


Fig. 2 Non-contiguous code refactoring framework for integrating static analysis and LLMs

图 2 融合静态分析与大语言模型的非连续代码重构框架

口”和“无关”四类，并赋予不同的推荐优先级。“共同类/方法”和“共同层级”的重构成本最低，优先级最高。

2）位置上下文（location context, LC）

克隆片段在代码中的具体位置也影响重构的可行性，如方法体内部、类成员级别。但本文主要关注方法体级别的克隆，因为它们最适合提取函数重构。

3）属性上下文（content context, CC）

克隆片段自身的软件度量特征，如圈复杂度、代码行数等。过于简单或复杂的片段通常不是好的重构候选。

算法 3. 重构机会识别算法。

输入：项目 P；

输出：推荐重构克隆对集合 $SuggestedPairs$。

① 初始化推荐重构克隆对集合 $SuggestedPairs$；

② 获取非连续克隆对集合 $Pairs$；

③ for (C_i,C_j) in $Pairs$ do

④ 对于每个非连续克隆对 (C_i,C_j)，计算其继承上下文维度特征得分 $score_{\text{relation}}$、位置上下文维度特征得分 $score_{\text{location}}$ 以及属性上下文维度得分 $score_{\text{content}}$；

⑤ if $isRefactorable(score_{\text{relation}}, score_{\text{location}}, score_{\text{content}})$ do

⑥ 将非连续克隆对 (C_i,C_j) 列为推荐重构克隆对，并更新推荐重构克隆对集合 $SuggestedPairs$；

⑦ end if

⑧ end for

算法 3 展示了重构机会识别的完整流程。对于每个输入的 $Pairs$，算法分别计算其在 3 个上下文维度上的得分，并最终通过一个决策函数 $isRefactorable$ 来判断其是否值得重构，其中，$isRefactorable$ 决策函数基于随机森林算法实现二分类，综合 3 个维度的特征得分来判断克隆对是否适合进行“提取函数”重构。位置信息相关的特征包括了克隆对的层级，比如方法级、类级以及接口级。与继承信息相关的特征主要分为共同类、共同层级、共同接口以及无关继承。克隆对代码属性特征涵盖代码耦合度、代码复杂度以及方法内聚度等特征。数据集采用了 Community Corpus-A，该数据集包含了由社区专家标注的是否应该进行“提取函数”重构的 Java 方法，为本文提供了训练样本。若克隆的位置、继承和属性上下文均满足推荐重构条件，则将该克隆对加入到最终推荐重构的克隆对集合 $SuggestedPairs$ 中。该算法作为筛选阶段，为后续 LLM 进行重构建议生成奠定了基础。

经过基于上下文信息的机会识别后，此时筛选出了具有潜在重构可能性的克隆对。接着，需要考虑执行自动化重构实例生成，即如何利用 LLM 生成正确且可靠的“提取函数”代码。这不仅是对 LLM 能力的核心考验，也是融合静态分析与 LLM 进行引导式生成与验证框架发挥作用的关键阶段。

3.2 融合 CoT 的精准重构建议生成

在识别出高价值的重构机会后，此时核心挑战转变为如何引导 LLM 生成逻辑正确且语义一致的重构代码。若采用简单的指令响应提示，LLM 虽能模仿形式，却常因缺乏深层逻辑推理而产生参数错误、函数签名不当或副作用丢失等幻觉问题。因此，通过设计基于 CoT 的少样本提示工程策略来解决此问题。

CoT 的核心思想并非直接从 LLM 得到最终代码，而是通过在提示策略的示例中，显式地展示推理过程，从而引导 LLM 模仿人类专家的思维模式。例如，首先分析克隆片段的外部依赖以确定函数参数；然后判断其副作用以决定返回值类型；接着依据其核心功能为其命名；最后生成函数体和重构后的调用代码。通过学习并遵循这种结构化的推理路径，LLM 能够更好地理解重构任务的本质，从而生成更准确可靠的重构建议，有效抑制幻觉的产生。

该提示由 3 部分构成：系统指令、少样本示例和待办任务。

1）系统指令。明确定义 LLM 的角色和任务目标，并规定输出的 JSON 格式。

2）少样本示例。为不同类型的代码克隆（Type-1/2、Type-3/4、非连续克隆）预置高质量的重构示例库。当一个克隆对进入此阶段，系统会动态选择 1~2 个最相似的示例。至关重要的是，每个示例中都通过注释形式嵌入了 CoT，显式地展示了人类专家进行重构时的关键决策过程，例如：

CoT 1：分析克隆片段依赖的外部变量（*url*, *body*），确定它们为新函数的输入参数。

CoT 2：检查克隆片段是否修改外部状态或有返回值。此处无，故新函数返回 *void*。

CoT 3：根据功能（打印详情）为新函数命名为 *printDetails*。

3）待办任务。将待重构的克隆对代码呈现给 LLM。通过学习这些蕴含逻辑推理的 CoT 示例，LLM 能够更好地理解重构的本质，从表面的文本替

换转向深层的逻辑抽象，从而显著提升生成建议的准确性和可靠性，有效缓解 LLM 幻觉。在原型实现中，本研究采用了 GPT-4 模型，为了增强可复现性，如图 3 所示给出了具体的 CoT 提示示例。

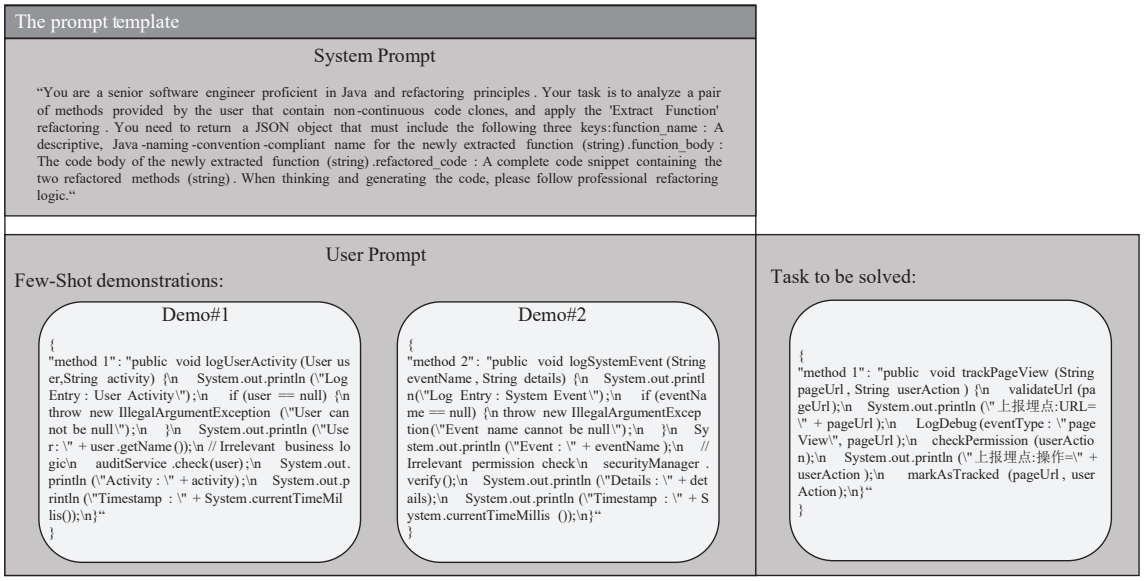


Fig. 3 An Illustration of few-shot prompting template based on CoT

图 3 基于 CoT 的少样本提示模板的示意图

3.3 基于蜕变关系的重构建议一致性验证

为解决 LLM 生成结果不可靠、存在“幻觉”的问题，本研究设计了一套自动化验证机制。传统测试方法依赖测试预言机，但在重构场景下难以构建。传统蜕变测试通过验证多次程序执行的输入输出关系是否满足预定义的蜕变关系来揭示错误。本文所提重构方法将这一思想从程序的输入输出域扩展到程序变换域，即验证重构前后的代码是否满足我们定义的语义与结构一致性蜕变关系。

该方法并非传统意义上的测试过程，因为它不依赖于生成具体的测试用例输入和检查输出，而是直接在代码表示层面上进行验证，也就是将重构操作本身视为一种程序变换，并定义了如下 2 个基于代码表示的蜕变关系，用于验证变换的正确性。

1)语义一致性(MR1)。重构操作不应改变程序的功能语义。采用预训练的代码大模型 CodeBERT 将重构前后的完整函数体编码为高维语义向量。理想的重构应保持语义不变，因此，定义 MR1 为：2 个向量间的余弦相似度高于预设阈值τ_{sem}(0.98)，这构成一个概率性的语义等价预言机。

$$Similarity_{\text{cosine}}(Embed_{\text{CodeBERT}}(Func_{\text{before}}),\ Embed_{\text{CodeBERT}}(Func_{\text{after}})) \geqslant \tau_{\text{sem}}\ 。 \quad (1)$$

2)结构一致性(MR2)。重构操作应保持程序的宏观结构相对稳定，尤其是在“提取函数”这类局部重构中。因此，利用对代码语法结构敏感的预训练模型 TreeBERT，将重构前后函数的抽象语法树编码为结构向量。MR2 定义为：2 个结构向量间的余弦相似度高于预设阈值τ_{struc}(0.85)。

$$Similarity_{\text{cosine}}(Embed_{\text{TreeBERT}}(AST_{\text{before}}),\ Embed_{\text{TreeBERT}}(AST_{\text{after}})) \geqslant \tau_{\text{struc}}\ 。 \quad (2)$$

上述 2 个余弦相似度阈值$\tau_{\text{sem}} = 0.98$和$\tau_{\text{struc}} = 0.85$基于以下考量：

1)语义一致性($\tau_{\text{sem}} = 0.98$)。“提取函数”重构的核心原则是严格保持程序外部行为不变。因此，选择了一个非常高的阈值，以最大程度地确保重构前后函数的核心语义没有发生偏移。该阈值设定倾向于高精确率，宁可错拒一些可能正确的重构，也要避免引入由语义改变导致的潜在错误。

2)结构一致性($\tau_{\text{struc}} = 0.85$)。重构本身就是一种代码结构的变换，抽象语法树必然会发生改变。例如，原始代码中的语句被一个新函数调用所取代。因此，该阈值相对较低，允许一定的结构性变化，但仍需确保整体的宏观结构不被破坏。

该阈值是在相关文献 [36–37] 的基础上，结合实验观察而确定的，其在实验中展现了良好的平衡性。然而，这些参数的选择可能影响验证的敏感度，所以，在未来的工作中，进行全面的参数敏感性验证与分析是十分重要的。

算法 4. 重构生成与验证算法。

输入：项目P；

输出：重构后的项目P。

① 获取推荐重构克隆对集合$SuggestedPairs$；
② for 推荐克隆对$pair$ in $SuggestedPairs$ do
③ 识别推荐克隆对$pair$的克隆类型；
④ 构造带 CoT 的少样本提示；
⑤ 调用 LLM 生成重构建议；
⑥ 判断重构前后代码的语义一致性$pass_{MR1}$与结构一致性$pass_{MR2}$；
⑦ if $pass_{MR1}$ and $pass_{MR2}$ do
⑧ 保留重构后的项目P；
⑨ end if
⑩ end for

这 2 个蜕变关系共同构成了一套轻量级但有效的自动化验证框架。虽然基于阈值的启发式判断无法提供形式化证明的绝对保证，但它在实践中能高效过滤掉大量由 LLM 幻觉导致的语义逻辑错误和结构破坏性修改，显著提升了自动化重构的可靠性。

算法 4 描述了完整的生成与验证流程。只有同时通过 MR1 和 MR2 双重验证的重构建议，才被视为最终的、可靠的重构结果输出。行②获取推荐的重构克隆对$SuggestPairs$。行③~⑩针对识别到的重构机会，构造带 CoT 的少样本提示，根据 LLM 提示工程生成重构建议，然后利用上述语义一致性与结构一致性获取蜕变关系验证结果。行⑪~⑬展示若所有蜕变关系均不违反则说明重构代码为理想化的，反之，则需要根据验证结果重新进行重构建议。

综上，重构建议生成的验证流程保证了 LLM 生成的每一条重构建议，都必须通过语义一致性和结构一致性的双重蜕变关系验证。只有通过验证的建议才被采纳为最终的重构方案，未通过的则被拒绝。该机制确保了自动化重构过程的语义保持性和结构稳定性，同时也是实现高可信度重构的关键。

4 实验分析

为全面评估本文提出的非连续代码克隆重构方法的有效性，本文设计了 3 组实验，旨在回答 3 个核心研究问题（research questions, RQs）：

RQ1：本文提出的非连续代码克隆检测方法，相较于现有最先进的工具，其有效性如何？

RQ2：本文提出的基于上下文信息的重构机会识别方法，其识别准确性如何？

RQ3：本文提出的自动化重构方法，在真实的软件项目中对软件质量的提升效果如何？

本节将依次介绍实验设置（包括数据集、基线方法和评估指标），并针对每个 RQ 展示和分析实验结果。

4.1 实验数据

BigCloneBench（BCB）是一个包含超过 800 万个真实 Java 克隆对的大规模基准，其克隆根据功能相似度被标记，非常适合评估面向功能的克隆检测。如表 2 所示，本文所提检测方法选取了其中 T1, T2, ST3, MT3, WT3/T4 等不同类型的总计 269 760 个克隆对进行实验。如表 3 所示，Google Code Jam（GCJ）是一个包含大量语义等价（Type-4）但实现各异的算法竞赛代码库，是评估语义克隆检测能力的黄金标准。鉴于混淆代码数据集能够评估所提方法处理非连续代码克隆重构的能力，所以从 GCJ 中随机抽取 108 个方法，通过程序化地对代码进行语句重排、插入无关语句等操作，人工构建了一个包含已知非连续克隆对的混淆数据集。Community Corpus-A 是一个由社区专家人工标注的、专门用于评估“提取函数”重构机会识别准确性的公开数据集。它包含了 122 个真实的、被认为应该或不应该进行重构的 Java 方法。如表 4 所示，Junit 和 MyWebMarket 是 2 个不同规模和领域的真实开源 Java 项目。Junit 是成熟的单元测试框架，而 MyWebMarket 是一个功能相对简单的 Web 应用。在这些真实项目上进行案例研究，能更好地评估所提方法的实际应用价值。本研究基于上下文窗口上限为 8 192 tokens 的 GPT-4 进行非连续代码克隆提取函数重构，如表 4 所示，单个请求的 token 长度由固定指令、少样本示例和待处理的克隆对代码决定。由于所有样本长度均低于 GPT-4 的

Table 2 Description of BCB Dataset Selected by Non-Contiguous Code Clone Detection

表 2 非连续代码克隆检测选取的 BCB 数据集的描述

数据集类型	克隆对	非克隆对
T1	48 110	48 110
T2	4 000	4 000
ST3	21 395	21 395
MT3	86 341	86 341
WT3	109 914	109 914
总计	269 760	269 760

Table 3 Description of GCJ Dataset Selected by Non-Contiguous Code Clone Detection

表 3 非连续代码克隆检测选取的 GCJ 数据集的描述

方法数	代码行数	类型	函数级克隆对数	语句级克隆对数
108	1 081	T4	432	5 225

token 上限,因此能够保证 LLM 能够接收关于非连续代码克隆对的完整上下文信息。这确保了在进行“提取函数”重构时的信息完整性,排除因上下文截断导致的重构任务出错。

Table 4 Description of Experimental Procedure Junit and MyWebMarket Dataset

表 4 实验程序 Junit 和 MyWebMarket 数据集的描述

实验程序	Junit (单元测试框架)	MyWebMarket (在线购物系统)
类个数	47	18
方法数	509	81
代码行数	5 450	1 515
平均方法代码行数	10.7	18.7
单样本输入平均长度/tokens	3 754	3 923
单样本输入最大长度/tokens	5 078	5 398
GPT-4 上下文上限/tokens	8 192	8 192

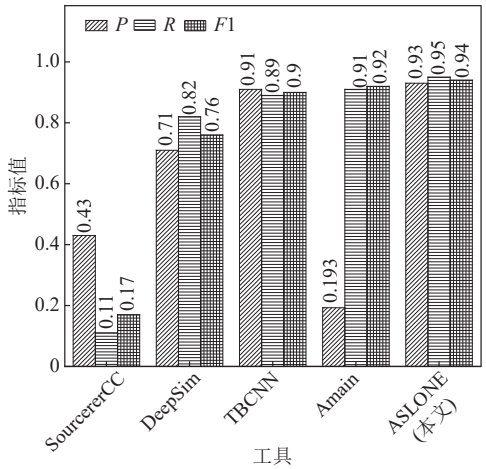
4.2 基准模型及评价指标

对于 RQ1,本文检测方法(ASLONE)与 4 种代表不同技术路线的 SOTA 克隆检测工具进行比较,分别是 SourcererCC[38] (基于词法)、DeepSim[39](基于深度学习)、TBCNN[40](基于树卷积网络)和 Amain[41](基于软件度量)。这些工具覆盖了从传统到前沿的多种主流技术,能够全面评估本文方法的性能。其中 SourcererCC 是一种基于 token 的克隆检测工具,能够用于检测 Type-3 克隆; DeepSim 是一种基于深度学习的代码功能相似性的测量方法,其核心思想在于从语义矩阵中学习高级特征,并通过二分类来测量相似性; TBCNN 是一种通过树卷积网络分析程序 AST 实现程序分类任务的方法,其计算代码对特征向量的余弦相似度,并设定阈值进行相似性检测; Amain 是一种语义相似性(Type-4)克隆检测方法,其利用马尔可夫链模型将 AST 转换为状态转移矩阵,再通过机器学习分类器检测克隆。对于 RQ2,本文重构机会识别方法(C1+C2+C3)与 3 种在 Community Corpus-A 上表现优异的 SOTA 重构机会识别工具进行比较,分别是 JDeodorant, JExtract 和 GEMS。

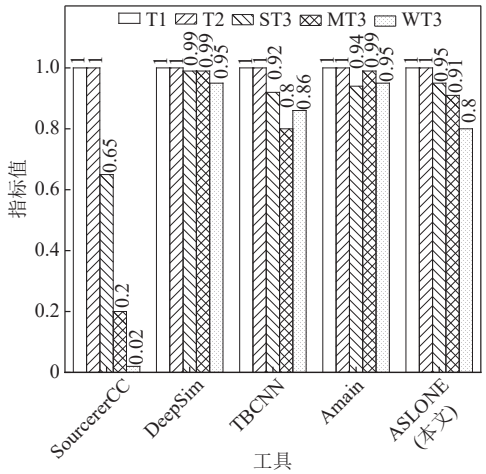
对于 RQ1 与 RQ2,采用标准的精确率(Precision, P)、召回率(Recall, R)和 $F1$ 分数($F1$-Score, $F1$)进行评估。对于 RQ3,采用以下 4 个软件质量度量指标的变化量来评估重构效果,分别是代码变化量(ΔN_e)、代码重复度变化量(ΔN_d)、McCabe 圈复杂度变化量(ΔN_c)和参数数量变化量(ΔN_p)。

4.3 实验结果与分析

针对 RQ1,从图 4 可以看出,在 GCJ 大型基准上,本文提出的检测方法总体 $F1$ 分数达到了 0.94,超越了所有基线方法,同时由表 5 可以看出,在 GCJ 数据集上相对于其他工具而言,检测时间最短、效率最高;而在 BCB 数据集上,在检测 T1 和 T2 类型克隆上 $F1$ 分数与其他工具持平,对于 ST3 类型和 MT3 类型,克隆与 SOTA 工具相差不大,而对于 WT3 类型而言,相对基线方法稍弱,这证明了基于功能性片段和软件度量的方法能有效捕捉代码的核心功能语义。而在专门构造的混淆数据集 GCJ 上,本文方法的召回率显著高于其他工具,这得益于程序切片技术能穿透无关语句的干扰,准确地将非连续的功能相关代码聚合在一起。本文提出的非连续克隆检测方法,在检测精度和综合性能上均达到了 SOTA 水平,尤其在处理由语句穿插、重排导致的非连续克隆时,展现出更强的鲁棒性。



(a) GCJ数据集实验对比



(b) BCB数据集实验对比

Fig. 4 Comparative diagram for experimental results on two datasets

图 4 2 个数据集上的实验结果对比图

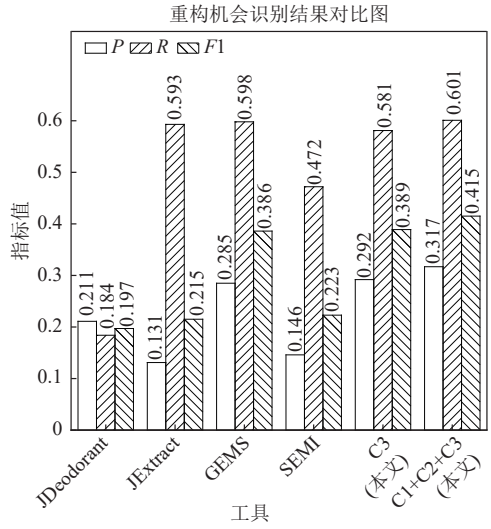
Table 5 Comparative Results of Average Detection Time on GCJ Dataset

表 5 GCJ 数据集上平均检测时间对比结果

工具	平均检测时间/s
SourcererCC	
DeepSim	13 545±1 000
TBCNN	41 168±2 589
Amain	1 017±80
ASLONE（本文）	110±20

注：“±”前后分别表示平均值和标准差。

针对 RQ2，如图 5 所示，在 Community Corpus-A 数据集上评估了所提重构机会识别方法的性能，并与 SOTA 工具进行了比较。为验证“上下文”的有效性，还进行了消融实验，即仅使用代码属性上下文进行识别。实验结果表明，仅使用属性特征时，所提重构机会识别方法（C1，C1+C2+C3）性能已与 SOTA 工具 GEMS 相当。而在完整地引入了继承上下文和位置上下文后，*F*1 分数达到了 0.415。总体而言，较 JDeodorant、JExtract、GEMS，SEMI 工具在 *P*、*R*、*F*1 值都有所提升，尤其，*F*1 值较先进工具 GEMS 提升了 7.5%。这表明综合考虑克隆在软件结构中的位置信息，能够显著提升重构机会识别的准确性，过滤掉大量仅在理论上可提取的重构候选。本文提出的基于上下文信息的重构机会识别方法，通过融合继承、位置和属性 3 个维度的上下文信息，其识别准确性超越了现有的 SOTA 工具。



注：C3 表示仅使用属性上下文，C1+C2+C3 表示融合继承、位置和属性 3 个维度的完整上下文信息。

Fig. 5 Comparative diagram for experimental results on Community Corpus-A dataset

图 5 Community Corpus-A 数据集实验结果对比图

尽管本文方法的 *F*1 值较 GEMS 有所提升，但从绝对值来看并不算高，这反映了对非连续的代码克隆进行重构任务的挑战性。本研究主要通过“提取函数”重构机会识别来进行代码的重构，在此阶段中，所依赖的上下文信息成为重构机会识别准确性的关键因素。本文所提的重构机会识别方法仅从继承、属性和位置上下文角度进行考虑，正由于静态分析技术在处理复杂的语言特性时通常是不完备的，存在导致未能捕捉到所有正确依赖的可能性，从而能影响重构机会的识别。程序静态分析提供的结果与重构机会识别高度关联，因此，丰富的上下文信息能够减少推荐重构克隆对的误报率。其次，Community Corpus-A 数据集来自真实 Java 项目，其包含了大量复杂业务逻辑，与人工合成数据集相比，这一特点使得自动化工具的表现有所下降。在此基准上，尽管其绝对性能不高，但本文方法证明了，对于复杂且富有挑战性的非连续代码克隆重构识别任务而言，融合静态分析与 LLM 能够在重构领域的任务中展现出优越性。

针对 RQ3，在 Junit 和 MyWebMarket 两个真实项目中应用了所提重构框架，并量化了重构前后软件质量度量的变化。表 6 为本文所提检测方法在具体项目上的检测结果。表 7 表明，本文的自动化重构方法显著提升了软件质量。平均每个重构实例能减

Table 6 Non-Contiguous Code Clone Detection Results on Experimental Procedure

表 6 实验程序的非连续代码克隆检测结果

实验程序	克隆类数量	克隆对数	连续克隆代码对	非连续克隆代码对	非连续克隆占比/%
Junit	100	226	211	15	6.63
MyWebMarket	72	291	260	31	7.30

Table 7 Comparison of Non-Contiguous Code Clone Extract Function Refactoring Effectiveness

表 7 非连续代码克隆提取函数重构效果对比

软件质量度量指标	项目	重构前	重构后	变化量	评估/%
代码量 N_e	Junit	68	22	−46	↓67.6
	MyWebMarket	376	109	−267	↓71.0
重复度 N_d	Junit	58	5	−53	↓91.4
	MyWebMarket	352	28	−324	↓92.0
复杂度 N_c	Junit	5	2	−3	↓60.0
	MyWebMarket	15	6	−9	↓60.0
参数数量 N_p	Junit	0	11	+11	↑
	MyWebMarket	0	8	+8	↑

注：“↓”表示重构后的软件质量度量指标相比于重构前的变化下降，“↑”表示重构后的软件度量指标相比于重构前的变化增长。

少约 48 行代码,并将代码重复度降低超过 90%。同时,由于消除了冗余的控制流,圈复杂度也大幅下降。代价是平均每个新提取的函数需要 1.7 个参数,这是一个在提高模块化和代码复用性时完全可以接受的、合理的接口复杂度增加。本文提出的自动化重构方法能够成功应用于真实软件项目,通过“提取函数”重构,显著减少代码量、消除重复、降低复杂度,有效提升软件的整体质量和可维护性。

由表 8 可见,在 Junit 项目的案例中,本文所提重构方法推荐了 15 个重构机会,但最终只实施了 4 个。这种差距并非是单一原因造成的,它反映了多阶段框架中每个环节的过滤作用。针对 11 个被拒绝的重构机会,分别进行了归因分析。在人工审查阶段,有 3 个由算法识别出的克隆对虽然在结构和度量上相似,但从人类开发者的角度来看,其逻辑功能差异较大,合并价值不高。这 3 个在机会识别阶段的误报,暴露了当前重构机会识别算法在深层次语义理解上的局限性。另外有 8 个重构机会在生成阶段与验证阶段被拒绝。其中 5 个由 LLM 生成的重构建议未能通过 MR2 验证,这是因为 LLM 未能正确处理所有相关的变量传递,导致生成的函数调用与原代码块不等价。其余 3 个生成建议未能通过 MR1 验证,这表明 LLM 在生成代码时存在幻觉,引入了与原始逻辑不符的错误。

Table 8 Intermediate Results of Refactoring Opportunity Recognition

表 8 重构机会识别中间结果

识别指标	关系维度	Junit	MyWebMarket
继承关系/%	共同类	12.9	28
	共同层级	81.4	39.5
	共同接口	3.2	7.9
	无关	2.5	24.6
位置关系/%	方法/构造器	89.0	90.5
	类级别	5.3	7.6
	接口/枚举级别	5.7	1.9
重构机会统计个数	推荐重构数	15	19
	实际重构数	4	12

这表明,验证机制成功识别了 8 个由 LLM 引入的错误重构,凸显了在自动化重构流程中识别欺骗性重构的必要性。同时,该问题也反映了重构机会识别的准确性仍然需要进一步提升。

虽然本文并未直接与 LLM 进行消融实验,但是根据现有实验结果能够充分得出静态分析预处理引导 LLM 生成并通过蜕变关系机制约束的框架有效性。在 RQ2 中,针对重构机会识别进行了上下文的消融实验,在仅使用属性上下文的情况下,性能指标明显低于完整框架。本文提出的完整框架通过静态分析为 LLM 提供了合理的证据,该消融结果间接证明了静态分析预处理是完整框架中不可或缺的一步。其次,直接利用 LLM 重构的另一个问题是“幻觉”副作用,而 RQ3 的实验结果证明,通过蜕变关系机制约束 LLM 能够有效避免重构的语义或结构一致性破坏,这证明了仅依赖 LLM 的生成能力是不可靠的,无法确保重构前后代码行为逻辑的一致性。

综上所述,本研究通过重构机会识别上下文消融与语义结构一致性验证实验设计,间接且充分证明了静态分析与 LLM 混合框架的重要性。

4.4 效度威胁分析

本研究的效度威胁以及缓解措施如下。

1)外部效度。本实验仍主要局限于 Java 语言,该方法对其他语言(如 C++, Python)的泛化能力有待进一步验证。此外,本研究的实现和评估是基于 GPT-4 模型,该模型具有最大上下文窗口 8 192 tokens,该限制是进行克隆检测与重构的前提约束条件。通过静态分析预处理来引导 LLM 进行代码重构的思路,从某种程度上仍然依赖于模型的代码语义理解能力。选取的模型在代码生成和理解能力方面越强,那么代码重构的结果准确性也就越高、代码逻辑错误率越低。因此,若将本框架应用于能力稍弱或特定领域的开源大模型(如 CodeLlama、DeepSeek-Coder 等),虽然整体的静态分析预处理、LLM 引导生成、基于蜕变关系验证的流程依然具有重要价值,但可能会面临以下挑战:生成建议的准确率下降,可能出现更多逻辑错误或不完整的重构;需要设计更精细、更具引导性的提示工程,以弥补模型推理能力的不足。

2)内部效度。本文所提原型工具可能存在实现缺陷。LLM 的随机性也可能影响结果,所以,通过设置固定的 temperature 参数和多次运行来确保结果的稳定性。此外,在实验过程中能够洞察到,静态分析的预处理结果会直接影响后续的代码重构任务。因此,即便本文所设计的核心变量选取规则能够覆盖绝大多数非连续代码克隆片段相关的变量,但仍然存在核心变量选取的遗漏。针对语言本身更复杂的特性,设计更准确的静态分析规则是未来工作中重要的一部分。在实验评估阶段,本文采用了学术界和工业界广泛认可的软件度量和评估指标。但这些指标仍可能无法完全捕捉软件质量的所有方面。

4.5 局限性与失败案例分析

尽管所提方法在多个基准和真实项目上取得了显著成果，但其仍然在进行重构时存在局限。

失败案例 1：复杂的副作用与隐式状态依赖

当克隆片段涉及共享对象或全局状态的隐式副作用时，虽然程序切片技术能识别依赖，但后续 LLM 即使在 CoT 的引导下，仍可能无法准确复现语义契约，导致生成的代码引入隐蔽错误。

失败案例 2：大规模跨模块的深层架构克隆

本方法主要关注函数级的语句克隆，难以处理跨模块、体现设计模式的高层次克隆（如不同类中遵循相同的“资源获取、异常处理、资源释放”模板代码）。因为程序切片范围的局限性导致无法捕捉跨模块的架构克隆。

上述失败案例揭示了本方法后续改进的方向。需要融合更深层次的程序语义分析技术处理副作用，并借助代码知识图谱等技术将重构能力由代码级提升至设计级。

本研究实现依赖 GPT-4 模型，其有利于重构效果，但影响复现。然而本文框架的核心在于用静态分析结构化任务，为 LLM 提供明确的重构目标，利用验证机制约束生成结果。通过静态分析引导 LLM 并加以约束的框架具有良好泛化潜力。在未来工作中，可以进一步探索该框架在不同 LLM 上的表现。

5 总　　结

本文提出了一种融合静态分析与 LLM 的非连续代码重构方法，为自动化重构领域提供了解决方案。此外，自动化验证机制为其他 AI 驱动的复杂软件工程任务，如自动程序修复、面向领域需求的测试用例生成等提供了思路和方法论借鉴。

尽管本文方法通过“提取函数”方法进行代码重构，并取得了有效的成果，但对于涉及复杂架构或设计模式的宏观重构，本文所提出的方法尚待进一步验证。基于代码表示相似度的蜕变关系验证手段，是一种高效的、概率性的验证手段，但并非形式化的正确性保证。对于包含复杂副作用或并发操作的代码，仅通过向量相似度可能无法完全捕捉到所有潜在的语义偏差。

未来的工作可以探索将本文的框架扩展到处理更高层次的“设计异味”。这需要研究更强大的程序表示方法，如代码知识图谱，以捕捉跨文件、跨模块的复杂依赖。在此基础上，可以引导 LLM 执行如“提取类”“移动方法”等更复杂的、涉及架构调整的重构操作。

作者贡献声明：嵇友晴完成实验并撰写论文；张迎周提出了论文思想和算法框架并进行了方案指导；苏玉鹏完成论文的撰写；王刚、张文智、谢金言修改论文。

参　考　文　献

[1] Martin Fowler. Refactoring: Improving the Design of Existing Code[M]. USA: Addison-Wesley Professional, 2002

[2] Kaur G, Singh B. Improving the quality of software by refactoring[C]//Proc of 2017 Int Conf on Intelligent Computing and Control Systems (ICICCS). Piscataway, NJ: IEEE, 2017: 185−191

[3] Khanzadeh S, Chan S A N, Valenzano R, et al. Opti code pro: A heuristic search-based approach to code refactoring[J]. arXiv preprint arXiv: 2305.07594, 2023

[4] Romano S, Vendome C, Scanniello G, et al. A multi-study investigation into dead code[J]. IEEE Transactions on Software Engineering, 2018, 46(1): 71−99

[5] Higo Y, Kamiya T, Kusumoto S, et al. Method and implementation for investigating code clones in a software System[J]. Information and Software Technology, 2007, 49(9-10): 985−998

[6] Saini N, Singh S. Code clones: Detection and management[J]. Procedia Computer Science, 2018, 132: 718−727

[7] Yarahmadi H, Hasheminejad S M H. Design pattern detection approaches: A systematic review of the literature[J]. Artificial Intelligence Review, 2020, 53(8): 5789−5846

[8] Baumgartner N, Iyenghar P, Schoemaker T, et al. Ai-driven refactoring: A pipeline for identifying and correcting data clumps in git repositories[J]. Electronics, 2024, 13(9): 1644−1667

[9] Hong J, Ryu S. Type-migrating C-to-Rust translation using a large language model[J]. Empirical Software Engineering, 2025, 30(1): 3

[10] Tsantalis N, Mazinanian D, Krishnan G P. Assessing the refactorability of software clones[J]. IEEE Transactions on Software Engineering, 2015, 41(11): 1055−1090

[11] Barbosa F S, Aguiar A. Removing code duplication with roles[C]// Proc of 2013 IEEE 12th Int Conf on Intelligent Software Methodologies, Tools and Techniques (SoMeT). Piscataway, NJ: IEEE, 2013: 37−42

[12] Zibran M F, Roy C K. Conflict-aware optimal scheduling of prioritised code clone refactoring[J]. IET Software, 2013, 7(3): 167−186

[13] Mazinanian D, Tsantalis N, Stein R, et al. JDeodorant: Clone refactoring[C]//Proc of the 38th Int Conf on Software Engineering Companion (ICSE' 16). Texas, Austion: ACM, 2016: 613−616

[14] Xiao Quanbing, Chen Yuan, Wu Yijian, et al. Function template mining and retrieval based on code clone difference analysis[J]. Journal of Software, 2025, 36(6): 2774−2793 (in Chinese)
(肖泉彬, 陈源, 吴毅坚, 等. 基于代码克隆差异分析的函数模板挖

掘和检索方法[J]. 软件学报, 2025, 36(6): 2774−2793)
[15] Xiao Yang, Chen Bihuan, Yu Chendong, et al. MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures[C]//Proc of the 29th USENIX Symp (Sec'20). Berkeley, CA: USENIX Association, 2020: 1165−1182
[16] Li Lin, Ye Jialin, Wang Chao, et al. PATEN: Identifying unpatched third-party APIs via fine-grained patch-enhanced AST-level signature[J]. IEEE Transactions on Software Engineering, 2025, 51(4): 990−1006
[17] Song Xiaonan, Yu Aimin, Yu Haibo, et al. Program slice based vulnerable code clone detection[C]//Proc of 2020 IEEE 19th Int Conf on Trust, Security and Privacy in Computing and Communications (TrustCom). Piscataway, NJ: IEEE, 2020: 293−300
[18] Peng Huiyun, Gupte A, Hasler R, et al. SysLLMatic: Large language models are software system optimizers[J]. arXiv preprint arXiv: 2506.01249, 2025
[19] Prakash Raj Ojha. Democratizing code: How GPT and large language models are reshaping the landscape of software creation[J]. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 2024, 10(5): 503−512
[20] Yeo S, Ma Y S, Kim S C, et al. Framework for evaluating code generation ability of large language models[J]. Electronics and Telecommunications Research Institute Journal, 2024, 46(1): 106−117
[21] Wermelinger M. Using GitHub Copilot to solve simple programming problems[C]//Proc of the 54th ACM Technical Symp on Computer Science Education V. 1. Toronto, Canada: ACM, 2023: 172−178
[22] Huang Kai, Xu Zhengzi, Yang Su, et al. A survey on automated program repair techniques[J]. arXiv preprint arXiv: 2303.18184, 2023
[23] Xu Zimao, Jiang Yanjie, Zhang Yuxia, et al. Large language model based decomposition of long methods[J]. Journal of Software, 2025, 36(6): 2501−2514 (in Chinese)
(徐子懋, 姜艳杰, 张宇霞, 等. 基于大语言模型的长方法分解[J]. 软件学报, 2025, 36(6): 2501−2514)
[24] Siddeeq S, Waseem M, Rasheed Z, et al. LLM-based multi-agent system for intelligent refactoring of Haskell code[J]. arXiv preprint arXiv: 2506.19481, 2025
[25] Letouzey P. Extraction in Coq: An overview[C]//Proc of Conf on Computability in Europe. Berlin: Springer, 2008: 359−369
[26] Wenzel M, Paulson L C, Nipkow T. The Isabelle framework[C]/Proc of Int Conf on Theorem Proving in Higher Order Logics. Berlin: Springer, 2008: 33−38
[27] Cadar C, Dunbar D, Engler D R. KlEE: Unassisted and automatic generation of high-coverage tests for complex systems programs[C]// Proc of 8th USENIX Symp on Operating Systems Design and Implementation(OSDI). Berkeley, CA: USENIX Association, 2008, 8: 209−224
[28] Sonnekalb T, Gruner B, Brust C A, et al. Generalizability of code clone detection on codeBERT[C]//Proc the 37th IEEE/ACM Int Conf on Automated Aoftware Engineering(ASE'22). New York: ACM, 2022: 1−3
[29] Almasri N, Korel B, Tahat L. Verification approach for refactoring transformation rules of state-based models[J]. IEEE Transactions on Software Engineering, 2021, 48(10): 3833−3861
[30] Chen T Y, Cheung S C, Yiu S M. Metamorphic testing: A new approach for generating next test cases[J]. arXiv preprint, arXiv: 2002.12543, 2020
[31] Xue Pengyu, Wu Linhao, Yang Zhen, et al. Exploring and lifting the robustness of LLM-powered automated program repair with metamorphic testing[J]. arXiv preprint arXiv: 2410.07516, 2024
[32] Sun Chang'ai, Xing Jiayu, Liu Baoli, et al. Path-directed source test case generation and prioritization in metamorphic testing[J]. Chinese Journal of Computers, 2025, 48(3): 675−693 (in Chinese)
(孙昌爱, 邢嘉煜, 刘宝莉, 等. 基于路径分析的蜕变测试组生成与优先级排序技术[J]. 计算机学报, 2025, 48(3): 675−693)
[33] Izbicki M. Algebraic classifiers: A generic approach to fast cross-validation, online training, and parallel training[C]//Proc of the 30th Int Conf on Machine Learning. Atlanta, USA: PMLR, 2013: 648−656
[34] Ioannis S, Georgios G, Diomidis S, et al. The SQO-OSS quality model: Measurement based open source software evaluation[C]//Proc of the Int Conf on Open Source Systems. Milano, Italy: Springer, 2008: 237−248
[35] Mikeizbicki. HLearn[EB/OL]. (2015-07-22)[2025-04-24]. https://github com/mikeizbicki/HLearn
[36] Lu Shuai, Guo Daya, Ren Shuo, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation[J]. arXiv preprint arXiv: 2102.04664, 2021
[37] Hindle A, Barr E T, Gabel M, et al. On the naturalness of software[J]. Communications of the ACM, 2016, 59(5): 122−131
[38] Sajnani H, Saini V, Svajlenko J, et al. SourcererCC: Scaling code clone detection to big-code[C]//Proc of the 38th Int Conf on software engineering(ICSE'16). Texas, Austin: ACM, 2016: 1157−1168
[39] Zhao Gang, Huang Jeff. DeepSim: Deep learning code functional similarity[C]//Proc of the 2018 26th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering(ESEC/FSE'18). New York: ACM 2018: 141−151
[40] Mou L, Li G, Jin Z, et al. TBCNN: A tree-based convolutional neural network for programming language processing[J]. arXiv preprint arXiv: 1409.5718, 2014
[41] Wu Yueming, Feng Siyue, Zou Deqing, et al. Detecting semantic code clones by building AST-based Markov chains model[C]//Proc of the 37th IEEE/ACM Int Conf on Automated Software Engineering (ASE'22). New York: ACM 2022: 1−13

Ji Youqing, born in 2000. Master. Her main research interests include program analysis and software engineering.
嵇友晴, 2000 年生。硕士。主要研究方向为程序分析、软件工程。

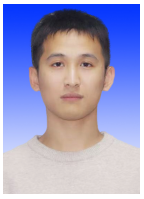
Zhang Yingzhou, born in 1978. PhD, professor, master supervisor. His main research interests include program analysis and program slicing.
张迎周, 1978 年生。博士, 教授, 硕士生导师。主要研究方向为程序分析、程序切片。

Su Yupeng, born in 2001. Master candidate. His main research interests include program analysis and defects detection.

苏玉鹏, 2001 年生。硕士研究生。主要研究方向为程序分析、缺陷检测。

Wang Gang, born in 2000. Master candidate. His main research interests include program analysis and malware detection.

王　刚, 2000 年生。硕士研究生。主要研究方向为程序分析、恶意软件检测。



Zhang Wenzhi, born in 2000. Master candidate. His main research interests include program analysis and malware detection.

张文智, 2000 年生。硕士研究生。主要研究方向为程序分析、恶意软件检测。

Xie Jinyan, born in 1999. Master. His main research interests include program analysis and program slicing.

谢金言, 1999 年生。硕士。主要研究方向为程序分析、程序切片。