中国科学 : 信息科学 2025 年 第 55 卷 第 11 期: 2886–2901
SCIENTIA SINICA Informationis
《中国科学》杂志社
SCIENCE CHINA PRESS
人工智能应用中的数据安全专刊 · 论文

从编译到反编译: 基于源码级转换的高效水印去除方法

黄昊[1,2], 周瑞桦[1,2], 罗家棠[3], 马晓欢[1,2], 李运鹏[1,2], 刘玉岭[1,2*]

1. 中国科学院信息工程研究所, 北京 100085
2. 中国科学院大学网络空间安全学院, 北京 100049
3. 中国科学院大学前沿交叉科学学院, 北京 100049
* 通信作者. E-mail: liuyuling@iie.ac.cn

收稿日期: 2025–05–30; 修回日期: 2025–09–09; 接受日期: 2025–09–25; 网络出版日期: 2025–11–06

国家重点研发计划 (批准号: 2023YFC3306305) 资助项目

摘要 大语言模型在高效生成代码的同时, 也引发了版权归属与代码滥用的潜在风险. 为此, 大语言模型水印技术应运而生以追踪代码来源. 然而, 当水印技术应用于代码这一特殊载体时, 其有效性与鲁棒性面临严峻考验. 为了检测代码水印鲁棒性, 现有基于抽象语法树 (AST) 的攻击策略在效率和稳定性上仍存不足. 为此, 本文提出一种名为 DeMark 的新型编译 – 反编译水印去除方法. 该方法首先将含水印代码编译为低级二进制表示, 利用编译器优化消除水印痕迹; 随后, 通过创新的自动化函数调用图构建算法, 将优化后的二进制代码反编译回保留核心功能与可读性的高级语言代码, 从而高效去除水印. 实验结果表明, DeMark 在攻击现有水印方法方面表现出显著有效性和更强的适应性, 尤其在处理涉及复杂语义保留变换的水印时优势明显. 此外, 本研究首次系统评估了 C++ 等编译型语言水印在代码重构和跨语言转换场景下的鲁棒性, 揭示了当前文本水印算法应用于代码生成的固有缺陷, 为未来设计更稳健的代码溯源机制提供了重要参考.

关键词 水印安全去除, 大语言模型安全, 可信代码生成, 编译安全优化, 对抗鲁棒性

1 引言

近年来, 以 DeepSeek [1] 和 Qwen [2] 为代表的大语言模型已能高效生成功能性代码 [3,4], 但也引发了代码滥用和版权归属的担忧 [5]. 为应对这一挑战, 研究者们开始将水印技术应用于大语言模型 [6~8], 旨在通过植入可检测标记来追踪代码来源. 然而, 将水印应用于代码这一特殊载体时, 其声称的有效性和鲁棒性面临着严峻的现实考验. 与自然语言文本相比, 代码具有显著的特点: 其 "低熵" 特性 [7]—— 为了实现特定功能并遵循严格的语法规则, 有效的词汇和结构组合选择空间相对有限 —— 不仅直接限制了水印的嵌入容量和隐蔽性, 更容易在嵌入过程中影响代码质量. 尽管现有水印研究努力提升其

引用格式: 黄昊, 周瑞桦, 罗家棠, 等. 从编译到反编译: 基于源码级转换的高效水印去除方法. 中国科学: 信息科学, 2025, 55: 2886–2901, doi: 10.1360/SSI-2025-0255
Huang H, Zhou R H, Luo J T, et al. From compile to decompile: efficient watermark removal via source-to-source transformations. Sci Sin Inform, 2025, 55: 2886–2901, doi: 10.1360/SSI-2025-0255

© 2025《中国科学》杂志社

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

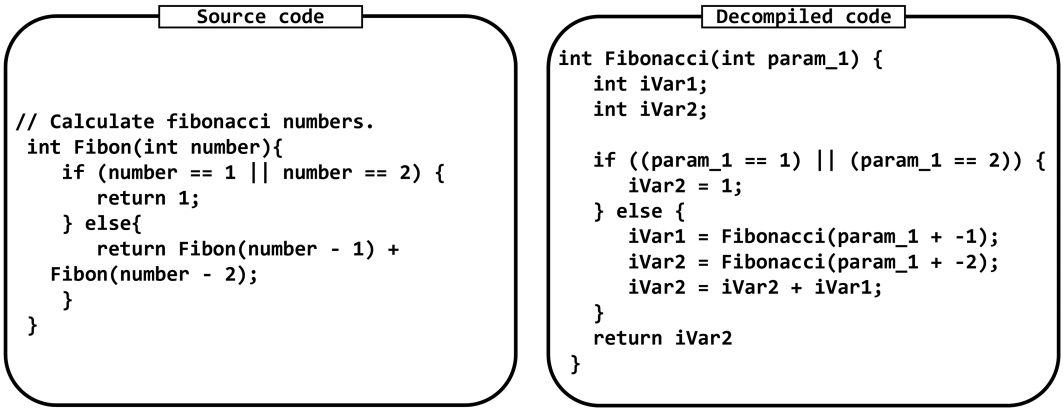

图 1　源代码与反编译代码对比示意图, 反编译过程会显著破坏源代码中既有的水印格式结构.

Figure 1　An illustration comparing source code and decompiled code, clearly showing that the latter is likely to disrupt the watermark format present in the source code.

在特定场景下的鲁棒性 [9], 但代码的这种结构化、易修改以及表达选择有限的本质, 使得我们有充分理由对其在真实对抗环境下的安全性提出疑问.

为了验证代码水印的鲁棒性, 研究者们探索了多种水印攻击策略. 例如, Suresh 等人 [10] 提出了一种基于抽象语法树 (AST) 的重构攻击, 通过随机应用语义保持转换 (如变量重命名、死代码插入等) 来破坏 Python 代码中的水印, 证明了现有水印技术在此类攻击下的脆弱性. 尽管这类基于 AST 的自动化攻击相较于纯手动修改有所进步, 但在实际应用中仍面临着挑战: 首先, 即使是自动化的 AST 遍历和转换, 在处理复杂代码或进行多次迭代时, 仍会消耗大量的时间和计算资源; 其次, 随机性的代码修改, 特别是涉及插入随机代码片段或大幅度重构子树时, 可能会显著降低代码的可读性、可维护性和功能一致性, 不利于后续的代码审计或协作开发; 最重要的是, 由于转换选择和应用位置的随机性, 其攻击效果存在情境依赖性, 使得在不同场景下的表现不够稳定和可靠.

针对上述传统攻击策略在效率、代码质量影响以及可靠性方面的不足, 本研究考虑从编译的角度出发进行代码水印的消除. 我们的动机在于, 如图 1 所示, 反编译代码相较于原始的源代码会发生显著的结构性转换, 具体表现为存在着变量重命名、结构重组、注释删除及格式变更等现象. 因此, 反编译过程可能会破坏代码嵌入式水印结构的完整性, 从而对其鲁棒性带来挑战. 基于此发现, 本研究提出了 DeMark —— 一种基于编译 – 反编译策略的自动化水印去除方法, 该方法首先将带有水印的代码编译为低级的二进制指令, 通过编译器优化去除水印相关信息, 随后将代码反编译回高级语言, 得到最终消除水印的代码. 具体而言, 在编译过程中, 编译器 (如 GCC, Clang 或 MSVC) 会执行大量优化, 源代码中包含的丰富高级语义信息 (如原始变量名、注释、编码风格、特定的语法糖和抽象结构), 在编译成机器码时会被大量丢弃或抽象化. 这些优化在保持代码功能的同时会显著改变其结构, 从而有效混淆或去除嵌入式水印. 在反编译阶段, 我们提出了自动化函数调用图构建算法, 通过遍历所有函数依赖关系, 生成完整的函数集合, 使得反编译工具在此基础上重构高级伪代码, 进一步确保其输出不再保留水印的原始模式. 最后, 我们利用大语言模型优化反编译输出结果, 生成无水印、功能完整且可读性高的代码. 通过这一过程, 我们的方法成功去除了嵌入的水印, 同时保留了代码的核心功能 (见图 2). 与传统方法相比, 我们的编译 – 反编译策略展现出更强的适应性, 尤其是在处理涉及语义保留变换的复杂水印方面. 本研究的主要贡献如下.

(1) 本文提出了一种基于编译 – 反编译策略的代码水印去除方法, 该方法能够在保持代码功能性和可读性的前提下高效移除水印.

(2) 本文系统性地评估了编译型语言水印在代码重构和跨语言转换场景下的鲁棒性, 证明了水印的保留强度与源语言和目标语言的语法相似度密切相关, 为后续研究奠定了重要基础.

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

Question

```
bool check_list_value(const std::vector<int>& l, int t)
    """Return true if all numbers in the list
    l are below threshold t.
    """
```

(a) Solution

```
    for (int elem : l) {
        if (elem >= t) {
            return False;
        }
        return True;
    }
```

(b) KGW, Strong watermark

```
    for (int k : l) {
        if (t <= k) {
            break;
        }
        return True;
    }
```

(c) KGW, Weak watermark

```
    for (int elem : l) {
        if (elem >= t) {
            return False
        }
        return True
    }
```

(d) SWEET Selective watermarking

```
    for (int k : l) {
        if (t <= k) {
            return false;
        }
        return true;
    }
```

(e) DeMark(ours)

```
    return std::all_of(list.begin(), list.end(), [threshold](int value) {
        return value > threshold;
    });
```

图 2　(网络版彩图) **KGW** [6], **SWEET** [9] 与本研究方法效果差异. 图为简化示例, 绿色标记 (b, c, d) 表示水印检测所需的关键令牌, 本研究的攻击方法能完全消除这些着色标记. (a) 正确实现; (b) **KGW** (强水印); (c) **KGW** (弱水印); (d) **SWEET** (选择性水印); (e) **DeMark** (本文方法).

Figure 2　(Color online) Simplified comparison of KGW [6], SWEET [9], and our watermark-removal attack. The green annotations (b, c, d) denote the key tokens required for watermark detection; our attack eliminates these highlighted tokens entirely in this illustrative example. (a) Solution; (b) KGW (strong watermark); (c) KGW (weak watermark); (d) SWEET (selective watermarking); (e) DeMark (ours).

(3) 本文通过开展全面的实验, 验证了 DeMark 针对现有水印方法攻击的有效性, 其中, Unigram 水印方法在 DeMark 攻击下的 AUROC 值达到了 54.96% 的下降率, 为大模型生成代码的水印抗攻击研究提供了重要见解.

本文的第 2 节介绍了相关工作, 第 3 节详细阐述了本文水印攻击方法, 第 4 节为实验设置、实验结果及对结果的分析与讨论, 第 5 节是通用性讨论, 第 6 节是对本文的总结和对未来工作的展望.

2　相关工作

2.1　大模型水印技术

大语言模型 (LLM) 水印技术主要通过在文本生成过程中巧妙嵌入特定信号, 旨在保证输出文本质量的同时, 实现对内容来源的鲁棒追溯和有效检测 [11~14]. 在众多大模型水印方法中, 早期的探索之一是 Aaronson 等人 [15] 提出的基于 EXP 采样的算法 (EXP). 该方法通过先前词元 (token) 生成一个伪随机值序列, 并在选择下一个词元时, 倾向于那些能够最大化自身概率与这些伪随机值加权函数的词元. 检测阶段则依赖于一个对数评分函数来评估文本与该伪随机序列的符合程度, 并通过设定阈值进行识别. EXP 方法在初步探索 LLM 水印时, 展示了在维持模型实用性的同时嵌入可检测信号的潜

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

力. 为了追求更强的控制性和可能的鲁棒性, 后续研究转向了对模型输出分布进行更直接干预的策略. 例如, Kirchenbauer 等人 [6] 提出的 KGW 方法, 采用了一种基于 logit 的绿名单机制. 具体而言, 它通过整合先前词元的哈希值来动态划分词汇表, 引导模型从 “绿色” 词元子集中选择下一个词元, 从而嵌入水印. 尽管如 KGW 等方法在水印嵌入方面取得进展, 但提升水印在面对复杂文本攻击 (如释义、重组) 时的鲁棒性仍然是一个核心挑战. 针对这一问题, Zhao 等人 [8] 提出的 Unigram, 开始探索简化水印嵌入机制的路径. Unigram 通过采用一种固定的绿红词表划分策略, 而非动态哈希, 旨在牺牲一定的不可见性或密钥空间以换取在特定攻击下的水印稳定性. 这种简化思路代表了在水印设计中对鲁棒性、不可见性和复杂度之间进行权衡的一种尝试.

现有水印方法在特定应用场景, 如代码生成任务中, 表现不佳. 这一挑战的核心在于代码的固有低熵特性: 与自然语言相比, 代码的结构化和规范性使其词汇选择的空间较小, 导致许多片段是高度确定性的. 这使得在不显著改变代码功能或可读性的前提下嵌入不可感知的、高容量的水印变得困难. 现有的基于绿/红列表或 logit 偏移的水印方法, 在面对这种低熵特性时, 往往难以找到足够的 “自由度” 来嵌入鲁棒的水印, 或者会导致代码质量的下降. 为解决这些局限性, 一种新方法 (SWEET) [9] 通过选择性排除低熵片段进行水印嵌入, 例如避免在语法强制或高度模式化的代码结构中嵌入水印, 而是专注于那些存在多种合法表达方式的片段, 从而在保持代码质量的同时显著提升了检测性能, 其表现远超现有方法. 然而, SWEET 在计算整个代码完成所需的次元时, 可能面临计算效率低的问题, 尤其是在处理大型代码库或实时生成场景中. 尽管取得了这些进展, 大语言模型水印领域仍面临诸多挑战和开放问题. 首先是水印鲁棒性与不可感知性的核心权衡: 如何在保证水印能抵抗各种攻击 (如释义、重组、对抗性攻击) 的同时, 最大限度地减少对生成文本质量和语义的负面影响, 是持续研究的重点. 其次是水印容量和多用户溯源能力: 如何嵌入足够的信息来识别不同的 LLM 模型、服务提供商乃至特定的用户, 以应对未来复杂的版权和责任归属问题. 此外, 随着 LLM 向多模态生成发展, 如何在文本、图像、音频、视频等多模态内容中实现统一且鲁棒的水印方案, 也成为一个新兴且紧迫的研究方向. 现有研究在统一评估平台和度量标准方面仍显不足, 使得不同水印方法之间的公平比较变得困难. 最后, 如何将水印技术与模型的自身安全性 (如模型窃取、滥用) 相结合, 例如通过模型水印在 LLM 本身中嵌入可识别信息, 也是未来值得探索的方向 [7, 16].

2.2　水印鲁棒性

现有的水印攻击方法主要可分为两大类. 一类是自然语言处理领域的转述攻击 (paraphrasing attack). 例如, He 等人 [17] 的研究表明, 通过将文本翻译至其他语言再翻译回来 (即 “往返翻译”), 可以有效破坏文本水印的统计特征, 这为代码水印攻击提供了思路, 并直接启发了本研究中将 C++ 代码转换为 Python 再转回的攻击基线设计. 另一类则是针对代码载体本身的结构性攻击. 在这方面, Suresh 等人 [10] 提出了一种基于抽象语法树 (AST) 进行代码重构的新方法, 用于评估水印算法对抗语义保持转换的能力. 该研究通过测试 Python 代码中常见的多种修改方式 (包括插入死代码、变量重命名、try-catch 代码块封装等), 系统评估了这些操作对 KGW [6] 和 Unigram [8] 等水印检测算法的影响. 实验结果表明, 即使是简单的修改 (如插入 print 语句或变量重命名) 也会显著降低水印检测的真阳性率 (TPR), 而更复杂的修改 (如嵌套 try-catch 块或死代码插入) 影响更为显著. 随着修改次数的增加, TPR 进一步下降, 其中 WrapTryCatch 修改经过 5 次迭代后使 KGW 水印的 TPR 从 0.79 降至 0.22. 研究还发现, 代码的执行性能随着修改次数的增加而恶化, 增加的逻辑导致执行时间延长. 尽管该研究为代码水印技术的鲁棒性提供了重要见解, 但仍存在若干局限: 首先, 该方法需要多轮计算, 消耗大量计算资源; 其次, 研究仅关注水印去除而未考虑代码可读性. 这些局限凸显了需要进一步探索既能高效去除水印又能保持代码可读性的新技术.

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

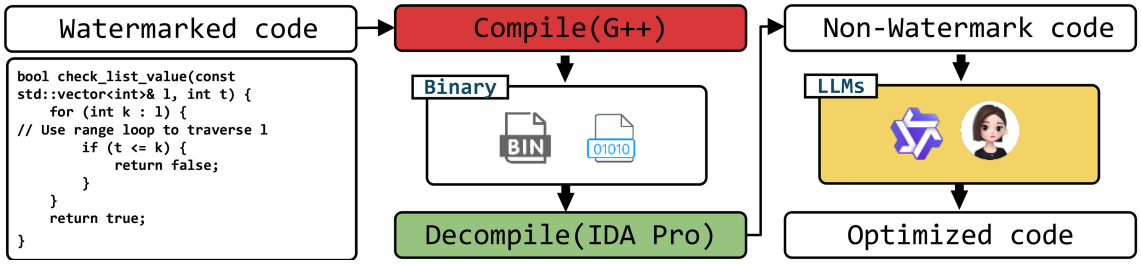

图 3　(网络版彩图) 方法工作流程.

Figure 3　(Color online) Overview of the proposed method.

3　方法

本文的 DeMark 方法旨在通过一个创新的编译 – 反编译 – 优化方法, 高效地从生成代码中去除水印. 本节将详细阐述 DeMark 方法的设计理念、关键组件及其工作流程.

3.1　DeMark: 编译 – 反编译 – 优化的水印去除方法概览

现有基于抽象语法树重构 [10] 的攻击方法虽然能一定程度上削弱水印, 但在效率、稳定性以及对代码质量的影响方面仍存在明显不足. 相比之下, 编译 – 反编译过程天然会导致源代码结构的显著变化, 例如变量重命名、格式重组和注释删除, 这些转换往往会破坏嵌入式水印的完整性. 因此, 我们提出利用编译与反编译的多层次转换作为水印去除的切入点, 并在此基础上进一步结合大语言模型进行优化. DeMark 方法的核心创新在于其独特的编译 – 反编译 – 优化流程, 结合了源码到源码转换技术与大语言模型的智能优化能力. 该方法通过在不同抽象层次间转换代码, 利用此过程中固有的信息丢失和结构重构特性, 来有效去除内嵌于 LLM 生成代码中的水印.

整个 DeMark 流程如图 3 所示: 首先, 将含水印的源代码编译为二进制文件; 再通过反编译将其还原为伪代码; 最后利用大语言模型优化得到功能正常且无水印的源代码. 这个过程可以形式化表示为一系列转换:

$$
\begin{aligned}
B &= \text{Compile}(C_{wm}),\\
C_{de} &= \text{Decompile}(B),\\
C_{opt} &= \text{LLM_Optimize}(C_{de}),
\end{aligned}
$$

其中, C_{wm} 表示带水印的原始源代码, B 是编译后的二进制代码, C_{de} 是反编译 (或初步去水印) 后的伪代码, C_{opt} 表示经过 LLM 优化后的最终代码.

我们的目标是清除代码中的水印, 同时保留原始功能并增强可读性. 这可以表示为

$$
\begin{aligned}
&\min D(C_{opt})\\
&\text{subject to } S(C_{opt}) = S(C_{src}) \quad (\text{功能等价性}),\\
&\text{and Readability}(C_{opt}) > \text{Readability}(C_{de}) \quad (\text{可读性增强}),
\end{aligned}
$$

其中, $D(C)$ 代表代码 C 中水印的可检测性 (或强度) 的函数, 其值越小表示水印越难以被检测; $S(C)$ 表示代码 C 的语义功能, 确保最终代码与原始无水印代码 C_{src} 功能一致; Readability(C) 衡量代码的可读性.

接下来, 我们将在 3.2 小节从理论角度进一步论证这一目标的可实现性.

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

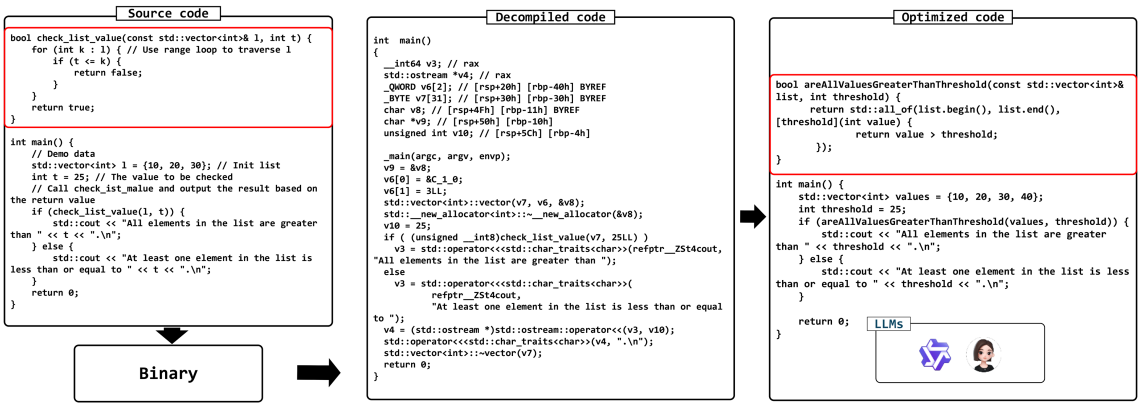

图 4　(网络版彩图) DeMark 方法流程示例.

Figure 4　(Color online) Example workflow of DeMark.

3.2　DeMark 方法的理论保证: 功能保持与可读性增强

基于前述流程, 本节将进一步说明 DeMark 方法在功能保持和可读性增强方面的理论保证. 具体而言, 方法依托于编译器的语义等价性、反编译工具的结构可恢复性, 以及大语言模型的语义优化能力, 来确保在去除水印的同时, 生成代码依然功能正确且可读性更高.

- **编译器保证功能等价性.** 现代编译器经过严格测试和验证, 其主要目标是在不改变程序语义 (即功能) 的前提下, 将高级代码转换为高效的机器码. 因此, 编译后的二进制 B 与原始源代码 C_{src} 在功能上是严格等价的: $S(B) \equiv S(C_{src})$.
- **反编译工具旨在恢复功能.** 反编译工具 (例如 IDA Pro) 旨在尽可能准确地从二进制代码中重建程序的逻辑和功能. 虽然重建的伪代码 C_{de} 可能缺乏 C_{src} 的可读性或精确的结构保真度, 但其核心功能通常得以保留: $S(C_{de}) \equiv S(B)$.
- **LLM 优化确保可执行性与可读性.** 在编译 – 反编译过程之后, DeMark 引入 LLM 对伪代码 C_{de} 进行进一步优化. 这一阶段不仅增强了代码的可读性、简洁性和标准化, 更重要的是, LLM 被指示在优化过程中保持代码的功能完整性, 并通过生成语法正确且可执行的代码来修复可能由反编译引入的潜在结构问题. 因此, $S(C_{opt}) \equiv S(C_{de})$.

结合这些步骤, 功能等价性在整个 DeMark 流程中得以保持: $S(C_{opt}) \equiv S(C_{de}) \equiv S(B) \equiv S(C_{src})$. 通过这些机制, DeMark 能够有效去除水印, 同时确保生成代码的功能完整性和可用性.

整个方法通过这种三阶段的协同工作, 在显著减少人工干预的同时, 实现了对水印的有效去除、代码功能的保持以及最终可读性的提升, 为 LLM 生成的代码提供了一种可靠的源代码恢复和水印去除方案. 各阶段处理效果的直观对比可参见图 4 红框标注区域, 其清晰展示了渐进式优化过程.

3.3　三阶段方法设计与实现

DeMark 方法的具体实现包含以下 3 个关键阶段, 每个阶段都利用特定的机制来达成水印去除和代码质量提升的目标.

3.3.1　第 1 阶段: 编译与水印初步混淆

此阶段使用标准编译器将包含水印的源代码 C_{wm} 编译为二进制代码 B. 编译过程通过以下机制实现对水印的初步混淆与去除.

- **抽象层次转换与高级语义信息丢失.** 编译器将高级语言代码转换为低级二进制指令. 在此过程中, 源代码中包含的丰富高级语义信息, 例如原始变量名、注释、编码风格、特定的语法糖和抽象结构, 在编译成机器码时会大量被丢弃或抽象化. 例如, 有意义的变量名会被替换为内存地址或寄存器

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

分配, 注释则被完全移除. 这有效地降低了水印载体信息的 “熵” 或特异性, 如果水印依赖这些信息进行编码, 其痕迹将在此阶段被有效清除或初步削弱.

• **代码结构重构与编译器优化.** 编译器根据优化级别对代码进行重组和转换, 以提高执行效率. 这包括但不限于函数调用约定、数据布局和控制流图的修改. 令 P_{wm} 表示嵌入在 C_{wm} 中的特定水印模式或结构. 编译器优化 O 将 C_{wm} 转换为 B, 使得原始水印模式 P_{wm} 极不可能在 B 或随后的 C_{de} 中保留, 即 $P_{wm}(C_{wm}) \neq P_{wm}(\text{Decompile}(\text{Compile}(C_{wm})))$. 这些优化通过以下方式破坏水印.

(1) 死代码消除. 移除对程序功能没有影响的代码段, 直接移除通过插入无用代码嵌入的水印载体.

(2) 函数内联. 改变程序的控制流图, 打破基于函数调用结构的水印模式.

(3) 循环展开. 显著改变循环的结构和长度, 破坏基于循环模式的水印.

(4) 常量传播与折叠. 简化代码并消除冗余, 可能破坏依赖于特定表达式结构的水印.

(5) 变量重命名与寄存器分配. 自动变量重命名 (即使是反编译工具的产物) 直接破坏了依赖于特定变量名或哈希值的水印方案.

这些机制共同作用, 为后续的深度水印去除奠定基础. 尽管水印设计者可能尝试在编译层嵌入逻辑, 但高级别优化往往能有效混淆或消除这些二进制层面的水印特征.

3.3.2　第 2 阶段: 反编译与水印深度去除

在得到二进制代码 B 后, 我们进一步利用专业反编译工具 (如 IDA Pro 等) 将其恢复至源代码级别, 得到伪代码 C_{de}. 由于反编译过程本身具备抽象层次转换的特性, 因此生成的伪代码相比原始代码在代码结构等方面会发生变化, 从而进一步去除原始水印的痕迹. 然而, 在这一过程中会面临着另一个问题: 在反编译过程中, 针对程序结构复杂的情况下, 可能会出现相关代码解析不全、反编译器生成的非标准输出处理不当的情况. 由于此类问题的存在, 虽然上述方法可以有效消除代码水印, 但是相比原始代码, 其自身功能发生了变化, 无法满足代码水印消除最本质的需求. 如何在水印去除的同时尽可能恢复原始结构, 保证反编译结果的完整性与一致性, 是需要解决的首要难题. 因此, 我们设计了下列方法.

• **自动化函数调用图构建.** 在二进制可执行文件的逆向工程中, 理解程序的完整结构是一个重要挑战. 我们提出算法 1 (子函数调用递归搜索算法), 从指定入口点开始构建完整的函数调用图. 算法 1 的有效性可以通过图 4 的示例加以说明. 该算法确保了对函数调用关系的深度遍历, 即便 main 函数调用的 check_list_value 函数内部存在对其他自定义函数的调用, 也可以被完整地识别与处理. 该算法通过使用栈管理递归深度, 有效规避了传统递归方法的缺陷, 从而能够稳健地遍历所有函数依赖关系, 最终生成完整的函数集合以供后续分析.

• **伪代码初步优化与重构.** 反编译生成的伪代码通常包含冗余模式、编译器特定关键字和格式不一致等问题. 为解决这一问题, 我们开发了算法 2 (代码替换与优化算法). 该算法采用基于规则的替换策略, 将编译器特定模式替换为标准表示形式, 移除不必要代码, 并进行格式化处理, 提高代码的初步可读性和一致性. 如图 4 所示, 在反编译代码 (decompiled code) 中诸如 `std::operator<<<...` 这类冗长的表达就是算法 2 要处理的典型目标.

此阶段的重点在于尽可能准确地恢复程序的函数调用关系和控制流, 并通过初步的自动化清理, 为后续的 LLM 优化做好准备.

3.3.3　第 3 阶段: LLM 优化与可读性最终增强

尽管经过编译 – 反编译流程后, C_{de} 中的大部分水印已被去除, 但由于反编译过程自身的问题, 生成的伪代码会面临着如底层抽象暴露、控制流复杂冗长, 以及缺乏高级语言的自然表达等问题, 导致代码的可读性和可用性不足. 为此, 在第 3 阶段我们利用大语言模型 (LLMs) 对伪代码 C_{de} 进行优化,

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

Algorithm 1 Recursive search of subfunction calls.

Input: Function name f_{start}, decompilation function Decompile($\cdot$).
Iteration: Initialize $\mathcal{F} \leftarrow \varnothing$; ▷ Set to store processed functions
Initialize a stack $\mathcal{S} \leftarrow [f_{\text{start}}]$; ▷ Stack to manage recursion
while $\mathcal{S}$ is not empty **do**
  Pop f_{current} from $\mathcal{S}$;
  if $f_{\text{current}} \notin \mathcal{F}$ **then**
    Add f_{current} to $\mathcal{F}$;
    $C_{\text{subs}} \leftarrow$ FindSubfunctions(Decompile(f_{current}));
    for each $f_{\text{sub}} \in C_{\text{subs}}$ **do**
      if $f_{\text{sub}} \notin \mathcal{F}$ **then**
        Push f_{sub} onto $\mathcal{S}$;
      end if
    end for
  end if
end while
Output: A set of all subfunctions $\mathcal{F}$.

Algorithm 2 Code replacement and optimization.

Input: Decompiled code C, a set of replacement rules $\mathcal{R}$.
Iteration: Initialize $C' \leftarrow C$; {Start with the original code}
for each replacement rule $(r_{\text{old}}, r_{\text{new}}) \in \mathcal{R}$ **do**
  Replace all occurrences of r_{old} in C' with r_{new};
end for
Remove unnecessary lines in C' (e.g., placeholder code, empty statements);
Format C' to ensure consistent indentation and structure;
Output: Optimized code C'.

旨在生成最终的高质量、高可读性的代码 C_{opt}.

为了实现这个目标, 一个直接的策略是由大语言模型对伪代码进行端到端的优化. 然而, 直接处理含有大量语法噪音和冗余结构的大规模伪代码, 会极大地增加 LLM 的推理负担, 容易诱发模型产生逻辑错误或代码幻觉 (hallucinations), 从而降低了代码重构的可靠性与稳定性.

为系统性地解决这些问题, 我们设计了一套基于 LLM 的自动化代码优化策略. 该策略的核心在于针对反编译代码的常见缺陷, 从以下几个关键维度进行精细化改进: (1) 增强代码结构清晰度; (2) 简化冗余或复杂的控制流; (3) 尝试恢复丢失的高级抽象概念与编程范式. 为了有效引导 LLM 在这些特定维度上进行优化, 我们精心设计了一系列目标导向的提示词 (prompt engineering). 如表 1 所示, 每个优化目标都对应一个核心提示指令, 并遵循特定的设计原则.

例如, 针对 “增强结构清晰度”, 我们的提示词 (见表 1, 第 1 行) 明确引导 LLM 关注代码的宏观结构模块化和格式一致性. 而对于 “恢复高级抽象概念”, 提示词则会告知 LLM 输入代码可能来源于反编译, 鼓励其识别并重建更符合人类认知的编程构造. 所有提示词均包含一个明确的指令 —— “Output only the refactored C++ code” —— 以确保 LLM 仅输出纯净的代码, 便于后续的自动化处理和评估, 避免了模型生成不必要的解释性文本. 这种基于细致提示工程的方法, 使得 LLM 能够更聚焦于特定的重构任务, 从而提升优化效果的精确性和可控性.

通过这种 LLM 驱动的优化, 我们不仅进一步提升了代码的标准化程度和风格一致性, 还有效修复了反编译过程中可能引入的结构性问题或不自然的表达. 相较于 Suresh 等人 [10] 提出的需要大模型执行多重转换 (包括变量重命名、死代码插入等多种操作) 的水印去除方案, 本方法在 LLM 优化阶段仅需执行一次目标明确的优化过程, 即可显著提升已脱水印代码的整体质量, 特别是可读性. 这一策略不仅大幅降低了计算资源消耗, 更重要的是, 它将研究目标从单一的水印去除扩展至代码可读性、

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

表 1　面向特定优化目标的 LLM 提示词设计.

Table 1　LLM prompt design for targeted code optimization.

Refinement goal	Core prompt instruction	Design principle/key elements
Enhance structural clarity	Refactor the following C function to improve its structural clarity by modularizing complex blocks and ensuring consistent formatting. Output only the refactored C code below: [CODE_PLACEHOLDER]	Guide the LLM to focus on the code's macrostructure and visual presentation; [CODE_PLACEHOLDER] is the code injection point; explicitly require code-only output.
Simplify redundant control flow	Analyze and simplify the control flow in the C function below, focusing on reducing nesting and eliminating redundant checks while preserving all original functionality. Output only the refactored C code: [CODE_PLACEHOLDER]	Emphasize functional preservation; explicitly specify the optimization directions (reduce nesting, remove redundancy); guide the model to conduct logic-level analysis.
Restore high-level abstractions	The following C code is likely from decompilation. Identify and restore higher-level programming abstractions or idiomatic C constructs to enhance conceptual clarity. Output only the refactored C code: [CODE_PLACEHOLDER]	Inform the LLM of decompilation-derived source context; guide pattern recognition and concept reconstruction; aim to improve the code's semantic interpretability.
General readability enhancement	Improve the overall readability and maintainability of the following C function. This includes refining variable names where context allows, simplifying expressions, and ensuring a clean, consistent style. Output only the refactored C code: [CODE_PLACEHOLDER]	Comprehensive optimization directive; allow the LLM to be fine-tuned across multiple aspects; avoid introducing excessive explanations or non-code content.

可维护性与可用性的综合提升. 实验结果充分证明, 经过 DeMark 第三阶段优化的代码 C_{opt} 不仅更为简洁高效, 其可读性和可维护性也得到了显著增强, 为后续的代码分析、人工审查和工程应用奠定了坚实的基础.

4　实验与分析

4.1　实验设置

实验平台. 所有实验均在 Ubuntu 22.04 系统的服务器上完成, 硬件配置包括: 8 核 AMD EYPC 7402 处理器 (主频 2.8 GHz)、128 GB 内存、12 TB 硬盘以及 4 块 NVIDIA A100 80G GPU. 编译工具链采用 GCC 11.2.0, 启用 C++17 标准与 -O0 优化; 反编译分析基于 IDA Pro 9.0.

模型配置. 我们使用 Doubao-1.5-pro-32k 对反编译后的代码进行优化. 作为水印嵌入模型, 我们采用以下两种大语言模型.

- Llama-3.1-8B. Meta 公司开发的 80 亿参数多语言大模型, 支持包括英语在内的 8 种语言, 具有 128k 的长上下文窗口, 适用于多语言商业和研究场景, 在效率与安全性方面表现突出 [18].
- StarCoder2. 基于 The Stack v2 数据集训练的 150 亿参数模型, 覆盖 600+ 编程语言. 该模型采用分组查询注意力机制, 支持 16384 token 的上下文窗口 (滑动窗口注意力为 4096 token), 在 4 万亿 token 上训练完成 [19].

水印设置. 在所有基线方法中, 我们遵循既有工作 [6,8,9] 的设定. 具体而言, 参数 γ 控制绿色词元的采样概率提升幅度, δ 控制红色词元的屏蔽强度; 检测阶段则采用基于 z-score 的统计检验, 并设定统一的阈值 z. 在此统一设置下, 我们对 3 种代表性方法进行比较.

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

• KGW [6]. 该方法是代码水印领域最早提出的方案之一, 其方法论对后续研究产生了深远影响, 许多衍生方法仍然继承了其在 token 层面进行嵌入的思路. 其通过构建 “绿 – 红” 词元列表并偏向选择 “绿色” 词元来嵌入信号, 检测方可利用统计检验高效识别机器生成文本. 其对文本质量影响较小且具备较好鲁棒性, 是该领域的一个重要基线. $\gamma = 0.5$, $\delta = 2.0$, $z = 4.0$.

• Unigram [8]. 该方法在整个生成过程中始终采用固定且全局统一的绿红名单, 避免了单个词元修改对整体信号的连锁影响, 从而显著提升了水印在编辑攻击下的鲁棒性. 属于语言模型水印的经典方案, 广泛用于文本和代码场景, 能够代表通用型水印方法. $\gamma = 0.5$, $\delta = 2.0$, $z = 4.0$.

• SWEET [9]. 该方法针对代码的低熵特性提出 “选择性” 水印策略, 仅在预测分布熵值较高时启用绿 – 红名单, 从而避免对功能关键的低熵词元造成干扰. 其在变量名、注释等高熵位置嵌入信号, 在保证功能完整性的同时显著提升了水印在代码领域的可用性与鲁棒性. 是近年来专门针对代码场景提出的改进型水印, 具备更强的鲁棒性和实用性. $\gamma = 0.5$, $\delta = 2.0$, 熵阈值 $e = 0.9$, $z = 4.0$.

因此, 这 3 类方法分别覆盖了早期奠基性方法、通用型水印方法, 以及针对代码优化的最新方法. 在这三者上验证 DeMark 的有效性, 能够间接说明其对绝大多数源码级水印方案均具备攻击潜力.

实验设置. 为评估攻击方法的实用性, 本文从 MultiPL-E 数据集 [20] 选取 161 个代码提示, 并采用 2.1 小节所述的文本水印方法生成响应代码. 所有提示均以英文提供, 而 LLM 生成的响应代码为 C++ 格式. 我们对比了以下两种方法.

• **代码重构 [10].** 通过不同方式 (如重命名、添加死代码、插入打印语句、封装 try-catch 等) 重构代码响应, 同时保持功能不变.

• **代码翻译 [17].** 将 C++ 代码先转换为中间语言 (如 Python 等), 再转回原 C++. 这些带有水印的代码响应通过 `Doubao-1.5-pro-32k` 被翻译成 3 种编程语言: Python, Java 和 C#. 在语言层面上, C++ 与 Java 和 C# 具有更高的相似性, 而与 Python 存在显著差异.

值得注意的是, 将编译 – 反编译作为一种攻击手段来去除大语言模型生成代码中的水印, 是一个新兴的研究方向. 我们的目标不仅是破坏水印, 还需将代码恢复至高质量、可读的源代码形式, 以便于后续的开发和维护. 据我们所知, 目前尚无公开的、专门针对此场景的同类攻击方法可供直接比较. 因此, 我们的工作不仅提出了一种有效的方法, 也为评估未来代码水印技术的鲁棒性提供了一个全新的攻击基准.

4.2　实验结果

4.2.1　水印攻击技术的对比分析

通过观察表 2 和 3 可以看出, DeMark 展现出最有效的攻击性能, 显著降低了 AUROC 和 TPR 值. 原因在于, 大多数现有水印技术并非为代码 (除 SWEET 外) 或跨语言代码转换设计, 这导致不同语言间水印一致性较弱. 此外, 自然语言文本中的重翻译和转述策略本质上是语义保持的文本重写, 这些策略通常会保留原始响应中的部分 n-gram 特征, 仍可能被水印检测算法识别. 然而在代码领域, 编程语言的语法规则差异巨大, 例如 Python 使用缩进表示代码块, 而 Java 和 C++ 依赖花括号. 确保这些结构性差异下的水印一致性是一个重大挑战. 因此, 无论是代码重构还是跨语言转换都会导致此类 n-gram 特征的减少. DeMark 性能最优的原因在于其通过编译和反编译优化, 正如 3.1.1 和 3.1.2 小节所详述的, 编译过程中的信息丢失、编译器优化以及反编译过程中的结构重构, 将代码转换为无水印的低级表示, 然后最大程度地重建代码功能.

4.2.2　DeMark 攻击对不同水印方案的有效性分析

从图 5 和表 2 可以明显看出, 在 DeMark 攻击下, 专为代码水印设计的 SWEET 技术展现出比其他水印方法更强的鲁棒性. KGW 和 Unigram 在 DeMark 攻击下的 AUROC 值分别下降了 30.16% 和

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

表 2　不同攻击方法对各类水印的有效性评估. DeMark 在 3 项关键实验中均优于基线方法 (Unigram, SWEET), 衡量指标包括 PASS@1 (%), AUROC, TPR 和 FPR (阈值 < 0.10). 粗体显示最佳结果.

Table 2 Effectiveness evaluation of different attack methods against various watermarking techniques. DeMark outperforms baseline approaches (Unigram, SWEET) across three key experiments, with metrics including PASS@1 (%), AUROC, TPR, and FPR (threshold < 0.10). Best results are in bold.

Attack	KGW				Unigram				SWEET			
	PASS@1	AUROC	TPR	FPR	PASS@1	AUROC	TPR	FPR	PASS@1	AUROC	TPR	FPR
Model 1: Llama-3.1-8B												
Origin	49.7	0.945	0.888	< 0.10	39.1	0.575	0.174	< 0.10	49.1	0.980	0.957	< 0.10
+ Refactor	87.6	0.778	0.280	< 0.10	83.2	0.445	0.012	< 0.10	88.2	0.826	0.484	< 0.10
+ Ours (DeMark)	**90.7**	**0.660**	**0.137**	< 0.10	**89.4**	**0.259**	**0.000**	< 0.10	**91.3**	**0.800**	**0.478**	< 0.10
Model 2: StarCoder2-15B												
Origin	14.9	0.985	0.963	< 0.10	21.7	0.930	0.795	< 0.10	23.6	0.970	0.938	< 0.10
+ Refactor	75.3	**0.772**	0.180	< 0.10	77.0	0.769	0.205	< 0.10	**84.5**	0.809	0.466	< 0.10
+ Ours (DeMark)	**92.5**	0.821	**0.161**	< 0.10	**78.3**	**0.752**	**0.180**	< 0.10	80.7	**0.624**	**0.236**	< 0.10

表 3　使用 Llama3.1-8B 对 DeMark 方法与基于翻译的方法在 C#, Java 和 Python 中的攻击有效性对比.

Table 3 Comparison of attack effectiveness for DeMark and translation-based methods across C#, Java, and Python using Llama3.1-8B.

TransLang	KGW				Unigram				SWEET			
	PASS@1	AUROC	TPR	FPR	PASS@1	AUROC	TPR	FPR	PASS@1	AUROC	TPR	FPR
Origin	49.7	0.945	0.888	< 0.10	39.1	0.575	0.174	< 0.10	49.1	0.980	0.957	< 0.10
$\sim$ Python	83.9	0.757	0.230	< 0.10	82.0	0.457	0.006	< 0.10	88.8	0.869	0.634	< 0.10
$\sim$ Java	91.3	0.767	0.304	< 0.10	85.7	0.414	0.000	< 0.10	91.9	0.872	0.646	< 0.10
$\sim$ C#	**91.9**	0.788	0.323	< 0.10	84.5	0.428	0.000	< 0.10	**95.0**	0.879	0.696	< 0.10
+ Ours (DeMark)	90.7	**0.660**	**0.137**	< 0.10	**89.4**	**0.259**	**0.000**	< 0.10	91.3	**0.800**	**0.478**	< 0.10

54.96%, 接近随机猜测水平. 相比之下, SWEET 方法的 AUROC 虽然也有所下降, 但从 0.98 降至 0.8, 仅下降 18.37%, 仍保持较高水平.

这一显著性能差异表明, 即使在 DeMark 这种严苛的水印攻击场景下, SWEET 仍能保持较好的有效性. 其相对较小的 AUROC 降幅进一步凸显了相较于其他方法的鲁棒性和抗干扰能力, 强化了其在实际应用 (尤其是对抗性或易失真环境中) 作为可靠水印技术的潜力.

值得注意的是, SWEET 较高的 AUROC 值可能源于其采用了一种旨在最大化召回率 (recall) 的严苛检测策略. 通过提高判别严格度, SWEET 可能更倾向于识别潜在水印目标, 即使代价是略高的误报率. 虽然这种策略可能导致一些错误拒绝, 但有效降低了漏报风险, 从而提升了水印检测的整体准确性和鲁棒性.

4.2.3　代码质量评估

如表 2 所示, 本研究采用 PASS@1 指标评估代码编译通过率, 使用 161 道测试题目 (源自 MultiPL-E 数据集) 进行评估. 实验结果表明, 部分攻击方法在去除水印的同时, 甚至能提升代码质量. 这主要是因为水印算法的嵌入有时会引入结构性问题 (如头文件位置错误), 而大模型在后续处理中可能修正这些瑕疵. DeMark 方法在此方面表现尤为突出. 其独特的编译 – 反编译方法 (如 3.1 小节所述) 是关键: 编译阶段不仅初步混淆水印, 其固有的优化机制也会修正部分由水印算法引入的低级错误; 反编译阶段虽然会丢失部分原始信息, 但其目标是重建功能正确的代码. 更重要的是, DeMark 最终利用大语言模型 (如 3.1.3 小节所述, 并结合表 1 的提示工程) 对反编译产物进行智能优化和重构. 这一阶段不

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

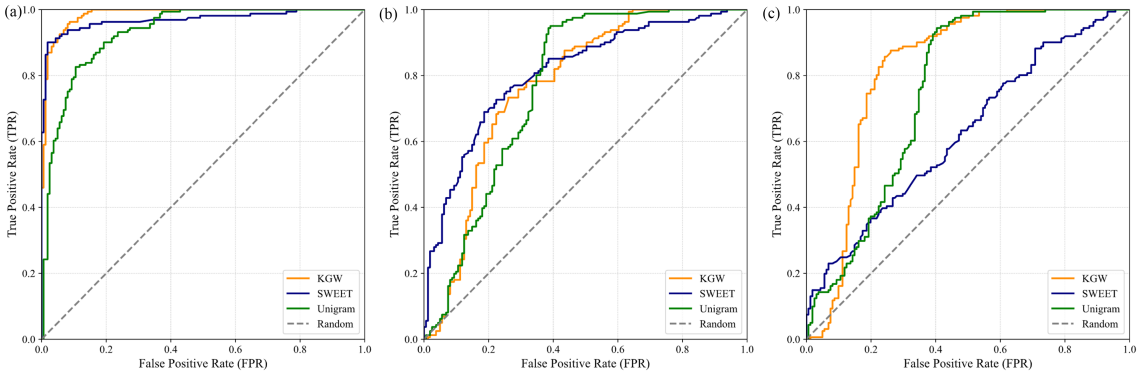

图 5　(网络版彩图) 在两种不同对抗攻击场景下, 3 种水印算法在 StarCoder2-15B 模型上的 ROC 曲线对比分析. (a) Origin; (b) Refactor; (c) DeMark.

Figure 5　(Color online) Comparative analysis of ROC curves for three watermarking algorithms evaluated on the StarCoder2-15B model under two distinct adversarial attack scenarios. (a) Origin; (b) Refactor; (c) DeMark.

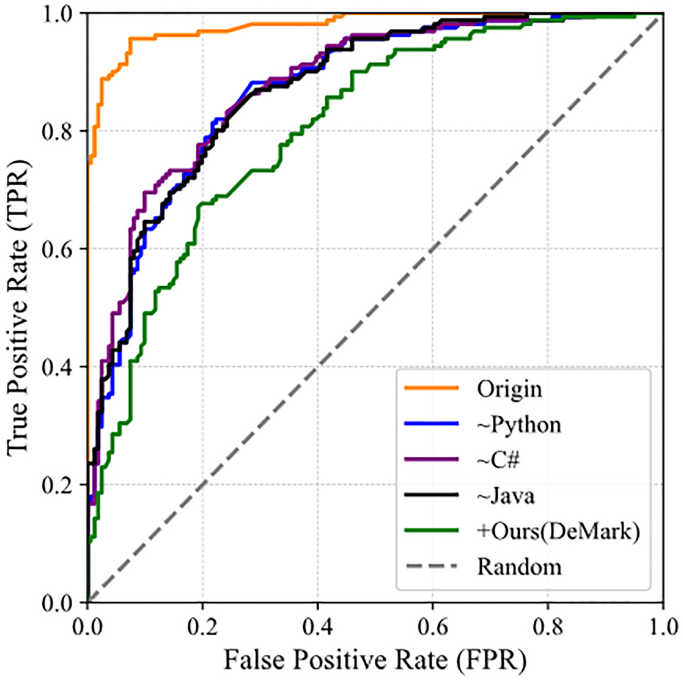

图 6　(网络版彩图) 在 Llama3.1-8B 模型上, 采用 SWEET 水印技术, 对 3 种编程语言 (Python, C#, Java) 实现的 ROC 曲线与本文提出方法的对比分析.

Figure 6　(Color online) Comparative analysis of ROC curves across three programming language implementations (Python, C#, Java) in contrast to DeMark, employing the SWEET watermarking technique on the Llama3.1-8B.

仅能进一步清除水印痕迹, 修复反编译可能引入的缺陷, 还能显著提升代码的清晰度、可读性和整体结构质量. 因此, 经 DeMark 处理后的代码, 在可执行性、稳定性和可维护性上均展现出显著优势. 这种 PASS@1 指标的提升, 尤其是在 DeMark 方法中, 显著归功于其第 3 阶段的 LLM 优化, 它不仅清除了水印残留, 还对代码的整体质量进行了提升.

4.2.4　代码相似性对水印持久性的影响

如表 3 和图 6 所示, 在代码翻译过程中, 水印的保留强度与源语言和目标语言的语法相似度密切相关: C# 和 Java 作为静态类型语言, 与 C++ 具有较高的结构相似性 (如显式类型声明、分号终止

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

表 4　在不同 G++ 编译器优化等级与 Llama3.1-8B 模型下, 不同攻击方法的去水印有效性对比. 数据显示了去水印的成功率 (%). 更高的成功率表示更强的攻击效果.

Table 4 Comparison of watermark-removal effectiveness across different attack methods under various G++ compiler optimization levels with Llama3.1-8B. The reported values indicate success rates (%); a higher rate signifies a more potent attack.

Levels	KGW				Unigram				SWEET			
	PASS@1	AUROC	TPR	FPR	PASS@1	AUROC	TPR	FPR	PASS@1	AUROC	TPR	FPR
Origin	49.7	0.945	0.888	< 0.10	39.1	0.575	0.174	< 0.10	49.1	0.980	0.957	< 0.10
-O0	90.7	0.660	0.137	< 0.10	89.4	0.259	0.000	< 0.10	91.3	0.800	0.478	< 0.10
-O1	**91.9**	**0.650**	0.081	< 0.10	87.0	**0.225**	0.000	< 0.10	92.5	0.787	0.435	< 0.10
-O2	90.7	0.664	0.087	< 0.10	**89.4**	0.232	0.000	< 0.10	**91.9**	**0.785**	0.435	< 0.10
-O3	87.6	0.655	**0.068**	< 0.10	88.8	0.236	**0.000**	< 0.10	90.7	0.787	**0.379**	< 0.10

符、类/方法定义), 在翻译过程中能保留更多原始文本模式和逻辑结构, 从而有利于依赖文本模式 (如 KGW) 或代码逻辑 (如 SWEET) 的水印机制持续存在. 相比之下, Python 作为动态类型语言, 其语法简洁灵活 (如隐式类型、基于缩进的控制流、简化的语法符号), 在翻译过程中需要进行结构抽象和简化 (如移除类型声明、重组控制语句), 导致依赖特定语法的隐写特征被弱化或消除. 此外, 针对语法差异较大的语言的翻译工具往往优先优化代码可读性, 这进一步剥离了与水印相关的冗余或非功能性元素.

与代码翻译相比, 编译 – 反编译处理通过多维机制实现更彻底的水印去除: 正如 3.1.1 小节所述, 编译器在编译过程中执行底层优化 (如注释删除、表达式简化、函数内联、符号重命名), 生成与原始逻辑解耦的中间表示 (如机器码、字节码), 而反编译器只能从这些优化后的表示中重建近似逻辑 —— 这种不可逆的破坏性转换直接消除了依赖语法的水印 (如 KGW 的文本冗余、SWEET 的隐写逻辑). 同时, 编译过程会丢弃所有非功能性信息 (如变量/函数名、注释), 反编译后的标识符被替换为通用符号 (如 func1, var2), 从而破坏依赖命名规范的水印方案 (如基于哈希的变量嵌入) —— 这一漏洞在代码翻译中较少出现, 因其常保留部分原始符号. 此外, 编译器驱动的逻辑等价转换 (如循环展开、分支合并、算术简化) 会破坏隐写模式 (如 SWEET 的冗余条件分支), 而代码翻译则优先保证语义等价和结构保留. 最后, 编译 – 反编译循环通过高级与低级表示之间的双向转换降低抽象层级, 生成类伪代码输出 (如从反编译 C++ 得到的低抽象度、可读性较差的 C++ 代码), 可以实现代码混淆, 从而降低水印的可检测性, 而代码翻译 (如 C++→Python) 生成的是人类可读的输出, 有可能保留可识别的水印特征.

4.2.5　不同编译等级对于水印去除效果的影响

为分析编译优化强度对去水印效果的影响, 我们在 G++ 的 4 个优化等级 (-O0/-O1/-O2/-O3) 下进行对比实验 (见表 4). 总体趋势清晰: 即便在不启用激进优化的 -O0 条件下, 仅 “编译 → 反编译” 的抽象层转换已足以显著削弱主流水印检测器的判别能力, 说明该流程本身的信息有损于结构重构, 能够破坏大量可检测信号.

在此基础上, 提高优化等级带来 “边际但稳定” 的额外收益: 从 -O1 优化开始, 编译器便会系统地触发死代码删除、常量折叠与基本块/CFG 重写等关键变换, 使依赖词元分布或语法模式的水印进一步退化; -O2/-O3 在循环变换、内联与向量化等方面的更强重构, 继续压低检测器真阳性率, 但整体收益呈现饱和.

不同水印方案对优化强度的敏感度存在差异: 以 Unigram 为例, 在 -O0 已接近随机判别, 继续提升优化强度仅引发轻微波动; 而面向代码场景设计的 SWEET 则在高等级优化下仍保持一定鲁棒性, 但其真阳性率与整体可分性仍随优化增强而下降.

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

值得注意的是, 除编译 → 反编译带来的信息有损外, LLM 在后续源码修复与重构阶段还引入了“风格归一化 (style canonicalization)” 效应: 模型为提升可读性与一致性, 会优先采用高频编码习惯与通用结构模板 (例如规范化的循环/条件展开与命名/缩进风格). 该过程对词法与局部结构的扰动往往强于 -O1 以上优化带来的增量差异, 因而在反编译后源码层面, 来自 -O1/-O2/-O3 的细粒度差别被系统性压缩, 表现为检测指标仅有小幅波动. 换言之, 模型容量越强、重构越充分, 跨优化等级的 “可分辨度” 越低, 这与我们在表 4 中观察到的边际收益一致.

5　讨论: DeMark 的通用性与适用范围

现有的主流水印方法大多属于基于统计特征的嵌入策略. 在这一类方法中, KGW, Unigram 与 SWEET 分别代表了早期的奠基性方案、经过理论化简的通用方案, 以及针对代码低熵特性提出的改进方案 [6,8,9]. 因此, 可以认为这 3 类方法已经覆盖了绝大多数典型的源码级水印设计范式. 然而, 无论采用哪种具体策略, 它们的共同点都是依赖于词元的统计分布偏移. 这类信号在面对结构性变换 (如语法重写、跨语言转换, 尤其是本文提出的编译 – 反编译过程) 时往往会被系统性破坏, 导致水印检测能力显著下降. 基于这一观察, 本文提出的攻击方法不仅能够对单一水印方案生效, 还能对现有主流方法均具有普适性. 这表明 DeMark 攻击具备跨方法的通用性, 能够为未来水印设计与对抗研究提供新的视角.

6　结论

本文提出了一种名为 DeMark 的创新方法, 旨在高效去除大语言模型 (LLM) 生成的代码中的水印. DeMark 采用一种新颖的编译 – 反编译 – 优化三阶段流程: 首先通过编译过程中的代码转换和固有的信息丢失来初步混淆水印; 接着, 利用反编译重建功能代码, 此过程会进一步破坏水印结构; 最后, 借助大语言模型的智能优化能力, 提升代码的可读性和规范性, 同时确保核心功能的完整性.

实验结果充分证明, DeMark 能够有效破坏多种主流水印技术 (包括 KGW, Unigram 及针对代码场景优化的 SWEET) 的可检测性, 其表现在 AUROC 和 TPR 等关键指标上显著优于传统的代码重构或跨语言翻译等攻击手段. 更重要的是, DeMark 不仅成功移除了水印, 还在保持甚至提升代码性能的同时, 显著改善了代码的可读性. 这项工作揭示了当前代码水印技术在面对编译和反编译这类深度代码转换时的脆弱性, 强调了这些转换过程对代码结构和语义的根本性改变足以抹除现有的水印痕迹.

鉴于大语言模型在软件开发中日益核心的地位, 确保生成代码的来源可追溯性、安全性与可靠性变得至关重要. 因此, 我们呼吁未来的研究致力于开发更为鲁棒的水印方案. 这些方案应考虑将水印信息嵌入到代码的更深层、更稳定的表征中, 例如抽象语法树 (AST)、中间表示 (IR) 或代码的语义逻辑层面, 以抵抗类似 DeMark 的转换攻击. 此外, 未来的工作还应深入探索针对基于编译 – 反编译的水印去除方法的防御策略, 评估现有水印技术在此类攻击下的真实鲁棒性, 并研究能够在编译、反编译及优化流程中保持水印完整性的新机制. 这对于维护代码知识产权、追踪代码来源以及保障软件供应链安全具有深远意义.

参考文献

1　Guo D, Yang D, Zhang H, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. Nature, 2025, 645: 633–638
2　Yang A, Yang B, Hui B, et al. Qwen2 technical report, 2024. ArXiv:2407.10671
3　OpenAI. ChatGPT. 2023. https://openai.com/blog/chatgpt/
4　Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code. 2021. ArXiv:2107.03374
https://www.sciengine.com/doi/10.1360/SSI-2025-0255

5 Ugare S, Suresh T, Banerjee D, et al. Incremental randomized smoothing certification. In: Proceedings of the 12th International Conference on Learning Representations, 2024

6 Kirchenbauer J, Geiping J, Wen Y, et al. A watermark for large language models. In: Proceedings of the 40th International Conference on Machine Learning, 2023. 17061–17084

7 Kuditipudi R, Thickstun J, Hashimoto T, et al. Robust distortion-free watermarks for language models. 2024. ArXiv:2307.15593

8 Zhao X, Ananth P, Li L, et al. Provable robust watermarking for AI-generated text. In: Proceedings of International Conference on Representation Learning, 2024. 43738–43772

9 Lee T, Hong S, Ahn J, et al. Who wrote this code? Watermarking for code generation. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, 2024. 4890–4911

10 Suresh T, Ugare S, Singh G, et al. Is watermarking LLM-generated code robust? In: Proceedings of the 2nd Tiny Papers Track at ICLR, 2024

11 Krishna K, Song Y, Karpinska M, et al. Paraphrasing evades detectors of AI-generated text, but retrieval is an effective defense. In: Proceedings of the 37th International Conference on Neural Information Processing Systems, 2024

12 Liu A, Pan L, Lu Y, et al. A survey of text watermarking in the era of large language models. ACM Comput Surv, 2025, 57: 1–36

13 Abdelnabi S, Fritz M. Adversarial watermarking transformer: towards tracing text provenance with data hiding. In: Proceedings of IEEE Symposium on Security and Privacy (SP), 2021. 121–140

14 Yang X, Zhang J, Chen K, et al. Tracing text provenance via context-aware lexical substitution. AAAI, 2022, 36: 11613–11621

15 Aaronson S, Kirchner H. Watermarking GPT outputs. 2022. https://www.scottaaronson.com/talks/watermark.ppt

16 Christ M, Gunn S, Zamir O. Undetectable watermarks for language models. In: Proceedings of the 37th Conference on Learning Theory, 2024. 1125–1139

17 He Z, Zhou B, Hao H, et al. Can watermarks survive translation? On the cross-lingual consistency of text watermark for large language models. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, 2024. 4115–4129

18 Grattafiori A, Dubey A, Jauhri A, et al. The llama 3 herd of models. 2024. ArXiv:2407.21783

19 Lozhkov A, Li R, Allal L B, et al. StarCoder 2 and The Stack v2: the next Generation. 2024. ArXiv:2402.19173

20 Cassano F, Gouwar J, Nguyen D, et al. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. IIEEE Trans Softw Eng, 2023, 49: 3675–3691

https://www.sciengine.com/doi/10.1360/SSI-2025-0255

From compile to decompile: efficient watermark removal via source-to-source transformations

Hao HUANG[1,2], Ruihua ZHOU[1,2], Jiatang LUO[3], Xiaohuan MA[1,2], Yunpeng LI[1,2] & Yuling LIU[1,2*]

1. *Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100085, China*
2. *School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China*
3. *School of Frontier Cross Science, University of Chinese Academy of Sciences, Beijing 100049, China*
* Corresponding author. E-mail: liuyuling@iie.ac.cn

Abstract While large language models (LLMs) efficiently generate code, they also introduce potential risks regarding copyright attribution and code misuse. To address this, LLM-watermarking techniques have emerged to trace code provenance. However, when applied to code, a unique medium, their effectiveness and robustness face severe challenges. Existing abstract syntax tree (AST)-based attack strategies for evaluating code watermark robustness suffer from inefficiencies and instability. To overcome this, this paper proposes DeMark, a novel compilation-decompilation-based watermark removal method. This method first compiles watermarked code into a low-level binary representation, leveraging compiler optimizations to eliminate watermark traces. Subsequently, using an innovative automated function call graph construction algorithm, it decompiles the optimized binary code back into high-level language code that preserves core functionality and readability, thereby efficiently removing the watermark. Experimental results demonstrate that DeMark exhibits significant effectiveness and superior adaptability in attacking existing watermarking methods, particularly when dealing with watermarks involving complex semantic-preserving transformations. Furthermore, this study pioneers a systematic evaluation of the robustness of watermarks in compiled languages like C++ under code refactoring and cross-language conversion scenarios. It reveals the inherent weaknesses of current text-based watermarking algorithms when applied to code generation and offers valuable insights for designing more robust code provenance mechanisms in the future.

Keywords secure watermark removal, LLM security, trustworthy code generation, secure compilation, adversarial robustness

https://www.sciengine.com/doi/10.1360/SSI-2025-0255