法律技术专栏｜大模型已经可以“破解”软件源码了吗？（上篇）——从二进制到“可运行代码”

原创 吴圣健 Law Review 2026年8月7日 12:21 美国

作者：吴圣健｜ 新加坡法律出版社特邀技术顾问

导语｜过去，拿到一个软件的安装包，并不意味着能够看懂它的源码。但大模型正在改变这件事。从2024年的LLM4Decompile，到2026年开始利用原程序运行结果不断纠错，AI反编译已经从“生成可读伪代码”，推进到了“重新编译、重新运行，并在特定测试中验证行为是否一致”。那么，AI现在真的已经能把软件“还原”成源码了吗？

当一份二进制被AI重新变成代码

一家软件公司向客户交付安装包、固件或者可执行文件，却从不交付源代码。长期以来，这不仅是一种交付方式，也是一道现实的技术防线：客户可以使用软件，却很难知道程序内部究竟如何实现；竞争者即使拿到程序文件，也需要专业工具和经验丰富的逆向工程人员，投入大量时间才能逐步理解其中的算法、数据结构和业务逻辑。

但这道防线是否正在被大语言模型削弱？

过去，反编译工具更多是把机器代码变成一份难读的“分析稿”；现在，大模型开始尝试把这份分析稿继续整理成正常代码，再交给编译器和测试程序不断“批改”。写错了就继续改，直到代码不仅能编译，还能尽量跑出和原程序一样的结果。

研究目标正在从“让反编译结果更容易读懂”，推进到“生成能够重新编译，并在测试范围内准确运行的替代代码”。

而这条路线真正快速发展，不过是最近两年。

2024年，LLM4Decompile发表于自然语言处理领域的重要国际会议EMNLP，推出了专门面向二进制反编译训练的开源模型；2025年，DecLLM把编译错误和运行反馈引入修复流程；到2026年，PCodeTrans、Agent4Decompile和AutoDecompiler等工作进一步把动态测试、多代理协作和多轮自我修正纳入反编译过程。[1]-[5]

从“专门训练一个会反编译的模型”，到“让AI像开发人员一样根据报错调试”，再到“用原程序的运行行为作为标准答案”，这条路线在两年左右的时间里完成了明显跃迁。

图1 LLM反编译代表性进展时间线（依据文献[1]-[5]整理）

于是，一个颇具冲击力的问题出现了：

当AI已经能够从二进制中重建可运行代码，企业长期依赖的“只交付二进制、不公开源码”，是否还足以保护核心算法？

一、从“看懂代码”到“重新运行”：LLM反编译发生了什么变化

1、反编译并不是大模型出现后才有

IDA、Ghidra、Hex-Rays等传统工具早已能够分析二进制程序。它们可以识别函数、条件判断、循环和调用关系，再生成一种看起来接近C语言的伪代码。对专业逆向人员而言，这已经足以成为理解程序的重要起点。

问题在于，这类结果往往只是一份“C语言风格的分析稿”，而不是可以直接交给开发团队维护的源码。它可能充满 `sub_401230` 、 `a1` 、 `v7` 之类的临时名称，数据结构和函数含义也需要分析人员继续推测。

2、为什么传统反编译无法恢复开发者原来的写法

源代码变成二进制时，注释、原始变量名、文件划分、部分类型信息和开发者的编码风格通常不会保留；编译器还会为了提高运行效率，对原代码进行合并、删除和重排。最终留下的机器码，可能已经与开发者最初写下的代码结构明显不同。

因此，反编译并不是把一个文件简单“倒放”回去。工具可以分析程序做了什么，却未必知道开发者当初为什么这样写，也无法从二进制中取回已经消失的注释和名称。

3、大模型补上的，是语义理解和自动纠错

大模型的价值，不是替代传统工具完成所有底层分析，而是利用大量代码经验推测“这段程序在做什么”。

它可以给函数和变量起更合理的名称，推测数据结构的含义，并把机器式、难以阅读的代码重新整理成开发人员熟悉的程序结构。

更重要的是，大模型可以进入一个反复试错的闭环：

传统反编译器先产生草稿；LLM补全和重写；编译器指出错误；原程序和恢复程序再处理相同输入，通过运行结果的差异继续纠错。

传统反编译器负责恢复程序结构，大模型负责理解和重写，编译器负责挑错，原程序则充当“标准答案”。

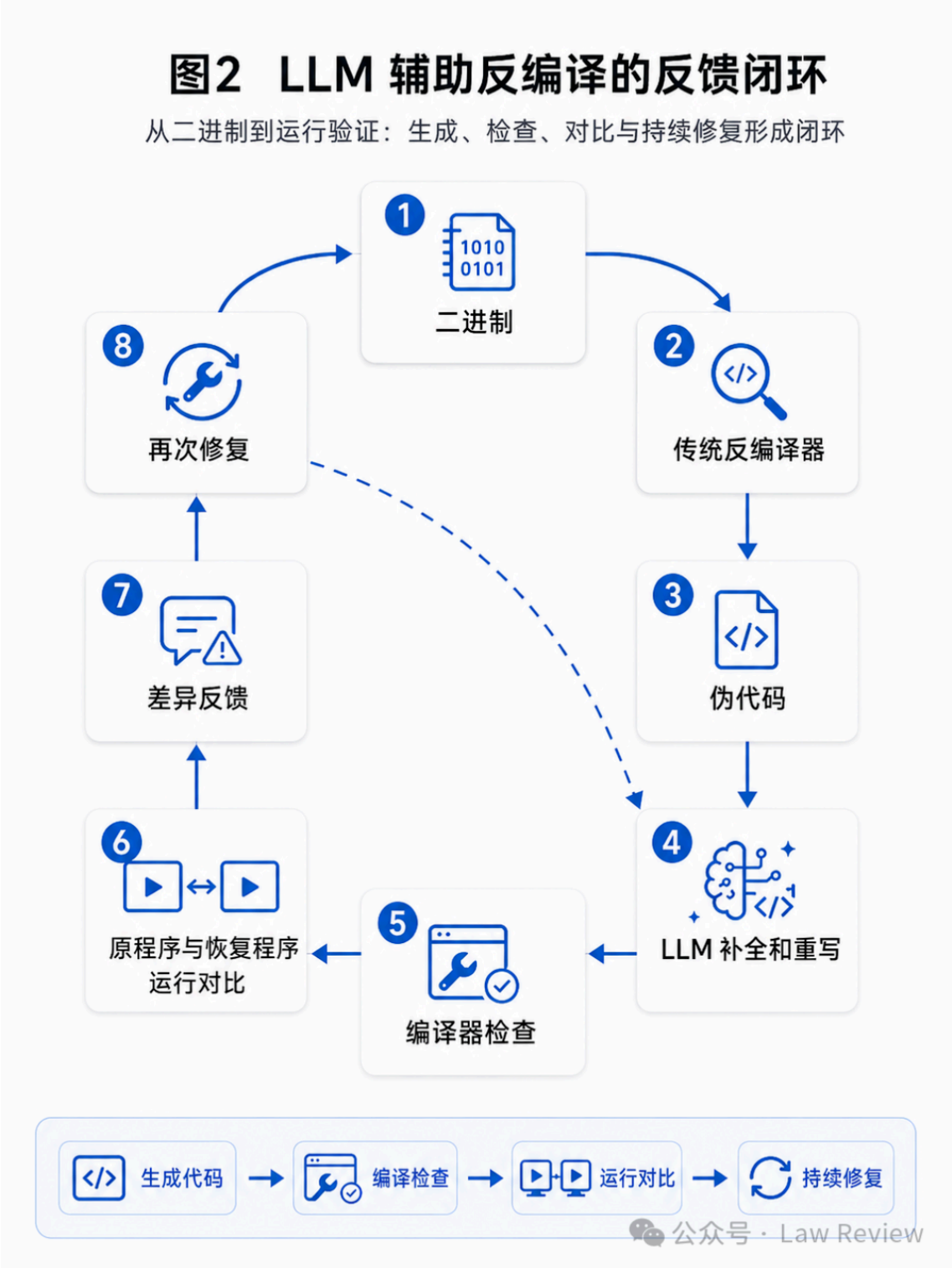

图2 LLM辅助反编译的反馈闭环（依据文献[2]-[5]整理）

这套组合真正改变的，是逆向工程的经济性。

过去，专业人员需要手工推测、修复和调试；现在，LLM可以承担其中一部分整理和试错工作。它未必能取代专家，却可能显著提高处理速度，并让更多普通开发人员具备初步分析和重建代码的能力。

二、从45%到特定实验中的99.5%：这些数字究竟说明了什么

说明｜不同研究的任务和验证条件并不相同，下面的数字更适合用来理解技术发展方向，而不是作为大型商业软件的“完整源码恢复率”。

1、2024年：LLM4Decompile证明大模型可以恢复一部分可运行代码

LLM4Decompile发表于EMNLP 2024，是这条快速发展路线的重要起点。它既可以直接把汇编代码转换成C代码，也可以先让Ghidra生成伪代码，再由专用模型进行修复和重写。[1]

在一组较短、相对简单的函数上，大约45.4%的恢复代码能够重新运行并通过测试；在更接近真实项目、依赖更多的函数中，这一比例约为18.0%。如果先让传统反编译器生成草稿，再由大模型修复，简单函数上的结果可以提高到约52.7%。

这组数字的意义很直观：大模型已经可以恢复一部分真正可运行的程序逻辑，但程序越复杂、依赖越多，难度就越高。

2、2025年：AI开始像开发人员一样根据编译错误调试

发表于ISSTA 2025的DecLLM把重点放在“让反编译代码重新进入编译器”。它不要求模型第一次就写对，而是把编译器诊断和运行反馈重新交给LLM，让模型继续修复。[2]

论文报告，在原本不能重新编译的反编译结果中，约七成可以被修复到重新编译成功。

也就是说，大量过去需要开发人员逐项处理的编译问题，开始能够交给AI自动完成。

3、2026年：研究重点从“编得过”转向“跑得对”

2026年的代表工作开始更强调动态验证。

Agent4Decompile让多个AI代理分别检查语法、编译和运行结果。在其v2实验中，只检查编译时，代码编译率可以达到99%—100%，但真正通过执行验证的比例仍可能只有32%—42%；加入执行反馈后，结果才进一步提高。[4]

“编得过”和“跑得对”是两回事。代码能够通过编译，只能说明它在形式上成立；它是否实现了原程序的功能，还要让两份程序做同一道题。

PCodeTrans则提供了另一种更充分的验证条件：把恢复函数重新编译，再逐个放回原二进制的运行环境，使用官方测试和执行轨迹继续纠错。

在GNU Coreutils和Binutils的函数级实验中，其测试验证行为一致率超过99.5%。[3]

这并不意味着任意大型软件都能获得99%的完整源码，而是说明：当模型拥有传统反编译结果、原程序运行环境、充分测试和持续反馈时，恢复代码可以达到很高的可用程度。

4、2026年：从一次生成走向多轮自我修正

同年出现的AutoDecompiler进一步训练模型利用多轮编译、执行和测试反馈持续修正，而不是生成一次后就停止。[5]

这一方向正在越来越像一个会自己调用工具、运行测试并不断调试的自动化工程师。

时间与工作	推进了哪一步	代表结果	普通读者可以怎样理解
2024 LLM4Decompile	生成、修复代码	简单函数45.4%；较复杂函数18.0%；修复Ghidra后52.7%	已能恢复部分可运行函数，复杂程序明显更难
2025 DecLLM	修复编译错误	约七成原本不可编译的结果被修到可以编译	编译错误开始能够大量交给AI处理
2026 Agent4Decompile v2	自动编译与执行验证	编译率99%—100%时，可重新执行率仍可能只有32%—42%	“编得过”并不等于“跑得对”
2026 PCodeTrans	逐函数运行验证	特定函数级实验中，行为一致率超过99.5%	反馈越充分，恢复代码越接近可用
2026 AutoDecompiler	多轮自我修正	同等模型和输入条件下优于单轮生成	研究正从一次生成走向自动调试

这些实验像几种难度不同的考试：

有的只给AI一段低层代码，让它独立写答案；有的先提供传统反编译器的草稿；有的允许不断查看编译错误；还有的让模型运行原程序、对比结果并持续改答案。

真正的趋势不是“源码恢复率从45%暴涨到99%”，而是AI获得的反馈越来越多，代码恢复正在从一次猜测变成持续调试。

三、99.5%是否意味着软件源码已经藏不住了？

1、一个函数，不等于整个软件

很多研究把程序拆成一个个函数进行测试。

一个商业软件可能包含成千上万个函数，还涉及界面、线程、数据库、网络和第三方库。单个函数能够恢复，并不等于整个软件可以一键还原。

2、能编译，不等于能正确运行

编译器主要检查语法、类型和符号是否成立。

一个本来应该做乘法的函数，即使被错误地写成加法，也可能顺利通过编译。

代码“编得过”只相当于语法考试及格；程序能否在相同输入下“跑得对”，才是功能考试。

3、通过测试，不等于对所有输入完全一致

论文中的“行为一致”通常意味着：在给定测试范围内，恢复代码与原程序产生了相同结果。

没有覆盖到的隐藏分支、异常输入和外部环境仍可能出错。

因此，实验中的99.5%并不是“任意软件99%的源码已经被找回”。

4、重建一份能工作的代码，不等于找回原始源码

开发者当初使用的注释、变量名、文件划分和编码风格，大多已经在编译过程中消失。

大模型能够根据剩余线索重新构造一个合理版本，却通常无法证明这就是开发者当初写下的原始工程。

一个较容易理解的比喻是：

> AI根据已经建成的房子，重新画出一套可以施工的图纸，而不是从房子里找回建筑师当年的原始草稿。

AI正在学会重建一个“能工作的版本”，而不是把开发者当年的源码原封不动找回来。

结语：真正值得讨论的问题才刚刚开始

回到文章开头的问题：只给AI一个软件，它现在能否反编译并“破解”出源码？

如果所谓“破解”是指完整找回开发者当初写下的变量名、注释、文件结构和原始工程，答案仍然是否定的；但如果它指理解程序逻辑，并生成能够重新编译、在一定测试范围内准确运行的替代代码，那么答案已经不再是简单的否定。

但技术问题回答到这里，一个更棘手的问题才真正出现：

如果大模型重新生成的代码已经换掉变量名、拆分函数、改变结构，甚至看起来与原源码完全不同，是否还能证明它来源于原软件？

如果恢复的是算法、规则或参数，而不是具体代码文本，又应该放在软件著作权、商业秘密还是其他法律框架下分析？企业又该怎样证明一套代码究竟来自独立开发，还是来自“反编译 + AI改写”？

这些问题，将是下篇真正要讨论的内容。

大模型还没有让所有软件变成透明的玻璃盒子，但“只要不交源码，别人就很难理解”的时代，可能正在逐渐过去。

附：文中几个概念

二进制文件：电脑可以直接加载和执行、但人很难直接阅读的程序文件，例如EXE、动态库和设备固件。

反编译：把机器码或汇编恢复成更接近C/C++等高级语言的伪代码或代码。

可重新编译：恢复代码可以被编译器接受，并生成新的目标文件或可执行文件。

可重新执行：重新编译后的代码能够运行，并通过规定的测试；它比“能编译”更严格。

行为一致：在给定输入和观察范围内，恢复程序与原程序产生相同结果。

主要参考资料

[1] Hanzhuo Tan et al., “LLM4Decompile: Decompiling Binary Code with Large Language Models,” EMNLP 2024.

[2] Wai Kin Wong et al., “DecLLM: LLM-Augmented Recompilable Decompilation for Enabling Programmatic Use of Decompiled Code,” ISSTA 2025.

[3] Yuxin Cui et al., “PCodeTrans: Translate Decompiled Pseudocode to Trace-Level Equivalent Source Code,” ASE 2026 Research Papers / arXiv:2603.14855.

[4] Yifan Zhang et al., “Constraint-Guided Multi-Agent Decompilation for Executable Binary Recovery,” arXiv:2604.23940v2, 2026.

[5] Peipei Liu et al., “Binary Decompilation LLM with Feedback-Driven Multi-Turn Refinement (AutoDecompiler),” arXiv:2606.16162, 2026.

[6] Luke Dramko et al., “A Taxonomy of C Decompiler Fidelity Issues,” USENIX Security 2024.

作者简介

吴圣健

新加坡法律出版社研究院

特邀技术顾问

吴圣健先生毕业于华东师范大学，先后取得软件工程本科学位及电子信息硕士学位。其长期从事人工智能、计算机视觉、图像处理、网络数据分析及软件系统研发工作，具有十余年技术研发与项目落地经验。技术方向涵盖目标检测、多目标跟踪、人脸识别、ReID、图像处理、模型部署推理、网络数据分析及后端系统开发等领域。曾以第一作者身份在AI领域的国际顶级会议（CCF-A类）上发表多篇论文，并作为第一完成人获得多项国家发明专利授权。其产业实践涉及无人智能零售图像识别、金融反欺诈、伪造检测、网络流量预警及后端系统建设等方向。