VIRBOX

专注软件安全，赋能数字未来

VirboxProtector

对抗 AI 自动化逆向分析白皮书

适用于二进制软件保护、AI 辅助逆向威胁分析与防护配置评估

版　本：**V1.0**

发布日期：**2026 年 5 月**

发布机构：**深盾科技软安实验室**

目录

VirboxProtector 对抗 AI 自动化逆向分析白皮书

一. 执行摘要

背景：AI 正在重塑软件逆向工程

随着大语言模型（LLM）和 AI Agent 工具链的成熟，过去依赖专家经验的软件逆向分析正在被自动化。对于包含本地核心逻辑的软件 — License 校验、核心算法、协议实现和关键业务流程 — 正在面临更低成本、更规模化的逆向分析风险。

当前，以 OpenAI GPT 系列、Anthropic Claude、DeepSeek、Google Gemini 为代表的通用大模型，已被攻击者与反编译工具（IDA Pro、Ghidra）通过 MCP（Model Context Protocol）协议深度集成，构建全自动化逆向分析 Agent，使得：

- **逆向分析的人力门槛大幅降低**— 过去需要资深安全研究员数天完成的分析工作，AI 辅助下可在数小时内完成
- **攻击的规模化成为可能**— AI 支持批量化、自动化地分析受保护二进制文件的关键逻辑与代码漏洞
- **专用 AI 逆向工具快速涌现**— LLM4Decompile [2]（专用反编译大模型）、DecompAI [5]（自主逆向 Agent）、ReverserAI [6] 等开源工具降低了攻击工具链的获取成本

这不再是假设性威胁，而是正在发生的现实。

适用范围

本文主要讨论二进制软件保护场景，包括桌面端应用、移动端应用及其他包含本地核心逻辑的软件。

核心发现

本文基于对当前 AI 辅助逆向工程能力的系统性研究，并结合深盾软安实验室针对 VirboxProtector 防护体系的实测验证，得出以下核心发现：

发现一：基础混淆对 AI 的防护力不足

公开研究显示（见附录 D [1]），仅依赖基于 LLVM 的开源混淆方案（如 Instruction Substitution、Control Flow Flattening、Bogus Control Flow 等），顶级 AI 模型仍可实现 20%-36% 的代码还原成功率。使用单一混淆技术时，部分场景下 AI 成功率甚至超过 50%。**这意味着仅依赖传统混淆已不足以保护 License 校验、核心算法和关键业务逻辑。**

发现二：代码虚拟化是对抗 AI 语义还原的关键防线

代码虚拟化通过将原生指令转换为自定义虚拟机字节码（Custom VM Bytecode），深度改变了 AI 的分析对象 — 自定义指令集极少存在于 LLM 的公开训练数据中，使 AI 难以依赖公开训练数据直接完成语义还原。**对于核心资产，代码虚拟化应作为首选防护手段，而非可选项。**

发现三：反编译器输出质量直接决定 AI 攻击效果

公开研究显示（见附录 D [1]），AI 从反编译伪代码（Pseudocode）输入的成功率比从原始汇编（Raw Assembly）输入高出 2-5 倍。这意味着**降低反编译器输出质量本身**就是一条高价值防线。**未开启导入表保护和字符串加密，会保留大量可被 AI 利用的语义锚点。**

发现四：仅有静态防护不够，运行时防护是必要补充

AI Agent 已具备通过 MCP 协议实时操控调试器（如 x64dbg、GDB）进行动态分析的能力。其核心优势在于"静态理解 + 动态验证"的闭环。**仅有静态防护无法阻断这个闭环，必须配合 RASP 运行时防护。**

结论

VirboxProtector 采用完全自研的保护引擎，构建了四层纵深防护架构：

•**代码虚拟化**— 自定义 VM 指令集，使 AI 难以依赖公开训练数据完成语义还原

•**高级代码混淆**— 自动集成垃圾代码、立即数加密、间接调用、指令替换、不透明谓词、随机多分支等多种变换，整体生效，无需逐项配置

•**代码加密与反编译对抗**— 代码加密（运行时按需解密）、导入表保护、符号表隐藏、字符串加密，系统性降低反编译输出质量

•**RASP 运行时防护**— 反调试、内存保护、完整性校验、Hook 检测，阻断 AI Agent 的动态验证闭环

该体系相较于基于 LLVM 的开源混淆方案，在防护深度和技术维度上均有本质性的提升。在 AI 能力持续进化的背景下，防护的关键已从"是否启用了保护"转变为"是否启用了足够深度的防护体系"。

对开发者的行动建议

紧急程度	行动建议
🔴 **立即**	对核心算法、License 校验逻辑启用**代码虚拟化**（而非仅使用混淆）
🔴 **立即**	启用**字符串加密**和**导入表保护**，降低反编译器输出可读性
🟡 **短期**	部署 RASP 反调试与完整性校验，阻断 AI Agent 的动态验证闭环

🟡 **短期**	按函数重要性分级施加防护 — 核心函数用**虚拟化**，重要函数用**混淆**，一般函数用**代码加密**，配合全局字符串加密与导入表保护形成纵深
🟢 **持续**	定期评估防护配置是否匹配最新 AI 模型能力的演进

> ***核心信息：AI 大幅降低了逆向分析的门槛，但以代码虚拟化为核心的多层纵深防护体系仍然有效。关键不在于"是否使用了防护"，而在于"是否为不同函数启用了匹配其重要性的防护级别"。***

二、AI 辅助逆向工程的威胁与攻击手段

2.1 AI 逆向能力现状与影响

过去，逆向分析高度依赖专业人才 — 攻击者需要精通汇编语言、熟悉操作系统内核机制、掌握 IDA Pro 等工具，并具备数年实战经验。这种高门槛为软件提供了一层天然屏障。2025 年以来，大语言模型（LLM）的介入正在瓦解这层屏障。安全研究机构已在实际场景中使用 AI 完成对混淆恶意软件的逆向分析 [3]，AI 辅助逆向不再是理论探讨。

通用大模型已具备逆向辅助能力

当前主流通用大模型 — OpenAI GPT 系列、Anthropic Claude、DeepSeek、Google Gemini — 由于训练数据中包含了大量开源代码、安全研究文献和反编译输出样本，已具备相当强的代码分析能力：

能力维度	说明
反编译代码理解	阅读 IDA Pro / Ghidra 生成的伪代码，理解其逻辑含义
变量与函数语义恢复	将 `sub_401230`、`v12` 等无意义标识符重命名为 `check_license`、`key_buffer` 等语义化名称
算法模式识别	识别代码中的加密算法（AES、RSA）、哈希函数、校验逻辑等已知模式
漏洞与弱点定位	分析代码中的安全漏洞、逻辑缺陷和可利用路径

行业测试数据显示（见附录 D [1]），**在未保护的代码上**，顶级 AI 模型的代码还原成功率可达 **72%-86%。**

对软件保护的冲击

维度	影响
门槛下降	过去需要数年逆向经验，现在会写 Prompt 的初级攻击者即可借助 AI 完成大部分分析
效率跃升	人工分析需要数小时的函数，AI 在几十秒内即可给出分析结果
规模化攻击	AI 使批量化、自动化分析大量受保护二进制成为可能，攻击者可同时扫描多个目标

更关键的是，AI 模型的能力仍在快速增长，每一代显著优于前一代。今天能够抵挡 AI 的防护策略，不一定能抵挡明年的模型。

但也应客观认识 AI 的当前边界：AI 并不意味着能够自动攻破所有软件保护。其分析效果高度依赖输入质量（反编译输出的可读性）、上下文窗口容量、以及训练数据中是否包含目标架构的相关知识。当这些条件被系统性削弱时，AI 的自动化分析能力会显著下降。

2.2 AI 自动化逆向的五个核心依赖

AI 自动化逆向的有效性建立在五个明确的技术依赖之上。理解这些依赖，是制定有效防护策略的前提。

一次典型的 AI 辅助逆向攻击遵循以下链路：

目标二进制 → 反编译工具提取伪代码/调用图 → LLM 语义分析与推断 → 调试器动态验证 → 定位关键逻辑 → 攻击利用（Keygen / Patch / 算法复制）

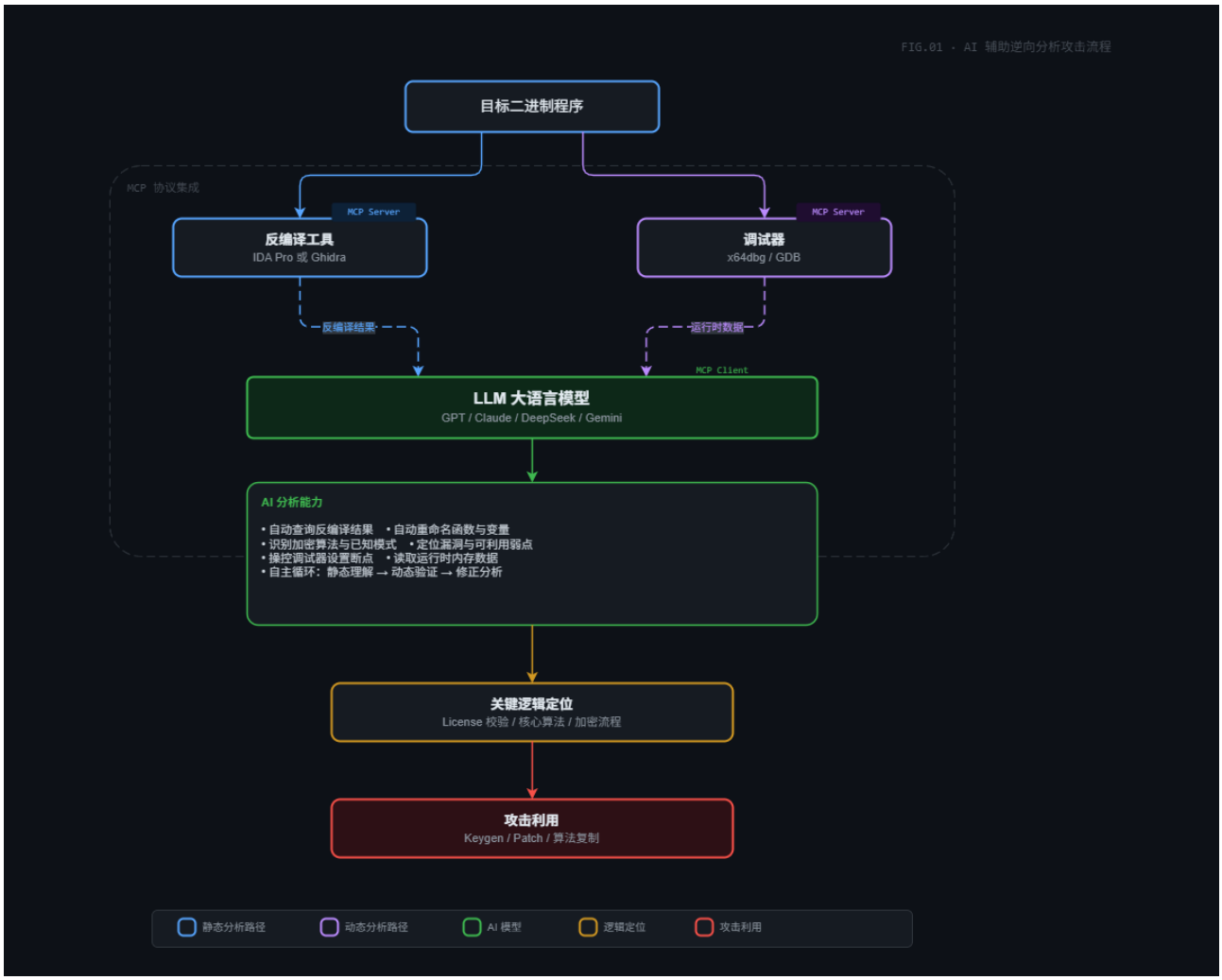


fig01-AI 攻击流程图

在这条链路中，攻击者通过 MCP（Model Context Protocol）协议将 LLM 与反编译工具、调试器直接连接，形成自动化分析流水线。已有成熟的开源方案（如 IDA / Ghidra MCP Server [7]）可直接部署使用，工具获取门槛极低。

在这套工具链中，AI 的分析效果取决于以下五个条件是否被满足：

依赖一：高质量的反编译输入

AI 的分析起点是反编译工具（IDA Pro / Ghidra）生成的伪代码、调用图和数据引用。行业数据显示（见附录 D [1]），AI 从伪代码输入的成功率比从原始汇编高出 2-5 倍。伪代码的可读性和完整性直接决定了 AI 分析的上限。

依赖二：语义线索

函数名、API 名称（如 `CreateFile`、`RegOpenKey`）、字符串常量（如`"Invalid license"`、`"AES-256"`）、错误信息等是 AI 理解代码功能的关键锚点。这些线索让 AI 能快速定位关键逻辑，而无需逐行分析全部代码。

依赖三：稳定的控制流与数据流

AI 需要清晰的控制流图（CFG）和数据流关系来理解程序逻辑。线性、可预测的代码结构让 AI 能够高效提取和还原业务逻辑。

依赖四：动态调试验证

AI Agent 通过 MCP 协议操控调试器（x64dbg / GDB），设置断点、读取运行时内存、跟踪执行流程，动态验证静态分析的推断。这种"静态理解 + 动态验证"的闭环大幅提高了分析的置信度和准确性。

依赖五：可复用的攻击经验

AI 的分析能力建立在训练数据中的已有知识之上 — 对标准指令集（x86 / ARM）的语义理解、对开源混淆方案（如 OLLVM）变换规则的模式识别。这种经验复用使 AI 能将已有知识迁移到新目标，显著降低每次分析的成本。

> ***这五个依赖同时也是五个可被攻击的支点。****VirboxProtector 的防护体系正是围绕系统性削弱和阻断这些依赖而设计的（详见第三章）。*

三、VirboxProtector 多层纵深防护架构

3.1 防护设计理念与架构总览

纵深防御（Defense in Depth）

VirboxProtector 的设计理念是**纵深防御**：多层异构防护技术叠加，每一层针对不同攻击阶段，形成互补而非重复的保护效果。即使某一层被攻破，攻击者仍面临下一层完全不同的技术障碍。

完全自研的保护引擎

VirboxProtector 采用完全自研的保护引擎，独立于开源生态。在对抗 AI 逆向分析时，这具有关键意义：

- **变换规则不公开**— AI 大模型的训练数据中包含了大量开源混淆方案（如基于 LLVM 的方案）的源码和技术文档，AI 能从中"学到"变换模式并反向还原。自研引擎的变换规则不在公开数据中，极大增加了 AI 预先学习的难度
- **独立于开源生态演进**— 不受开源方案的已知漏洞和公开攻击研究的影响

防护架构总览

fig02-四层防护架构图

VirboxProtector 构建了四层纵深防护体系，各层从不同维度对抗 AI 分析：

防护层	核心机制	对 AI 的作用
代码虚拟化	原生指令 → 自定义 VM 字节码，虚拟机自身带有混淆自保护	**破坏语义**— 显著打断 AI 的语义恢复路径
高级代码混淆	垃圾代码 / 立即数加密 / 间接调用 / 指令替换 / 不透明谓词 / 随机多分支	**污染语义**— AI 理解到的语义是错误的
代码加密与反编译对抗	代码加密 / 压缩 / 导入表保护 / 字符串加密 / 符号表隐藏	**消除语义**— 非运行时 AI 读不到代码
运行时防护 (RASP)	反调试 / 内存保护 / 完整性校验 / Hook 检测	**阻断动态分析**— AI Agent 难以稳定完成调试器驱动的动态验证

如第二章所述，AI 自动化逆向依赖五个核心条件。VirboxProtector 的四层防护体系逐一削弱和阻断这些依赖：

AI 自动化逆向依赖	VirboxProtector 对抗能力	效果
高质量反编译输入	代码虚拟化、代码加密、反编译对抗	反编译工具无法生成可读伪代码
语义线索（字符串/API/符号）	字符串加密、导入表保护、符号表隐藏	消除 AI 的语义锚点
稳定的控制流与数据流	高级混淆（不透明谓词、随机多分支、间接调用）	AI 自动化分析置信度大幅下降
动态调试验证	RASP（反调试、内存保护、Hook 检测）	阻断 Agent 的"静态+动态"验证闭环
可复用的攻击经验	自研 VM 引擎、多态化保护	单次分析结果无法复用到其他目标

3.2 代码虚拟化（Code Virtualization）

代码虚拟化是 VirboxProtector 防护体系的核心技术支柱，也是当前对抗 AI 逆向分析最有效的防护手段。

技术原理

代码虚拟化将受保护函数的原生 CPU 指令（x86、ARM 等）转换为**自定义虚拟机字节码**，并内嵌一个专用的虚拟机解释器（VM Interpreter）来执行这些字节码。

fig03-二进制虚拟化保护原理图

程序内部相当于运行着一个**未公开的 CPU 执行器**，对虚拟机字节码数据流进行循环读取-解释-执行。每条虚拟指令的格式和意义均未公开，**使 AI 的自动化语义还原结果不可直接利用。**同时，虚拟机自身也带有混淆自保护，显著增加了反编译工具分析虚拟机代码的难度，连解释器的执行逻辑都难以被清晰还原。

为什么代码虚拟化是对抗 AI 语义还原的关键防线

1. 自定义指令集超出 AI 的知识边界

当前所有 LLM 的训练数据都基于公开的代码和技术文献。x86、ARM 等标准指令集的语义和模式已被 AI 充分学习。但 VirboxProtector 的自定义 VM 指令集是**私有的、不公开的**，极少出现在公开训练数据中。AI 面对 VM 字节码时，相当于面对一种从未见过的编程语言，且缺乏文档参考 — 这显著削弱了 AI 进行语义理解的基础。

2. 反编译输出被深度改变

AI 的逆向分析高度依赖反编译工具生成的伪代码。虚拟化后，VM 字节码作为数据而非指令存在，破坏了反编译工具的基本假设。AI 即使能"读懂"解释器本身的代码，也难以推断出被解释的字节码代表什么业务逻辑。加之 handler 映射关系不公开，dispatch loop 与真实业务逻辑完全解耦，使自动化分析结果不可直接利用。

3. 多态性使攻击不可复用

VirboxProtector 的虚拟化引擎支持生成不同的 VM 架构变体。即使攻击者成功分析了一个版本的 VM 指令集映射关系，这一成果也**无法直接复用**于其他受保护的程序或后续版本。每次分析都是全新的工作量。

3.3 高级代码混淆

高级代码混淆从两个层面对抗逆向分析：第一，混淆后的代码本身就大幅增加了反编译工具反汇编的难度；第二，即使 AI 直接读取混淆后的指令进行分析，得到的语义也是**被污染的、错误的**，AI 需要进一步识别并对抗每一种混淆策略才有可能还原真实逻辑。

技术实现

VirboxProtector 的混淆引擎集成了多种变换技术：

变换技术	作用
垃圾代码注入	插入大量功能无关但语法合法的代码，膨胀代码体积
立即数加密	将常量替换为运行时计算表达式，隐藏原始数值
间接调用	直接函数调用转换为函数指针或跳转表的间接调用，切断静态调用关系
指令替换	标准指令替换为功能等价但形态不同的指令组合
不透明谓词	插入分析工具无法判定的条件分支，制造虚假控制流路径
随机多分支	线性代码转换为多路分支结构，打散原始执行流

对 AI 的阻碍

•**消耗 Token 预算**— 混淆后代码体积膨胀数倍乃至数十倍，大量垃圾代码和虚假分支占据 AI 上下文窗口，挤压真实逻辑的分析空间

•**误导 AI 判断**— 不透明谓词和随机多分支制造大量"看起来会执行"但实际不可达的路径。AI 不是不能读混淆后的代码，而是**自动化分析的置信度大幅下降**— 无法区分真假路径，输出结果的可靠性不足以支撑后续利用

•**自研方案不可预测**— 开源方案（如 OLLVM）的变换规则公开，AI 可从训练数据中学习其模式。VirboxProtector 完全自研，显著增加了 AI 通过模式匹配"识破"混淆的难度

3.4 代码加密与反编译对抗

这一层的核心目标是：**使 AI 在静态分析阶段缺少可直接理解的代码输入。**

代码加密

代码加密将受保护函数的代码块视为数据进行加密存储。程序运行到该函数时，才在内存中解密并执行。

•**对静态分析的影响**：反编译工具加载二进制文件时，看到的是加密后的数据段而非可执行代码，无法生成有意义的伪代码

•**对动态分析的影响**：只有运行时才解密，配合内存保护可防止内存 dump

反编译对抗

AI 从伪代码输入的分析成功率比从原始汇编高出 2-5 倍，因此**降低反编译工具的输出质量**本身就是高价值防线：

技术	效果
导入表保护	隐藏 API 调用引用（如 `CreateFile`、`RegOpenKey`），切断 AI 通过 API 名推断功能的路径
符号表隐藏	删除函数名、变量名等调试信息，AI 只能面对 `sub_xxxx`、`var_xx` 等无意义名称
字符串加密	加密所有字符串常量（如`"Invalid license"`、`"AES-256"`），消除 AI 定位关键代码的线索
智能压缩	对受保护区域进行压缩，增加额外解析层

组合效果：反编译输出中缺乏 API 名称、字符串线索和符号信息 — AI 的分析输入被从源头消除。

3.5 运行时防护（RASP）

前三层防护主要针对静态分析。运行时防护覆盖的是**动态分析维度**的威胁。AI Agent 的核心优势在于"静态理解 + 动态验证"的闭环 — 先通过反编译形成推断，再通过调试器验证。RASP 的目标是**阻断这个验证闭环**，使 Agent 的静态推断无法得到动态确认：

防护能力	技术手段	对 AI 动态分析的影响
反调试	检测调试器附加（内核级和用户级）	剥夺 AI Agent 的动态观测能力
内存保护	Anti Memory Dump、内存访问检测	阻止通过内存 dump 获取解密后的代码

完整性校验	运行时持续校验代码段完整性	阻止基于 AI 分析结果修改二进制
Hook 检测	识别 Inline Hook、IAT Hook 及 Frida 等插桩框架	阻止通过 Hook 拦截关键函数调用

定位说明：*RASP 不是替代虚拟化和混淆，而是覆盖静态防护无法解决的动态验证阶段。*

四层协同：*虚拟化破坏语义 → 混淆污染语义 → 加密消除语义 → RASP 阻断动态分析。四层作用于不同维度，形成不依赖单点能力的纵深防护体系，显著提高自动化逆向的失败率和攻击成本。*

四、防护有效性验证

4.1 行业基线：开源混淆方案的防护上限

在评估 VirboxProtector 的防护效果之前，首先需要了解行业基线 — 基于 LLVM 的开源混淆方案（OLLVM）在面对 AI 时的实际防护效果。以下数据来自 Promon Security Research 团队的公开研究（见附录 D [1][4]），覆盖 10 个主流 AI 模型、2000 个混淆二进制样本，并对 8 个商业模型进行了深度定性评测。

测试方法

项目	说明
测试规模	200 个 C 程序 × 5 种 OLLVM 混淆配置（SUB / FLA / BCF / FLA+BCF / SUB+FLA+BCF）
AI 模型	10 个主流模型，覆盖 Anthropic、OpenAI、Google、DeepSeek
验证方式	两阶段 — ① GCC 编译通过 ② 执行输出与原始程序完全一致

核心数据

AI 代码还原成功率：

场景	顶级模型成功率
未保护代码	**72%-86%**
三重混淆叠加（SUB+FLA+BCF）	**20%-36%**
伪代码输入 vs 汇编输入	伪代码成功率高出 **2-5 倍**

各模型对不同混淆技术的解混淆能力（0 = AI 自主完成，5 = 完全失败）：

模型	BCF（虚假控制流）	IS（指令替换）	CFF（控制流平坦化）	三重组合
Claude 3.7 Sonnet	0	1-2	0-1	**5**
Grok 3	1	4	0	**5**
GPT-4.5	1-2	4	0	**5**
GPT-o1 Pro	3	—	0	**5**
DeepSeek R1	5	5	1-2	**5**

0 = AI 自主完成，5 = 完全失败，— = 不适用。数据来源：附录 D [1][4]。

关键发现

1.**单一混淆不足以对抗 AI**— 多个顶级模型对 BCF 和 CFF 可自主完成解混淆（Level 0-1）

2.**三重组合使所有模型完全失败**— 所有被测模型在三重组合下均为 Level 5，混淆层的效果是乘法而非加法

3.**伪代码输入显著提升 AI 成功率**— 降低反编译输出质量本身就是一条高价值防线

4.**但大规模测试中顶级模型仍有 20%-36% 的成功率**— 对于高价值目标，仅依赖 OLLVM 的防护仍然不够

结论：开源混淆方案的组合使用可以显著增加 AI 分析难度，但在面对顶级模型时仍存在被还原的风险。有效防护需要超越开源混淆，引入代码虚拟化、代码加密和运行时防护等多层纵深手段。

4.2 VirboxProtector 防护效果实测

为验证 VirboxProtector 各防护层级对 AI 逆向分析的实际阻断效果，我们设计了一组针对性测试，分别对同一测试样本施加不同防护配置，观察 AI 在各配置下的分析表现。

测试方法论

项目	说明
测试执行	深盾软安实验室
测试样本	C/C++ 综合样本，涵盖 AES-128 加密算法、License 校验逻辑（自定义 hash + AES 校验）、网络通信（TCP 心跳上报）
测试模型	GPT-5.5（OpenAI 最新商业模型，具备 MCP 协议驱动的 Ghidra 交互能力）
反编译工具	Ghidra 12.0
评估维度	代码还原成功率、语义理解程度、攻击耗时
验证方式	与 4.1 一致：① 编译通过 ② 执行输出与原始程序完全一致（功能等价）

> ***说明：****测试使用 AI Agent 模式，AI 通过 MCP 协议直接操控 Ghidra 进行反编译分析、变量重命名、类型修正等操作，模拟真实攻击者的自动化逆向流程。*

AI 分析提示词

本次测试使用的提示词要求 AI 对目标函数进行全面分析还原，具体包括：

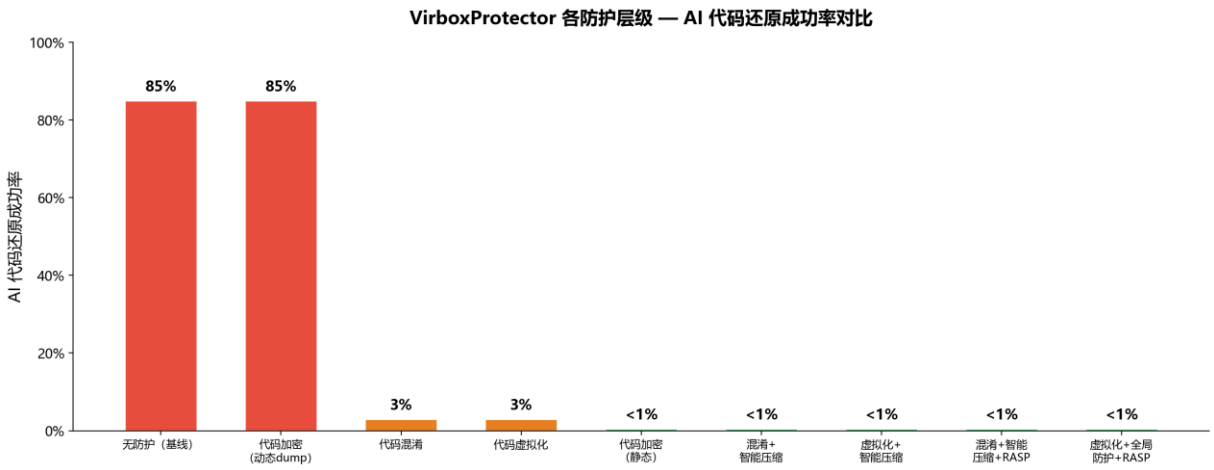
```
你的任务是全面分析并还原给定地址的函数。你需要理解函数的输入输出与功能逻辑，
并确保分析结果能够满足我们的需求。

使用以下系统方法：
1. 反编译分析 – 仔细检查反编译器的输出，添加详细注释
2. 提高数据库可读性 – 将变量重命名为描述性名称，修正类型
```

3. 必要时进行深入分析 – 检查反汇编代码，使用子代理执行详细分析
4. 重要限制 – 所有结论均应基于实际数据分析，而非假设

各防护层级对比

防护配置	AI 代码还原成功率	说明
无防护（基线）	85%	AI 可直接从高质量伪代码还原大部分逻辑
代码加密	<1%/85%	纯静态分析下 AI 无法获取可读代码（<1%）；但若攻击者通过动态 dump 获取解密后代码，则退化为无防护状态（85%）
代码混淆	≈3%	自研变换规则不在 AI 训练数据中，多种混淆技术自动叠加，AI 几乎无法还原
代码虚拟化	≈3%	自定义 VM 字节码超出 AI 知识边界，反编译输出丧失原始语义
混淆 + 智能压缩	<1%	混淆阻断语义还原，智能压缩阻断静态分析输入，本测试中未观察到成功还原
虚拟化 + 智能压缩	<1%	叠加智能压缩后，AI 无法获取可分析的代码，静态分析路径被有效阻断
混淆 + 智能压缩 + RASP	<1%	在混淆 + 智能压缩基础上阻断动态分析路径，AI Agent 未能完成验证闭环
虚拟化 + 全局防护 + RASP	<1%	静态与动态分析路径同时被阻断，AI Agent 未能完成"静态+动态"验证闭环



各防护层级 AI 代码还原成功率对比

> ***代码加密的双重成功率说明：****代码加密的防护效果取决于攻击路径。在纯静态分析场景下，代码以密文形式存储，反编译工具无法生成伪代码，本测试中 AI 未能有效还原（<1%）。但若攻击者具备动态分析能力（如内存 dump），可在运行时获取解密后的代码，此时防护退化为无防护状态。这正是代码加密需要与 RASP（反调试、内存保护）配合使用的原因。*

无防护效果展示

以下为未经任何保护的测试样本在 Ghidra 中的反编译输出。可以看到，函数名称（`verify_license`、`send_heartbeat`）、API 调用（`socket`、`connect`、`send`）、字符串常量（`"VBP-"`）、算法特征（AES S-Box）等语义信息均可直接读取：

```
/* WARNING: Function: __security_check_cookie replaced with injection: security_check_cookie */

bool FUN_140001680(byte *param_1)

{
  longlong lVar1;
  undefined1 auStack_58 [32];
  ulonglong local_38 [2];
  byte local_28;
  byte local_27;
  int local_18 [2];
  ulonglong local_10;

  local_10 = DAT_140023000 ^ (ulonglong)auStack_58;
  lVar1 = -1;
  do {
    lVar1 = lVar1 + 1;
  } while (param_1[lVar1] != 0);
  if ((lVar1 == 0x13) && (*(int *)param_1 == 0x2d444544)) {
    local_38[1] = 0;
    local_38[0] = (ulonglong)
                  ((((((((((((((*param_1 ^ 0x811c9dc5) * 0x1000193 ^ (uint)param_1[1]) * 0x1000193
                               ^ (uint)param_1[2]) * 0x1000193 ^ (uint)param_1[3]) * 0x1000193 ^
                           (uint)param_1[4]) * 0x1000193 ^ (uint)param_1[5]) * 0x1000193 ^
                       (uint)param_1[6]) * 0x1000193 ^ (uint)param_1[7]) * 0x1000193 ^
                   (uint)param_1[8]) * 0x1000193 ^ (uint)param_1[9]) * 0x1000193 ^
               (uint)param_1[10]) * 0x1000193 ^ (uint)param_1[0xb]) * 0x1000193 ^
           (uint)param_1[0xc]) * 0x1000193 ^ (uint)param_1[0xd]) * 0x1000193);
    FUN_140001150((uint *)local_38,(uint *)&local_28);
    FUN_140001120((undefined1 *)local_18,"%02X%02X",(ulonglong)local_28,(ulonglong)local_27);
    return local_18[0] == *(int *)(param_1 + 0xf);
  }
  return false;
}

```

无防护 — Ghidra 反编译输出

AI 分析结果：

GPT-5.5 通过 MCP 协议驱动 Ghidra，在数分钟内完成了对所有函数的自动化分析。AI 准确识别出 AES-128 加密算法（通过 S-Box 常量表特征匹配），正确还原了 License 校验的格式解析、hash 计算、AES 加密校验等完整流程，并为所有变量和函数添加了语义化命名。

```
bool validate_ded_token(byte *token)

{
  longlong token_len;
  undefinedl auStack_58 [32];
  ulonglong hash_block [2];
  byte digest_byte0;
  byte digest_byte1;
  int expected_hex_chars [2];
  ulonglong local_10;

  local_10 = DAT_140023000 ^ (ulonglong)auStack_58;
  token_len = -1;
  do {
    token_len = token_len + 1;
  } while (token[token_len] != 0);
                    /* Reject unless token length is exactly 0x13 bytes before NUL. */
                    /* Required little-endian prefix bytes are 44 45 44 2d, i.e. ASCII "DED-". */
  if ((token_len == 0x13) && (*(int *)token == 0x2d444544)) {
                    /* Computes a 32-bit FNV-1a style chain over token bytes 0..0x0d using offset
                       0x811c9dc5 and prime 0x01000193. */
    hash_block[1] = 0;
    hash_block[0] =
         (ulonglong)
         ((((((((((((*token ^ 0x811c9dc5) * 0x1000193 ^ (uint)token[1]) * 0x1000193 ^
                     (uint)token[2]) * 0x1000193 ^ (uint)token[3]) * 0x1000193 ^ (uint)token[4]) *
                0x1000193 ^ (uint)token[5]) * 0x1000193 ^ (uint)token[6]) * 0x1000193 ^
              (uint)token[7]) * 0x1000193 ^ (uint)token[8]) * 0x1000193 ^ (uint)token[9]) *
            0x1000193 ^ (uint)token[10]) * 0x1000193 ^ (uint)token[0xb]) * 0x1000193 ^
          (uint)token[0xc]) * 0x1000193 ^ (uint)token[0xd]) * 0x1000193);
                    /* Encrypts the 16-byte block whose first dword is the hash result and remaining
                       dwords are zero, using the fixed AES-128 key embedded at 0x1400206a0. */
    aes128_encrypt_block_fixed_key((uint *)hash_block,(uint *)&digest_byte0);
                    /* Formats the first two ciphertext bytes as four uppercase hex characters via
                       "%02X%02X". */
    format_string_wrapper
              ((char *)expected_hex_chars,"%02X%02X",(ulonglong)digest_byte0,(ulonglong)digest_byte1
              );
                    /* Validation succeeds when those four generated ASCII hex characters equal
                       token bytes at offset 0x0f..0x12. Byte 0x0e is not read by this comparison.
                        */
    return expected_hex_chars[0] == *(int *)(token + 0xf);
  }
  return false;
}
```

AI 分析结果 — 算法识别与变量重命名

校验逻辑如下：

```c
bool validate_ded_token(byte *token) {
    if (strlen(token) != 0x13)
        return false;

    if (*(uint32_t *)token != 0x2d444544)
        return false;   // bytes: 44 45 44 2d, ASCII "DED-"

    hash = fnv1a_style(token[0..0x0d]);

    block[0] = hash;
    block[1] = 0;
    block[2] = 0;
    block[3] = 0;

    ciphertext = AES128_encrypt_fixed_key(block);

    expected = sprintf("%02X%02X", ciphertext[0], ciphertext[1]);

    return memcmp(expected, token + 0x0f, 4) == 0;
}
```

AI 分析结果 — 函数逻辑还原

AI 分析结论：GPT-5.5 快速识别出 AES-128 加密算法，并合理还原了函数逻辑。代码还原率约 **85%。**AI 能够成功的核心原因在于：反编译输出中保留了完整的语义锚点 — 函数名、API 名、字符串常量、算法特征常量，这些信息为 AI 提供了充足的推理依据。

代码虚拟化效果展示

对同一测试样本的关键函数施加代码虚拟化保护后，Ghidra 的反编译输出发生了根本性变化。原始的业务逻辑（License 校验、AES 加密等）被转换为自定义 VM 字节码，反编译工具只能呈现 VM 解释器的 dispatcher 循环：

```
1
2 /* WARNING: Control flow encountered bad instruction data */
3
4 void thunk_FUN_140042c48(void)
5
6 {
7   char *pcVar1;
8
9   pcVar1 = (char *)FUN_14003d020();
10  *pcVar1 = *pcVar1 + (char)pcVar1;
11  *pcVar1 = *pcVar1 + (char)pcVar1;
12  FUN_14004f448();
13  FUN_1400397d8();
14                     /* WARNING: Bad instruction - Truncating control flow here */
15  halt_baddata();
16 }
17
```

代码虚拟化 — Ghidra 反编译输出

AI 分析结果：

GPT-5.5 面对虚拟化后的代码，尝试进行了深入分析，但最终承认无法还原原始业务逻辑。AI 能够识别出这是一个"经过对抗/混淆处理的函数"，但几乎无法推断出具体的保护手段（未识别为虚拟化），更无法还原验证函数的实际逻辑。

```
/* WARNING: Control flow encountered bad instruction data */
/* Setting prototype: int verify_license_obfuscated_body(char *license) */

int verify_license_obfuscated_body(char *license)

{
  char *pcVar1;

                    /* Do not trust the linear pseudocode here. The first call enters a trampoline
                       that rewrites the return address; the apparent CMOV/ADD/CALL/bad-instruction
                       sequence is an obfuscation artifact. */
  pcVar1 = (char *)FUN_14003d020();
  *pcVar1 = *pcVar1 + (char)pcVar1;
  *pcVar1 = *pcVar1 + (char)pcVar1;
  FUN_14004f448();
  FUN_1400397d8();
                    /* WARNING: Bad instruction - Truncating control flow here */
  halt_baddata();
}
```

AI 分析结果 — 虚拟化代码分析（1）

对抗/混淆逻辑

`0x140042c48` 的反编译输出不能按线性伪代码理解。它先调用 `0x14003d020`，该函数通过 `LEA RSP`、`CALL`、`XCHG [RSP]`、`ADD [RSP]` 等操作修改栈上的返回地址，把控制流跳到隐藏 `.text` 代码块。

关键证据：

- `0x140042c48` 线性反编译出现 bad instruction
- `0x14003d020` 在 `0x14003d05c` 修改栈返回地址
- Ghidra xref 显示计算目标包括 `0x140066ed8`
- `0x140066ed8` 位于可执行 `.text: 140028000 - 1400739ff`
- 但当前未被 Ghidra 建成函数，说明该区域属于隐藏/非线性控制流块

这属于返回地址篡改式控制流混淆，不是普通函数调用。

AI 分析结果 — 虚拟化代码分析（2）

AI 分析结论：AI 可根据上下文推断这是一个许可验证相关的函数，但对虚拟化保护本身缺乏识别能力 — 既无法判断保护手段的类型，也无法穿透 VM 字节码还原底层逻辑。代码还原率约 **3%**。

这一结果验证了第三章的核心论断：VirboxProtector 的自定义 VM 指令集不在任何 LLM 训练数据中，AI 面对 VM 字节码相当于面对一种从未见过的"语言"，其语义还原能力被从根本上破坏。

代码混淆效果展示

对测试样本施加 VirboxProtector 代码混淆保护后，原始代码被自研混淆引擎自动施加了多种变换（垃圾代码注入、立即数加密、间接调用、指令替换、不透明谓词、随机多分支），代码体积和控制流复杂度大幅膨胀：

```
/* WARNING: Control flow encountered bad instruction data */
/* Setting prototype: int validate_license_key_obfuscated(char *license_key) */

int validate_license_key_obfuscated(char *license_key)

{
  stack_return_obfuscation_dispatcher();
                    /* WARNING: Bad instruction - Truncating control flow here */
  halt_baddata();
}
```

代码混淆 — Ghidra 反编译输出

AI 分析结论：代码还原率约 **3%**。AI 面对混淆后的代码，无法有效区分真实逻辑路径与虚假路径，分析输出的可靠性不足以支撑后续利用。VirboxProtector 的自研混淆引擎由于变换规则未公开，AI 无法从训练数据中学习其模式并反向还原，这是其相较于开源 OLLVM 方案的核心优势。

智能压缩效果展示

对测试样本施加智能压缩保护后，受保护程序的代码段以压缩数据形式存储，运行时才由 loader 解压并执行。Ghidra 无法直接获取原始代码：

程序功能总结

当前程序是一个 Windows x64 PE，导出入口只有 `entry @ 1400d9778` 。入口调用 `FUN_140157810` ，该函数在汇编层面通过栈返回地址变换实现跳板式控制流，Ghidra 反编译显示坏指令截断，说明程序存在打包/混淆式入口控制流。

可稳定反编译的主体代码大量匹配 Zstandard 压缩库逻辑，包括 Huffman/FSE 表读取、四路 Huffman 解码、LZ 匹配查找、哈希、序列统计和本地 `memcpy/memset` 。目前未从可见字符串和导入中发现普通业务 UI、命令行、网络协议文本、文件路径或 C2 配置。导入表极小，只有 `send` 和 `SetFilePointerEx` ，且未发现普通函数 xref。

我已在 Ghidra 数据库中对明确函数做了重命名和注释。

关键函数

`1400d9778` ， `entry` ：程序入口，调用入口跳板。

`140157810` ， `FUN_140157810` ：入口跳板/混淆控制流函数。

`14015ba08` ， `memcpy_byte_loop` ：本地字节复制函数。

`14015bae0` ， `memset_byte_loop` ：本地字节填充函数。

`1401359a8` ， `bitstream_init_reverse` ：反向 bitstream 初始化。

`140135b20` ， `bitstream_refill_reverse` ：反向 bitstream 重装载。

`140135fe8` ， `fse_decode_symbol_to_output` ：FSE/Huffman 表项解码并写入输出。

`1401369a0` ， `huf_decompress_4x_workspace_wrapper` ：四路 Huffman 解压包装入口。

`1401369d8` ， `huf_decompress_4x_dispatch` ：读取表后分派四路 Huffman 解压。

`140136ae8` ， `huf_decompress_4x1_using_dtable` ：四路 Huffman 解压实现之一。

`140137808` ， `huf_decompress_4x2_using_dtable` ：四路 Huffman 解压实现之二。

`1401387e8` ， `huf_read_dtable_from_header` ：从压缩头构建 Huffman 解码表。

`140157288` ， `huf_read_weights` ：读取 Huffman 权重。

`140156d08` ， `fse_decompress_huf_weights` ：用 FSE 解压 Huffman 权重。

`140156f78` ， `fse_read_normalized_counts` ：读取 FSE normalized counts。

`14014bdf8` ， `buffer_common_prefix_len` ：计算两个缓冲区公共前缀长度。

`14014bf20` ， `buffer_common_prefix_len_two_ranges` ：跨两段缓冲区计算公共前缀长度。

`14014d560` ， `hash_mix_and_shift` ：Zstd 匹配查找用乘法哈希。

`140143f40` ， `zstd_find_match_entry_checked` ：检查缓冲区边界后执行匹配查找。

`140144878` ， `zstd_count_2segments_ext_dict_match` ：带外部字典/双段缓冲区的匹配查找。

`140145148` ， `zstd_count_single_segment_match` ：单段缓冲区匹配查找。

智能压缩 — Ghidra 分析结果

AI 分析结论： AI 通过代码特征匹配识别到了 zstd 压缩库的存在，但无法在静态分析阶段解压获取被保护的程序代码，导致分析流程中断，无法进行下一步操作。这表明智能压缩有效地在反编译工具层面阻断了 AI 的输入来源。

核心结论

以下结论基于深盾软安实验室的实测验证（4.2 节），行业基线数据（4.1 节）来自 Promon Security Research 的公开研究。

1.**代码虚拟化是质变级防护**— 从无防护时的 85% 成功率骤降至虚拟化的约 3%，这不是量变而是质变。VM 字节码从根本上超出了当前 AI 的知识边界和推理能力，反编译工具无法生成有意义的伪代码，AI 的分析起点被严重破坏。

2.**多层叠加使还原成功率降至不可用水平**— 虚拟化 + 全局防护（智能压缩）+ RASP 的组合下，本测试中未观察到有效还原样本（<1%）。每一层的作用互补而非重复：虚拟化破坏语义、全局防护消除线索、RASP 阻断动态验证。本章以 GPT-5.5（当前最强商业模型之一）的分析过程进行展示，其余模型的测试结果同样验证了这一结论。

3.**自研引擎的不可预测性是持续优势**— VirboxProtector 的混淆和虚拟化变换规则均为完全自研、未公开，AI 无法从训练数据中学习对抗模式。随着 AI 模型能力的持续增长，这种"不在训练分布内"的特性提供了长期有效的防护基础。

五、防护配置最佳实践

5.1 防护策略选择原则

安全性与性能的平衡

VirboxProtector 的各项防护技术在安全强度和性能开销上存在差异：

防护技术	安全强度	性能影响	适用范围
代码加密	★★	小（性能友好）	广泛覆盖，函数运行时解密为明文，适合非核心但需要保护的代码
高级代码混淆	★★★	中等	重要函数，在安全性和性能之间取得平衡
代码虚拟化	★★★★★	较大（解释执行开销）	核心函数，提供最高安全强度

基本原则：分层施加，精准配置。

•代码加密作为基础层广泛覆盖，性能开销小，但安全性有限（函数执行时代码会解密为明文，存在被内存 dump 的风险）

•混淆用于重要函数，提供中等强度的防护

•虚拟化针对最关键的函数启用，以较高的性能代价换取最强的安全保障

如何判断哪些函数需要最高级别防护

优先保护以下类型的函数：

•**License / 授权校验逻辑**— 被破解的直接目标

•**核心算法实现**— 知识产权的核心载体

•**加密 / 解密流程**— 密钥处理和加解密操作

•**通信协议关键逻辑**— 服务器验证、数据签名

•**敏感业务判断**— 权限控制、功能开关、计费逻辑

5.2 典型场景配置方案

VirboxProtector 支持对原生二进制文件（PE .exe/.dll、ELF 可执行文件/.so、Mach-O 可执行文件/.dylib）进行保护。以下针对三类典型业务场景，给出推荐的防护配置。

场景一：核心算法与知识产权保护

适用对象：包含自研算法、数学模型、信号处理、AI 推理引擎等核心知识产权的原生模块（.dll / .so / .dylib）。

防护层	配置建议
核心算法函数	**代码虚拟化**— 最高强度保护，防止算法逻辑被 AI 还原
辅助计算函数	**代码混淆**— 在性能与安全间取得平衡
全局防护	启用**智能压缩**+**导入表保护**+**字符串加密**
运行时防护	启用**反调试**+**内存保护**，防止运行时 dump 获取解密后代码

> ***要点：****核心算法是竞争对手和攻击者的首要目标，必须使用代码虚拟化。算法函数通常调用频率有限，虚拟化的性能开销可控。*

场景二：License 校验与授权保护

适用对象：包含 License 校验、激活验证、功能授权判断逻辑的原生程序或动态库。

防护层	配置建议
License 校验函数	**代码虚拟化**— 这是攻击者的直接目标，必须最高强度
授权判断分支	**代码混淆**— 防止 AI 识别出判断逻辑并定位 patch 点
全局防护	启用**智能压缩**+**字符串加密**（消除`"license"`、`"expired"`等关键字符串线索）
运行时防护	启用**反调试**+**完整性校验**+**Hook 检测**，防止绕过校验或注入篡改

> ***要点：****License 校验是最常见的攻击目标。仅靠混淆保护校验函数，AI 仍有可能识别出校验逻辑的模式并生成 patch 方案。虚拟化 + RASP 的组合可同时阻断静态还原和动态篡改两条攻击路径。*

场景三：通信协议与数据安全

适用对象：包含私有通信协议实现、数据加解密、签名验证、安全通道建立等逻辑的原生模块。

防护层	配置建议

加解密与签名函数	**代码虚拟化**— 防止密钥处理流程和加密逻辑被还原
协议解析与组包函数	**代码混淆**— 隐藏协议格式和字段含义
全局防护	启用**智能压缩**+**导入表保护**（隐藏加密库 API 调用） +**字符串加密**
运行时防护	启用**反调试**+**内存保护**+**Hook 检测**，防止通过 Hook 拦截通信数据

> ***要点：****协议逆向的核心在于理解数据格式和加密方式。AI 可通过 API 名称（如 `SSL_read`、`encrypt`）和字符串常量快速定位关键代码，导入表保护和字符串加密在此场景中尤为关键。*

配置原则总结

无论哪种场景，核心原则一致：

1.**核心函数用虚拟化，重要函数用混淆**— 按函数的业务价值分级施加防护

2.**全局防护不可省略**— 智能压缩 + 字符串加密 + 导入表保护是消除 AI 语义锚点的基础层

3.**RASP 补齐动态防线**— 静态防护再强，也需要运行时防护阻断调试和内存 dump

4.**性能影响可控**— 虚拟化仅针对关键函数，不影响程序整体性能

5.3 常见误区

误区	事实
"开了混淆就够了"	行业数据表明，仅靠混淆 AI 成功率仍可达 20%-36%。混淆是基础层，不是终点
"虚拟化性能损耗太大，不敢用"	虚拟化应针对关键函数启用，而非全局施加。大多数 License 校验和核心算法函数的调用频率有限，性能影响可控
"只做静态防护就行"	AI Agent 已能通过 MCP 协议操控调试器进行动态分析，必须配合 RASP 运行时防护
"保护了主要函数就行"	字符串和 API 名称是 AI 定位关键代码的重要线索，不开启字符串加密和导入表保护相等于给 AI 自动逆向分析留下重要线索

六、未来展望

6.1 AI 逆向能力的演进趋势

AI 辅助逆向工程仍处于快速发展期。以下趋势将在未来 1-3 年内显著改变攻防格局：

通用模型能力持续跃升

每一代大模型在代码理解、逻辑推理和上下文处理能力上都大幅超越前代。上下文窗口从 32K 扩展到 128K 乃至更大，意味着 AI 可以一次性分析更大规模的代码。

专用逆向模型的威胁正在形成

随着更多安全研究数据被用于训练，专门针对逆向工程任务训练的模型（如 LLM4Decompile [2]）已经出现，未来的专用逆向模型将具备：

- 更强的汇编/伪代码理解能力
- 对常见保护模式的识别与绕过能力
- 针对特定保护方案的定向攻击能力

Agent 自动化程度持续加深

AI Agent 正在从"辅助分析"向"自主攻击"演进。未来的逆向 Agent 将具备更完善的工具链集成、更长的自主分析链路、以及多 Agent 协作分析的能力，进一步压缩人工介入的必要性。

6.2 VirboxProtector 的持续进化方向

面对 AI 逆向能力的持续增长，VirboxProtector 的防护体系同样在持续进化：

VM 指令集的动态更新与多态化

代码虚拟化的核心优势在于自定义指令集超出 AI 的知识边界。通过定期更新 VM 指令集架构、增强每次保护的多态性，确保即使 AI 模型迭代升级，也始终面对未知的指令体系。

对抗 AI 特征的针对性研究

深入研究 AI 模型在代码分析中的行为模式和依赖特征，开发针对性的防护技术 — 不仅让 AI "看不懂"，更让 AI "判断错"，主动利用 AI 的弱点增加其分析错误率。

新平台与新架构的持续覆盖

随着 RISC-V 等新架构的兴起以及更多应用场景的出现，VirboxProtector 将持续扩展平台和架构支持，确保防护能力覆盖客户的全部部署场景。

核心观点：AI 逆向能力的增长是确定性趋势，但防护技术同样在进化。关键在于选择具备持续演进能力的保护方案，而非依赖某一时刻的静态防护配置。软件保护不是一次性工作，而是伴随产品全生命周期的持续过程。

附录

A. 术语表

术语	全称	说明
LLM	Large Language Model	大语言模型，如 OpenAI GPT 系列、Anthropic Claude、DeepSeek、Google Gemini
MCP	Model Context Protocol	模型上下文协议，用于将 LLM 与外部工具（反编译器、调试器等）直接连接
Agent	AI Agent	AI 智能体，具备自主决策、调用工具、循环执行的 AI 系统
RASP	Runtime Application Self-Protection	运行时应用自保护，在程序运行期间检测并阻止攻击行为
VM	Virtual Machine	虚拟机，此处特指代码保护中用于执行自定义字节码的解释执行引擎
VM Bytecode	—	虚拟机字节码，代码虚拟化后生成的自定义指令序列
Pseudocode	—	伪代码，反编译工具将二进制代码还原为类 C 语言的高级表示
IDA Pro	Interactive Disassembler	业界主流的商业反编译与逆向分析工具
Ghidra	—	美国国家安全局（NSA）开源的逆向工程框架
Control Flow Flattening	—	控制流平坦化，一种将程序分支结构替换为集中调度器的混淆技术
Bogus Control Flow	—	虚假控制流，注入永远不会执行的虚假分支路径
Instruction Substitution	—	指令替换，将标准指令替换为功能等价的复杂序列

Opaque Predicate	—	不透明谓词，结果在编译时已确定但静态分析工具无法判定的条件表达式
OLLVM	Obfuscator-LLVM	基于 LLVM 编译器框架的开源代码混淆方案
Hook	—	钩子，通过修改函数入口或调用表拦截程序执行流的技术
Frida	—	开源的动态插桩工具，广泛用于移动端逆向分析
Memory Dump	—	内存转储，将程序运行时的内存内容导出为文件进行离线分析
IAT	Import Address Table	导入地址表，PE 文件中记录外部 API 调用地址的数据结构
Token	—	大模型处理文本的基本单位，一个 Token 约等于 3-4 个英文字符或 1-2 个中文字符
Context Window	—	上下文窗口，LLM 单次能处理的最大 Token 数量

B. 测试方法论详情

B.1 行业基线数据来源

来源一：大规模统计测试（参考文献 [1]）

项目	说明
数据来源	Promon Security Research,《App Threat Report 2026 Q1》
测试对象	OLLVM（SUB / FLA / BCF 三种变换，五种组合配置）
AI 模型	10 个主流模型，覆盖 Anthropic、OpenAI、Google、DeepSeek
测试样本	200 个 C 程序 × 2 架构 × 5 混淆配置 = 2000 个混淆二进制

反编译工具	IDA 9.2、Ghidra 11.2.1
验证方式	两阶段 — ① GCC 编译通过 ② 执行输出与原始程序完全一致

来源二：深度定性评测（参考文献 [4]）

项目	说明
数据来源	Tkachenko et al.,《Deconstructing Obfuscation》, arXiv:2505.19887
测试对象	OLLVM（BCF / IS / CFF 三种变换 + 三重组合）
AI 模型	8 个商业模型：GPT-4o、GPT-4.5、GPT-o1 Pro、GPT-3o Mini、Claude 3.7 Sonnet、DeepSeek R1、Grok 3、Grok 2
评估方式	0-5 级攻击者知识等级（0 = AI 自主完成，5 = 完全失败）

B.2 VirboxProtector 实测方法论

项目	说明
测试执行	深盾软安实验室
测试样本	C/C++ 综合样本，涵盖 AES-128 加密算法、License 校验逻辑（自定义 hash + AES 校验）、网络通信（TCP 心跳上报）
保护工具	VirboxProtector 3.5.5.22114
测试模型	GPT-5.5、Claude Sonnet 4.6
反编译工具	Ghidra 12.0
防护配置	8 个层级：无防护 → 代码加密 → 代码混淆 → 代码虚拟化 → 混淆 + 智能压缩 → 虚拟化 + 智能压缩 → 混淆 + 智能压缩 + RASP → 虚拟化 + 全局防护 + RASP
验证方式	两阶段 — ① GCC 编译通过 ② 执行输出与原始程序完全一致（功能等价）
评估指标	代码还原成功率、语义理解程度、攻击耗时

C. VirboxProtector 支持平台与格式

操作系统与架构

操作系统	支持架构	文件格式
Windows	x86, x86-64	PE（.exe, .dll）
Linux	x86, x86-64, ARM32, ARM64	ELF（可执行文件, .so）
macOS	x86-64, ARM64	Mach-O（可执行文件, .dylib）
Android	x86, x86-64, ARM32, ARM64	APK, DEX, .so
iOS	ARM64	Mach-O
ARM Linux	ARM32, ARM64	ELF（可执行文件, .so）

支持的语言与框架

类别	支持项
原生语言	C, C++, Delphi
.NET 生态	C#, VB.NET, .NET Framework, .NET Core
Java 生态	JAR, WAR, Class
脚本语言	Python, PHP
游戏引擎	Unity3D（Mono / IL2CPP）, Unreal Engine 4

D. 参考文献

[1] Tkachenko, A. (2026). *App Threat Report 2026 Q1: The State of Code Obfuscation Against AI*. Promon Security Research. https://promon.io/security-news/app-threat-report-2026-q1-the-state-of-code-obfuscation-against-ai

[2] Tan, H., Luo, Q., Li, J., & Zhang, Y. (2024). *LLM4Decompile: Decompiling Binary Code with Large Language Models*. EMNLP 2024. https://github.com/albertan017/LLM4Decompile

[3] Check Point Research. (2025). *Leveraging Generative AI to Reverse Engineer XLoader*. https://research.checkpoint.com/2025/generative-ai-for-reverse-engineering/

[4] Tkachenko, A., Suskevic, D., & Adolphi, B. (2025). *Deconstructing Obfuscation: A Four-Dimensional Framework for Evaluating Large Language Models Assembly Code Deobfuscation Capabilities*. arXiv:2505.19887. https://arxiv.org/abs/2505.19887

[5] DecompAI — 基于 LLM 的自主逆向分析 Agent，集成 Ghidra、GDB 和 Shell 工具链. https://github.com/Djirka/DecompAI

[6] ReverserAI — 本地离线运行的 AI 辅助逆向工程工具. https://github.com/mrphrazer/reverser_ai

[7] Ghidra MCP Server — 通过 MCP 协议将 Ghidra 反编译能力暴露给 LLM 的开源集成方案. https://github.com/LaurieWired/GhidraMCP