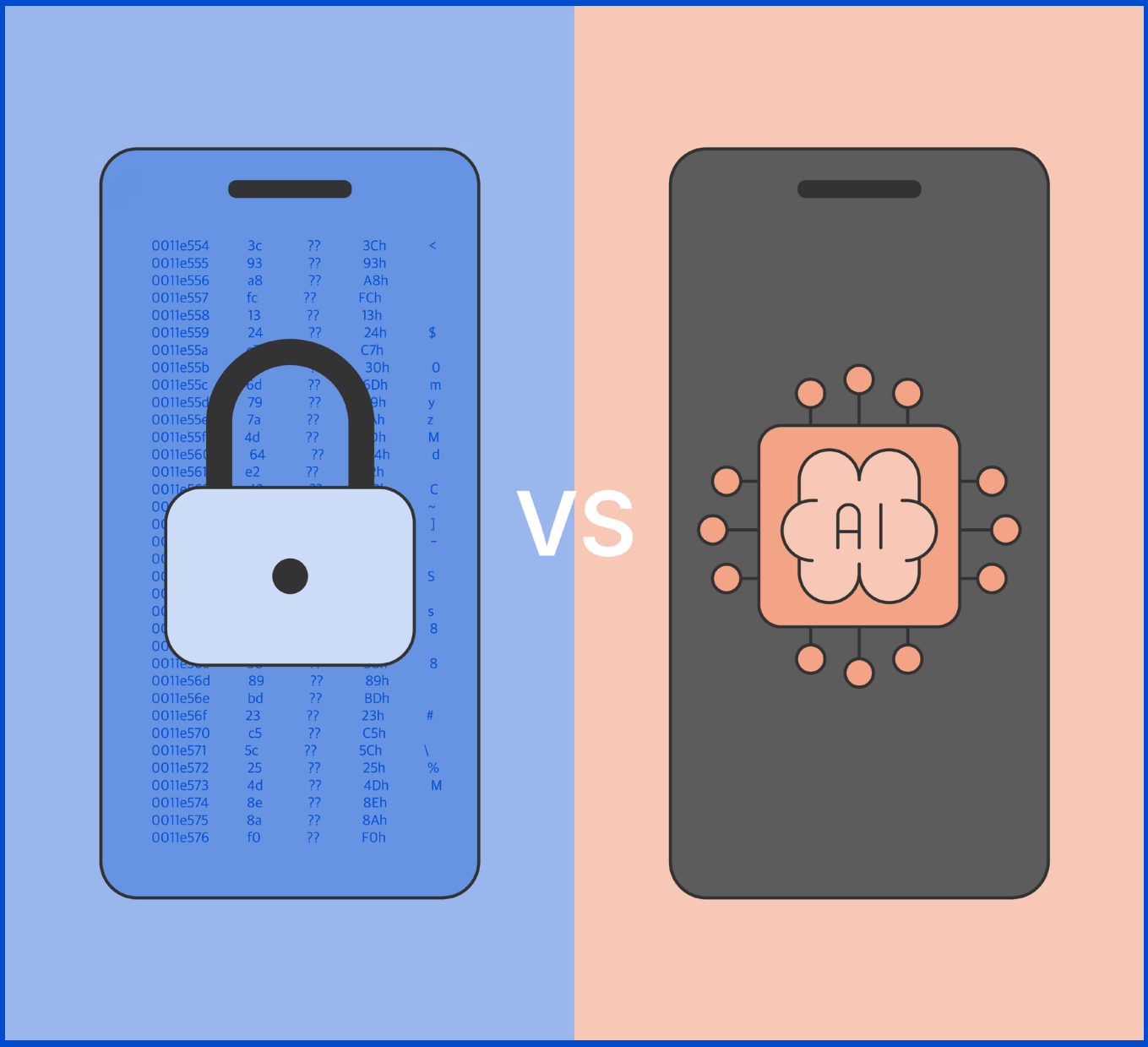
App Threat Report
Q1 2026

PROMON

The State of Code Obfuscation Against AI



Prepared by: Promon Security Research Team

Contents

03 Introduction

04 Code obfuscation and why it matters

06 Results
07 Key takeaways

12 Recommendations
13 1. Deploy layered obfuscation rather than relying on a single technique
14 2. Treat ARM and x86 targets differently
14 3. Recognize that AI tools are imperfect and design defenses to exploit this
14 4. Consider anti-decompilation as a strategic investment
14 5. Factor model capability into your threat model
15 6. Plan for capability evolution

16 Methodology
17 The Benchmark
17 Testing environment and process

19 Citations

Introduction

For years, code obfuscation has been one of the most reliable tools in the mobile application security toolkit. Obfuscation scrambles the internal structure of an app's code, making it difficult to read, understand, or copy. Its purpose is to protect intellectual property, frustrate would-be attackers, and help legitimate software evade detection by malware scanners. It's a foundational layer of defense for banks, fintech firms, enterprises, and any organization that distributes software into potentially hostile environments.

Artificial Intelligence, specifically large language models (LLMs) like GPT, Claude, Gemini, and DeepSeek, has become a capable assistant for software developers and, increasingly, for those who want to reverse engineer the software of others to breach sensitive data and commit fraud. These models can read disassembled code, analyze its logic, and attempt to reconstruct the original program. The question facing every security engineer today is whether obfuscation still holds up.

Promon's Security Research Team set out to answer that question empirically, testing ten of the world's most capable AI models against one of the most widely deployed commercial obfuscation frameworks: Obfuscator-LLVM (OLLVM). This testing took place across two processor architectures (ARM and x86) and thousands of code samples. The results contain both reassurance and warnings.

Code obfuscation and why it matters

When a developer writes an application, the source code is relatively readable. It contains logic, function names, comments, and structure that reflects the programmer's intent. Once compiled into a binary for distribution, much of that clarity is stripped away, but it can still be partially recovered using reverse engineering tools. Obfuscation takes the compiled binary and deliberately distorts it further, making reconstruction far more difficult.

OLLVM is a compiler-level obfuscation framework that applies three core transformation techniques:

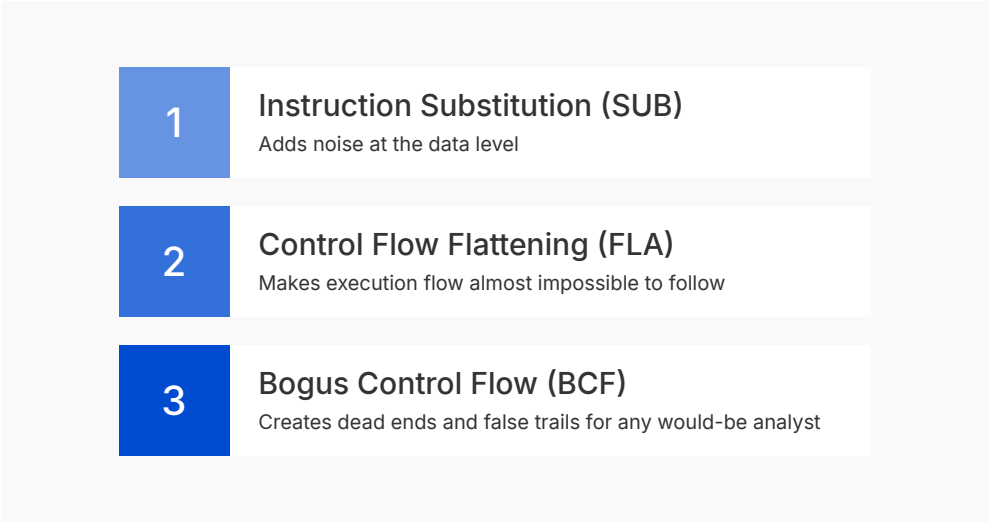
Instruction Substitution (SUB) replaces standard arithmetic operations with more complex but functionally identical sequences. Think of it as replacing the word "cat" with an obscure synonym that means the same thing but takes longer to parse.

Control Flow Flattening (FLA) dismantles the natural branching structure of a program, which means the if/then/else logic and loops. It then replaces it with a central traffic-controller that routes execution. The result is a program whose flow is almost impossible to follow visually.

Bogus Control Flow (BCF) inserts fake conditional branches and unreachable code paths into the program, creating dead ends and false trails that consume an attackers' time and attention.

These techniques can be applied individually or in combination, and OLLVM is used extensively in both legitimate commercial applications and, unfortunately, sophisticated malware. Understanding how well these protections hold against AI-powered reverse engineering has direct implications for anyone responsible for protecting deployed software.

OLLVM compiler-level obfuscation framework: three core transformation techniques



Results

Table 1: Results

Obfuscation level	ARM Pseudocode	ARM Assembly	x86 Assembly
No obfuscation (baseline)	~79.0%	~63.7%	~74.0%
Instruction Substitution only	77.5%	51.6%	67.9%
Bogus Control Flow only	73.6%	45.2%	60.0%
Control Flow Flattening only	55.0%	33.9%	52.3%
FLA + BCF	42.4%	21.0%	32.8%
Full three-pass (SUB + FLA + BCF)	26.1%	8.5%	20.6%

The table above summarizes average success rates across all ten models tested, showing how obfuscation complexity and input type affect AI deobfuscation performance.

These results show the average across all ten models tested. Top-performing individual models significantly exceed these averages, while the weakest performers fall far below. The baseline row shows performance on completely unobfuscated code. This is the attacker's starting point before any protection is applied.

Key takeaways

Our research delivered several significant findings that security and product teams should be aware of.

AI does pose a real threat but not equally everywhere

Under the most aggressive combination of all three obfuscation passes, the best-performing AI models still managed to successfully reconstruct working code roughly 20–36% of the time on x86 architecture (the processor architecture used in most desktop and server computing). That is not trivial. For every three or four obfuscated functions an attacker feeds to a top-tier AI tool, one or more is likely to be successfully cracked.

ARM architecture is the processor used in virtually all modern smartphones and tablets. With it, the picture is meaningfully better for defenders. Under the same maximum obfuscation, average success rates dropped to just 8.5%, with most models achieving single-digit percentages. The most capable model on ARM assembly achieved 24% success, versus 50% when given a higher-quality decompiled input.

The practical message is that OLLVM's layered protections are genuinely effective on ARM, the primary platform for mobile apps. They are less effective on x86, where AI tools, and especially flagship models, maintain a concerning foothold.

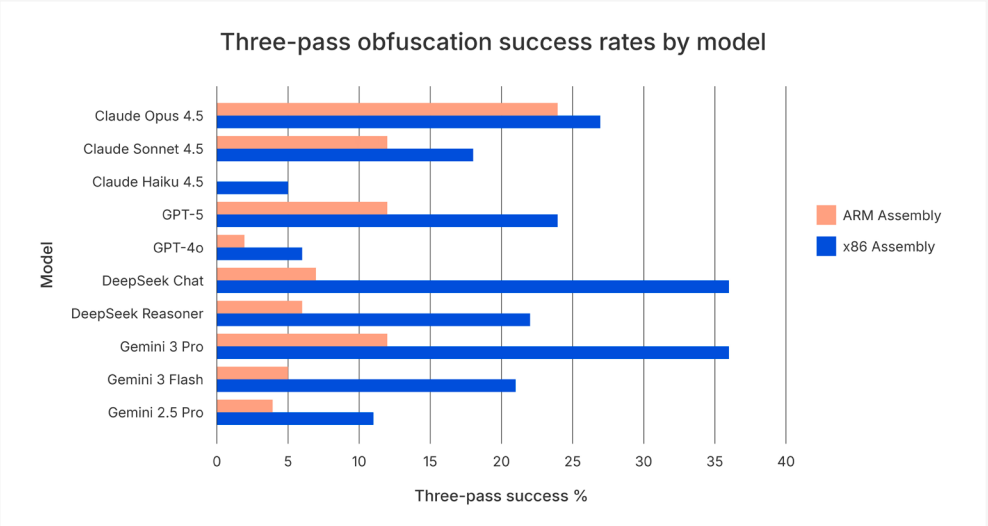
The architecture gap is larger than expected

The gap in AI performance between x86 and ARM is striking, given that the underlying mathematical complexity of the obfuscated code is nearly identical across both. When all three obfuscation passes are applied, the code complexity (as measured by the number of basic building blocks in the program) reaches around 610–611 blocks on both architectures. Structurally, the problem is the same. Yet AI success rates are 2.4 times higher on x86 on average, with individual models showing gaps as large as 5 times.

The most likely explanation is training data. The models tested were trained on enormous quantities of code and documentation drawn from the public internet. x86 assembly language is vastly better represented in that data than ARM. Decades of PC and server computing documentation, reverse engineering blogs, malware analysis write-ups, and forum discussions are predominantly x86. ARM, despite its dominance in mobile computing, is newer to high-profile reverse engineering discourse.

This means ARM-based mobile applications currently benefit from a degree of "natural" AI resistance simply because AI models have less experience with ARM code. That advantage may erode as training datasets expand.

Three-pass obfuscation success rates by model



AI tools have a built-in error rate before obfuscation

This is one of the most important and underappreciated findings in the research, as it fundamentally changes how defenders should think about the numbers. Other findings in this report describe how obfuscation degrades AI performance. But this asks a different

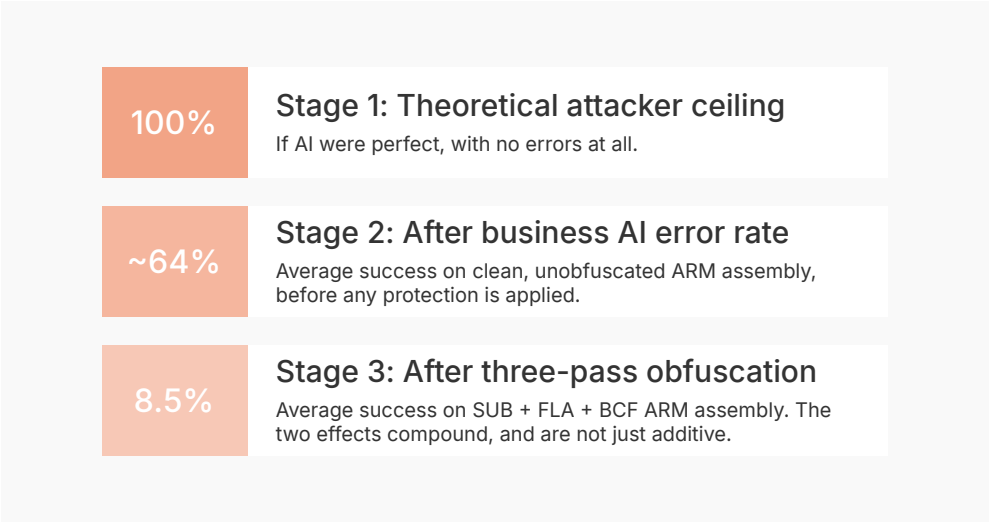
question: *How well do AI deobfuscation tools perform on code that was never obfuscated in the first place?*

The reason the researchers tested this is rooted in how real attacks works. An adversary targeting an application does not know in advance which functions are obfuscated and which are not. In any automated attack pipeline, they would simply run the deobfuscation tool across everything, including both clean code and obfuscated code. The research captures this scenario through the "None" baseline condition, where models were asked to deobfuscate unprotected code.

The results showed that no model in the study achieved above 86% success on clean, unobfuscated code under any input type. GPT-4o on clean x86 assembly managed 48%, meaning more than half of the unobfuscated functions it processed came out corrupted, even though there was nothing to deobfuscate. Claude Opus 4.5 was the strongest model in the entire study. But it failed on 16% of clean ARM pseudocode and 28% of clean ARM assembly. Averaged across all ten models, clean ARM assembly produced only 63.7% success before a single obfuscation technique had been applied.

What this means for defenders is significant. Obfuscation is not pushing an attacker down from a starting point of 100% effectiveness. Rather, it is pushing them down from a starting point that is already well below 100%. The two effects compound together. On three-pass ARM assembly, the average success rate across all models is 8.5%. But on clean ARM assembly with no obfuscation at all, the average is already only 63.7%. Obfuscation does not turn a capable attacker into an ineffective one. It takes someone who was already failing more than a third of the time and making their position dramatically worse.

Theoretical attacker ceiling vs average success rates



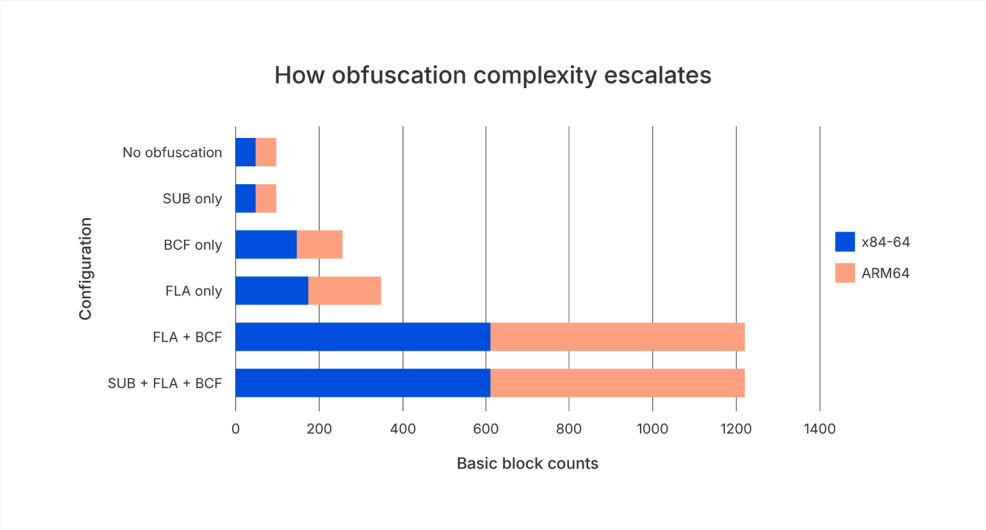
There is also a practical operational cost for attackers running at scale. In any automated pipeline processing a real application with hundreds or thousands of functions, the attacker cannot easily identify and discard the incorrectly deobfuscated outputs before acting on them. Every corrupted function is either flawed logic that gets acted upon (potentially undermining the entire reverse engineering effort) or wasted time spent on manual verification. Across a large codebase, that cost accumulates rapidly.

Obfuscation layers work multiplicatively, not merely additively

One of the most important technical findings relates to how obfuscation techniques interact. When Control Flow Flattening and Bogus Control Flow are applied together, the resulting complexity is not simply the sum of the two techniques. Instead, there is a multiplied effect.

FLA creates a central dispatcher that routes the program's execution. BCF then injects fake branches at every one of those routing points. The more routing points FLA creates, the more fake branches BCF can insert. On x86, this interaction amplifies code complexity by 4.18 times compared to BCF alone. On ARM, the amplification is even stronger at 5.50 times.

How obfuscation complexity escalates



For security engineers, the takeaway is that using FLA and BCF together is not just two measures of protection stacked on top of each other. Rather, it is a force-multiplier that creates significantly stronger resistance than either technique alone. Adding a third pass of Instruction Substitution on top does not further increase structural complexity, but it does add a substantial layer of arithmetic noise that can frustrate attackers.

Decompiler quality changes the equation

Assembly code is the low-level machine instructions of the compiled binary. Pseudecode is a higher-level, more human-readable representation generated by a decompiler tool such as Ghidra. When AI models were given raw assembly code, they performed significantly worse than when given pseudocode.

Claude Opus 4.5, the best overall performer in the study, achieved 50% success on three-pass ARM obfuscation when working from pseudocode, but only 24% when working from raw ARM assembly. For weaker models, the gap was even more dramatic. GPT-4o achieved 10% from pseudocode versus just 2% from raw assembly.

This is important for threat modelling. An attacker using raw binary analysis tools faces a substantially harder task than one who has access to a high-quality decompiler. It also suggests a currently underexplored defensive measure. Techniques that specifically degrade decompiler output quality by making the pseudocode it produces misleading or corrupted could meaningfully increase resistance to AI-assisted reverse engineering beyond what structural obfuscation alone provides.

The fact that not all AI models are equal impacts your threat model

The ten models tested showed wide variation in capability. Top-tier models (Claude 4.5 Opus, DeepSeek Chat and Reasoner, and Gemini 3 Pro) maintained meaningful success rates even under maximum obfuscation. Other models performed significantly worse under the same conditions. Claude 4.5 Haiku and GPT-4o both achieved as little as 1–2% on three-pass ARM assembly. This represents a near-complete failure to reconstruct functional code, regardless of their general capabilities.

This stratification matters when assessing risk. An automated, cost-optimized attack pipeline using smaller models is significantly less threatening than a targeted effort by a sophisticated actor with access to the most capable frontier models. Organizations with high-value IP or facing nation-state-level threats should calibrate their defenses accordingly.

Recommendations

These are recommendations for security and product teams that flow from our findings.

Recommendations for security and product teams

1. Deploy layer obfuscation

Use SUB + FLA + BCF in combination. The multiplicative effect of combined techniques is a far stronger than any single pass

2. Treat ARM and x86 targets differently

ARM targets are 2.4x more resistant to AI attack. X86 deployments require additional protections beyond OLLVM.

3. Exploit the AI error rate

AI tools already fail on clean code. Design defenses that make it harder for attackers to distinguish correct outputs from corrupted ones.

4. Invest in anti-decompilation

Pseudocode gives attackers 2-5x advantage. Decompiler-degrading techniques are a high-value investment currently outside OLLVM's scope.

5. Match your defense to attacker sophistication

Flagship AI models succeed far more often than the smaller ones. High-value targets should assume access to the best available tools.

6. Plan for capability evolution

Today's ARM resistance advantage may erode as training data diversifies. Build in longitudinal re-evaluation of obfuscation effectiveness.

1. Deploy layered obfuscation rather than relying on a single technique

The most important practical finding of this research is that combining Control Flow Flattening and Bogus Control Flow produces dramatically stronger protection than either technique alone. Any OLLVM deployment that uses only one obfuscation type is leaving substantial protection on the table. Three-pass obfuscation (SUB + FLA + BCF) should be the baseline configuration for any application handling sensitive logic or proprietary algorithms.

2. Treat ARM and x86 targets differently

If your primary deployment target is mobile (iOS, Android), the current AI threat landscape is substantially more manageable. ARM assembly is significantly harder for current AI models to deobfuscate, and three-pass OLLVM on ARM provides meaningful protection even against the best available models.

For x86 deployments (e.g., Windows applications, server-side components, or desktop software) OLLVM alone is insufficient against a sophisticated attacker with access to flagship AI models. These targets require additional protection layers, including virtualization-based obfuscation (such as that offered by CoVirt or VxLang), runtime integrity checks, and anti-tampering measures.

3. Recognize that AI tools are imperfect and design defenses to exploit this

The baseline error rate finding has a direct implication for how you think about attacker pipelines. At scale, an automated attack using AI deobfuscation will produce a significant volume of corrupted or incorrect outputs alongside any successes. There are defenses that increase the difficulty of distinguishing good outputs from bad e.g., through increased structural complexity, misleading code patterns, or anti-analysis constructs. These defenses amplify this operational cost for the attacker, not just the raw difficulty of any individual deobfuscation attempt.

4. Consider anti-decompilation as a strategic investment

Current OLLVM transformations make code structurally complex, but they do not specifically target the quality of decompiler output. Since pseudocode input gives AI models a 2–5x improvement in success rates, measures that degrade decompiler output (i.e., corrupted type information, misleading function boundaries, anti-decompiler constructs) represent a potentially high-value defensive investment that sits outside the current OLLVM scope.

5. Factor model capability into your threat model

Your threat landscape may include sophisticated, well-resourced attackers, such as competitors, criminal groups, or state actors. If so, assume they have access to the best available AI tools and know how to leverage them. Under that assumption, top-tier model performance figures (not average figures) are the relevant benchmarks. For organizations facing lower-sophistication automated attacks, the average performance numbers are more appropriate, and current

three-pass OLLVM provides strong protection.

6. Plan for capability evolution

The current AI advantage on x86 versus ARM is at least partly an artefact of training data composition. As AI models are trained on more diverse datasets, and as ARM computing becomes more dominant in data centers and embedded systems, the gap may narrow. Security strategies should not be built on the assumption that today's AI limitations are permanent. Longitudinal evaluation of obfuscation effectiveness should be a recurring part of your security assessment program.

Methodology

This is how our research was conducted.

The Benchmark

Our team developed the Promon Deobfuscation Benchmark (PDB-LLM), a dataset of 200 unique C programs designed specifically for this evaluation. The programs span two domains: 150 general programs covering data transformation tasks (string manipulation, arithmetic, validation), and 50 security-oriented programs reflecting real defensive logic found in mobile application protection, including jailbreak detection, root detection, anti-debugging, and hook detection.

All programs were written with generic function naming (rather than descriptive names like "hash_djb2") to prevent AI models from identifying algorithms by name rather than by analyzing their logic. Programs range from 15 to 70 lines of code. This is long enough to contain meaningful complexity, but short enough to fit within LLM processing limits.

Testing environment and process

Binary preparation

Each of the 200 programs was compiled for both x86-64 and ARM64 architectures using OLLVM, across five obfuscation configurations: SUB alone; FLA alone; BCF alone; FLA+BCF combined; and the full three-pass SUB+FLA+BCF. This produced 2,000 unique obfuscated binaries in total.

Complexity analysis

Structural complexity metrics (e.g., basic block counts, instruction counts, and cyclomatic complexity) were extracted from each binary using IDA 9.2. Ground-truth complexity was established before obfuscation to allow precise measurement of the transformation effect.

Input preparation

Two types of input were prepared for the AI models. Raw assembly was extracted using IDA 9.2 for both architectures. High-level pseudocode (a decompiled, C-like representation) was generated using Ghidra 11.2.1 for ARM64 binaries. Binary identifiers were removed from all inputs to prevent name-based pattern matching.

AI evaluation

Each of the ten models received identical zero-shot prompts, with no hints about the obfuscation techniques used or access to the original source code. This helped ensure that realistic attacker

conditions were simulated. Models were also tested on completely unobfuscated code to establish a clean-code baseline, reflecting the reality that attackers in automated pipelines cannot know in advance which functions are protected. Models were instructed to produce clean, compliable C code implementing the same functionality as the input.

Models evaluated

Claude 4.5 Opus, Claude 4.5 Sonnet, Claude 4.5 Haiku (Anthropic); GPT-5, GPT-4o (OpenAI); Gemini 3 Pro, Gemini 3 Flash, Gemini 2.5 Pro (Google); DeepSeek Chat, DeepSeek Reasoner (DeepSeek).

Validation

Results were validated through a strict two-stage pipeline. First, each AI-generated C program was compiled using GCC. Any output that failed to compile was counted as a failure. Second, successfully compiled programs were executed against the same test inputs used to generate the original ground-truth outputs. An exact match of all outputs, along with matching exit codes, was required for a result to be counted as a success.

Quality scoring

Beyond pass/fail, successfully compiled outputs were scored on a structural simplification scale (0.0–1.0) measuring how closely the AI's output matched the original code in terms of length, complexity, and the removal of obfuscation artefacts.

Citations

Collberg, C., Thomborson, C., and Low, D. (1997). A taxonomy of obfuscating transformations. University of Auckland Technical Report 148.

Brezinski, K. and Ferens, K. (2023). Metamorphic malware and obfuscation: A survey of techniques, variants, and generation kits. Security and Communication Networks.

Raubitzek, S. et al. (2024). Obfuscation undercover: Unraveling the impact of obfuscation layering on structural code patterns. Journal of Information Security and Applications, 85, 103850.

Junod, P. et al. (2015). Obfuscator-LLVM: Software protection for the masses. Proc. IEEE/ACM SPRO, pp. 3–9.

Blazquez, E. and Tapiador, J. (2025). Practical Android software protection in the wild. ACM Computing Surveys, 58(2).

Tkachenko, A. (2026). Promon Deobfuscation Benchmark. https://doi.org/10.5281/zenodo.18377774

Beste, D. et al. (2025). Exploring the potential of LLMs for code deobfuscation. DIMVA, pp. 267–286.

Patsakis, C., Casino, F., and Lykousas, N. (2024). Assessing LLMs in malicious code deobfuscation of real-world malware campaigns. Expert Systems with Applications, 256, 124912.

Tkachenko, A., Suskevic, D., and Adolphi, B. (2025). Deconstructing obfuscation: A four-dimensional framework for evaluating large language models assembly code deobfuscation capabilities. arXiv:2505.19887.

PROMON

Promon leads the way in proactive mobile app security. For 20 years, we've been making the world a safer place by securing any app, on any device—in no time at all.

Today, we protect over 2 billion users, secure 13 billion monthly transactions, and safeguard $2.5 trillion in market cap.

Promon is headquartered in Oslo, Norway, with offices in more than 15 countries around the world.

promon.io

Promon AS
Cort Adelers Gate 30
0251 Oslo
Norway