Executable but Unlearnable: Designing Code that Resists LLM-Based Learning

Anonymous Author(s)

Abstract

The LLM foundation model era has inverted a fundamental assumption in software engineering. Code, once written, no longer belongs exclusively to its creators. Any publicly accessible code becomes training data, absorbed into models that can reproduce, adapt, and redistribute it without consent. This paper argues that such circumstances represent not merely a legal or ethical challenge, but a *technical* one requiring new defensive primitives. We introduce the concept of **statistical opacity**, defined as the deliberate design of code representations that resist neural pattern extraction while preserving human readability and machine executability. We articulate a research agenda spanning theory, mechanisms, tools, and evaluation. Statistical opacity will become as fundamental to software security as cryptography became to data security. Just as the community learned to design systems assuming adversaries could intercept communications, we must now learn to design systems assuming adversaries can learn from code.

Keywords

Statistical opacity, unlearnable code, AI-resistant software, foundation models, software security

ACM Reference Format:

Anonymous Author(s). 2026. Executable but Unlearnable: Designing Code that Resists LLM-Based Learning. In *Proceedings of AIware 2026 (7 July 2026, Montreal, Canada).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In contemporary software security, the efficiency and reliability of protective mechanisms depend on robust threat models and comprehensive defensive strategies. Traditional approaches assume adversaries seek to *execute* unauthorized operations, leading to defenses built around access controls, input validation, memory safety, and cryptographic protocols. These defenses assume adversaries interact with software through its *interfaces*.

Large language models (LLMs) such as GPT-4, Claude, and Gemini invalidate this assumption. The new adversary does not need to execute code. It only needs to *read* it, or, more precisely, to extract statistical patterns from it during training. Once learned, those patterns can be reproduced, adapted, and deployed in ways the original authors never intended and cannot control. This paradigm shift enables attackers to derive capabilities from code without ever running it, fundamentally altering the threat landscape for software protection. The trajectory of this threat is evident in recent developments. In July 2025, researchers at Carnegie Mellon University demonstrated that large language models, when equipped with a structured abstraction layer, can plan and execute sophisticated multi-host cyberattacks on realistic emulated enterprise networks [19]. The system, Incalmo, enables LLMs to specify high-level attack actions that domain-specific agents translate into concrete exploit sequences. Evaluation on MHBench, a benchmark of 40 emulated networks modeled on documented incidents including the 2017 Equifax breach, showed that Incalmo-equipped LLMs achieved attack goals in 37 of 40 environments within 12 to 54 minutes at under $15 in API costs. Without the abstraction layer, even frontier models succeeded in only 3 of 40 environments, underscoring both how close LLMs are to operational offensive capability and how rapidly the remaining scaffolding gap is narrowing.

In November 2025, Google's Threat Intelligence Group identified PROMPTFLUX and related malware families as the first confirmed instances of malicious code that invokes large language model capabilities during execution [7]. Written in VBScript, PROMPTFLUX queries Gemini's API through a "Thinking Robot" module that requests obfuscation and evasion techniques, establishing a recursive cycle of mutation in which the malware periodically regenerates its own source code. Although GTIG assessed PROMPTFLUX as experimental and not yet capable of compromising a victim network, the report characterized this class of malware as a significant step toward more autonomous and adaptive threats. State-sponsored actors from North Korea, Iran, and China have separately been observed misusing Gemini across all stages of attack operations, from reconnaissance and phishing lure creation to command-and-control development and data exfiltration. Evaluation across 1,354 test cases from five real-world projects [21] showed that the decrease in test pass rate ranged from 15.3% under symbol-level obfuscation to 62.5% under combined symbol, structure, and semantic transformations. Even at the highest obfuscation intensity, models retained over a third of their code generation capability, and lighter transformations had minimal effect. The JsDeObsBench study at CCS 2025 [4] found that LLMs achieve 97.23% syntactic and 60.93% semantic deobfuscation on JavaScript programs, with GPT-4o and Codestral successfully deobfuscating more programs than traditional baseline approaches.

As organizations increasingly deploy AI-assisted development tools, several critical challenges emerge:

- Concerns regarding intellectual property protection as proprietary algorithms become training data for foundation models
- Inability to maintain security guarantees when attackers can learn vulnerability patterns from publicly available code

AIware 2026, Montreal, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- Growing evidence that traditional obfuscation provides insufficient protection against neural comprehension

To address these obstacles, we propose **statistical opacity** as a new security primitive. This property ensures that code remains functional for intended users while degrading its utility as training data for machine learning systems. The approach exploits the gap between what makes code executable (syntactic and semantic correctness) and what makes code learnable (statistical regularity).

1.1 Contributions

This work provides a comprehensive vision for AI-resistant software, including formal definitions, theoretical foundations, and a research agenda. The contributions can be summarized as follows:

- Providing a definition and theoretical grounding for statistical opacity as a security primitive
- Identifying candidate mechanisms for achieving opacity in code representations
- Articulating a research agenda for the community

2 Statistical Opacity

More precisely, let P denote a program with defined input–output semantics and let T denote a transformation such that $T(P)$ is semantically equivalent to P for all valid inputs. Let $\mathcal{L}$ represent a gradient-based language model trained on a corpus containing $T(P)$ and let $\mathcal{L}'$ represent a baseline model trained on the same corpus with P in its untransformed form. Statistical opacity holds when the generalization performance of $\mathcal{L}$ on tasks derived from P degrades measurably relative to $\mathcal{L}'$, where degradation is quantified through standard evaluation benchmarks such as Pass@k for code completion or accuracy on vulnerability detection. The transformation T is opacity-conferring to the extent that it preserves execution semantics while maximizing this generalization gap across a defined family of gradient-based learners. This property is distinct from traditional obfuscation, which aims to resist *human* comprehension or *algorithmic* analysis. Statistical opacity specifically targets the learning mechanisms of neural networks. Traditional obfuscation targets any polynomial-time adversary, an impossibly strong guarantee per Barak et al.'s impossibility results [2]. Statistical opacity targets the narrower class of gradient-based learners, a tractable goal.

Recent research demonstrates that LLMs, when integrated into hierarchical agent frameworks with high-level abstraction layers, can function as autonomous red team agents capable of coordinating and executing multi-step cyberattacks across realistic enterprise network environments [19]. As these capabilities grow, so does the value of pattern resistance. Second, foundation model developers have exhausted most publicly available code. The next frontier is private codebases, proprietary systems, and enterprise software. Organizations face a choice between using their code as training data or developing technical countermeasures. Third, the same models that help developers write code can help attackers understand it. Every security-critical system becomes more vulnerable as AI comprehension improves. We propose a fundamental reframing of the security question. The previous formulation asked how to keep adversaries from seeing code. The proposed formulation asks how to ensure that seeing code does not help. This mirrors the conceptual shift in cryptography, where the community stopped trying to hide algorithms and instead designed algorithms that remain secure even when fully known.

We propose a fundamental reframing:

Reframing the problem

From: `"How do we keep adversaries from seeing our code?"`

⇓

To: `"How do we ensure that seeing our code doesn't help?"`

3 Theoretical Foundations

Statistical opacity rests on three theoretical pillars, each representing decades of research waiting to be connected. Prior work [8] established that source code is *more* statistically regular than natural language, exhibiting lower cross-entropy, more predictable patterns, and stronger local regularities. This naturalness is precisely what makes code amenable to machine learning. Our observation is that naturalness is a design choice, not a law of nature. Code is natural because programmers write it naturally. If code is written unnaturally while preserving correctness, learnability can be degraded. Recent work on the Structured Naturalness Hypothesis extends this finding to abstract syntax trees and control flow graphs, demonstrating Zipfian distributions in syntactic structures. This suggests that opacity must operate at multiple levels including lexical, syntactic, and semantic.

Research on unlearnable examples [9] demonstrated that images can be made unlearnable through imperceptible perturbations that prevent models from extracting useful patterns during training. The perturbations exploit how gradient-based learning operates, creating shortcuts that models latch onto instead of genuine features. The key insight is that the learning process can be poisoned without corrupting the data's primary function. Subsequent work [10] adapted this approach to code, using CodeBERT-guided perturbations to make training data unusable while maintaining 67.5% code similarity. Another study [20] demonstrated that untargeted corruption combined with watermark backdoors causes significant model degradation at only 10% poisoning rate. The challenge for code is that perturbations must be discrete (continuous noise cannot be added to tokens) and functionality-preserving (the code must still run).

A theoretical result [3] connects learning and obfuscation as dual operations. If a concept class can be learned with M mistakes, it can be canonically obfuscated in time $O(M \cdot \text{poly}(n))$. Conversely, certain obfuscations are impossible precisely because the dual class is learnable. This duality suggests that making code unlearnable is equivalent to a form of obfuscation optimized against learning-based adversaries. Recent work [6] proved that PAC learnability implies backdoor defendability, but not the converse, with direct implications for the security of code against poisoning attacks.

4 Candidate Mechanisms

We propose three families of mechanisms for achieving statistical opacity, representing promising directions grounded in how current models process code. Modern language models use fixed tokenization schemes (BPE, WordPiece) trained on general text corpora. Code is tokenized using these same schemes, which creates an exploitable structure. The first mechanism involves designing identifiers that fragment in unpredictable ways under standard tokenizers. A variable named `calculateTotal` tokenizes predictably, while one named `cAlCuLaTe__tOtAl` may fragment into many more tokens, disrupting learned representations. Models learn associations between token sequences, so if semantically related code produces wildly different token sequences, cross-example generalization fails. Empirical testing across 13 LLMs on 250 Java problems [15] found that variable renaming alone caused an 18.6% accuracy drop, while encryption of literals caused a 21.4% drop. These effects compound with multiple obfuscation layers.

The second mechanism creates a vocabulary where keywords trigger incorrect priors. Models learn that the keyword `for` strongly predicts iteration, `class` predicts object-oriented structure, and `async` predicts concurrency patterns. In an opacity-enhanced representation, these associations could be systematically violated, leading to transfer learning failure. The model's knowledge becomes a liability rather than an asset. The third mechanism inserts functionally neutral code (dead branches, redundant computations, unreachable paths) crafted to dominate gradient updates. The model spends its capacity learning patterns that have no predictive value. This constitutes an adversarial attack on the training process itself, analogous to data poisoning but operating through the code's structure rather than its labels. Research on covert poisoning [1] demonstrated approximately 20% attack success rates while evading signature-based cleansing.

5 Research Agenda

We propose a phased research agenda spanning theory, mechanisms, tools, and evaluation. During the first phase, the community should formalize statistical opacity by rigorously defining connections among learnability theory, information theory, and program semantics. Establishing baselines requires systematically measuring how existing obfuscation tools affect model learning. The Loki framework [18] serves as the current gold standard for ML-resistant obfuscation, leveraging VM-based protection and formally verified MBA expressions to reduce synthesis success to 19%. Understanding which techniques transfer to neural adversaries is essential. Developing evaluation frameworks requires determining appropriate metrics, such as Pass@k for code completion, accuracy for vulnerability detection, or transfer learning degradation. During the second phase, the community should design opacity-specific transformations that move beyond repurposed obfuscation toward techniques specifically designed for neural resistance. Building source-to-source transpilers requires creating tools that transform standard code into opacity-enhanced representations, with parameterizable, reversible transformations for authorized maintainers. Exploring language-level solutions requires investigating whether certain programming paradigms are inherently more opaque.

During the third phase, IDE and toolchain integration should make opacity as easy to enable as compiler optimization flags. Selective opacity techniques should protect security-critical sections while leaving other code transparent. Opacity-aware development practices should address how teams write code that will later be opacity-transformed. During the fourth phase, red teaming efforts should develop adversarial training methods, preprocessing defenses, and multi-model ensemble approaches that attempt to break opacity. Adaptive opacity systems should detect when their opacity is being probed and respond dynamically. Formal verification should demonstrate that certain transformations guarantee opacity bounds against specific model classes.

6 Applications and Implications

The most immediate application is protecting proprietary code from unauthorized training. Organizations invest substantial resources in developing unique algorithms, optimizations, and implementations. Statistical opacity offers a technical complement to legal protections. Code that implements authentication, cryptography, or access control is especially sensitive. If attackers can learn patterns from similar implementations, they can more easily identify vulnerabilities or craft exploits. CMU research demonstrates that LLMs can already autonomously coordinate multi-step attacks. Opacity makes security-critical code harder to analyze via learning.

As foundation models become infrastructure, questions of consent and compensation for the use of training data become urgent. Statistical opacity gives code authors a technical mechanism for withholding their work from training pipelines, independent of legal frameworks that vary by jurisdiction. We acknowledge that statistical opacity is dual-use. Techniques that protect legitimate software equally protect malware. GTIG's analysis of PROMPTFLUX demonstrates that threat actors are already exploring how to make their code resistant to analysis [7]. This is not a reason to abandon the research. Cryptography faces the same duality, and we do not regret developing it. But it is a reason to proceed thoughtfully, with attention to responsible disclosure and defensive applications.

Several open problems remain. Unlike images, code cannot tolerate continuous noise, so every transformation must preserve syntactic validity and semantic correctness. Models will adapt if opacity techniques become widespread, so defenses must be designed for an adversary that knows the techniques and optimizes against them. Research on deobfuscation pre-training [11] demonstrated that training on deobfuscation yields 12.2% improvement in code translation tasks, suggesting that obfuscation itself can become a training signal. A key research challenge is achieving differential opacity, where transformations degrade machine comprehension while preserving human readability. Scaling these techniques from experimental codebases to production systems with millions of lines represents another essential direction for future work.

7 Scenarios and Feasibility

The research agenda outlined above rests on assumptions about adversary improvement, mechanism robustness, and ecosystem dynamics. This section subjects those assumptions to three stress tests.

7.1 Adversary Evolution

On HumanEval, Pass@1 scores rose from 28.8% [5] to 90.2% [16] to over 96% [17] in three years. These figures reflect the base HumanEval benchmark; on HumanEval+, which adds more rigorous test cases, the highest-performing models score approximately 89%, indicating that headline capability figures overstate robustness. If deobfuscation capability tracks a comparable trajectory, mechanisms designed during Phase 1 of the agenda will confront substantially stronger adversaries by the time Phase 2 designs are validated. This creates a pacing asymmetry. Defense mechanisms require multi-year research cycles. Attack capability improves with each model generation on an annual cadence. The agenda must therefore pursue mechanisms in parallel, accepting that early deployments will rely on defenses whose shelf life may be measured in model generations. A further threat compounds this problem. Neuro-symbolic tools that pair neural pattern recognition with formal program reasoning are already deployed. The IRIS framework combines LLMs with static analysis and detects 103.7% more vulnerabilities than CodeQL alone across Java projects averaging 300,000 lines [14]. MoCQ, a related framework, autonomously generated 12 previously unknown vulnerability patterns that human analysts had missed [12]. AutoBug formalizes LLM-powered symbolic execution, decomposing whole-program reasoning into path-level sub-tasks [13]. These hybrid systems cross-validate neural predictions against formal semantics, reducing the effectiveness of any defense that relies solely on misleading statistical representations. Whether opacity can resist the combination of learning with formal methods is an open problem the current agenda does not address.

7.2 Mechanism Boundary Conditions

Each mechanism fails entirely when a specific assumption is violated. Tokenization adversariality assumes subword tokenization trained on natural code distributions. Character-level and byte-level models eliminate this attack surface, and the trend toward longer contexts and flexible input representations makes tokenizer diversification increasingly likely. Semantic misdirection assumes models trust statistical associations carried from pretraining. An adversary that fine-tunes specifically on opacity-transformed code may learn to discount misdirection patterns. Whether fine-tuning fully recovers comprehension or misdirection creates a persistent penalty is an empirical question the Phase 1 study must resolve. Gradient poisoning assumes adversary pipelines ingest protected code without neutralizing preprocessing. Dead code elimination, a standard compiler optimization, removes many functionally neutral insertions. Effective poisoning must therefore use live constructs whose contribution to program output is negligible, a substantially harder design constraint. No single mechanism survives all plausible configurations. Layered deployment ensures that circumventing any one defense requires a different strategy than circumventing the others. The cost of comprehensive defeat grows multiplicatively with the number of independent layers.

7.3 Adoption Thresholds

Individual protection (transforming a specific codebase) is feasible with current techniques. Systemic protection (degrading models trained on public code) requires adoption at pipeline-contaminating scale. Production pipelines deduplicate aggressively, filter by quality signals, and weight sources by reliability. Opacity-transformed code exhibiting unusual identifiers, anomalous control flow, or inflated binary size may trigger quality filters and be excluded before reaching the training stage. If pipelines filter the majority of transformed code, the nominal adoption rate required for systemic effect becomes impractical. This creates a design tension the research community has not yet characterized. Transformations must disrupt learned representations (the goal of opacity) while preserving natural-looking surface statistics (the requirement for pipeline evasion). Historical adoption of comparable tools suggests 5–15% ecosystem penetration within three to five years of a production release is achievable, likely sufficient for individual protection but below the systemic threshold unless the filter evasion problem is solved.

8 Conclusion

Fifty years ago, the software engineering community faced a new threat: networked adversaries capable of intercepting communications. The response was cryptography, a technical primitive that assumed adversaries would see the data but would not be able to derive any useful information from it. Cryptography is now so fundamental that we cannot imagine secure systems without it. Today, we face an analogous threat: learning-based adversaries that can extract patterns from code. The demonstration that LLM-based agents can execute multi-host attacks on realistic enterprise network environments [19], the documentation of self-rewriting malware harnessing foundation model capabilities [7], and the empirical evidence that obfuscation provides only partial protection against neural comprehension all point to the same conclusion. The response must be statistical opacity, a technical primitive that assumes adversaries will analyze our code but ensures that such analysis will not yield useful capabilities. This is a generational challenge. The code we write today trains the models of tomorrow. If we do not develop defenses, the most valuable software becomes training data for systems we do not control. The unlearnable machine is not a fantasy. It is a research program, waiting to be pursued.

References

[1] Hojjat Aghakhani, Wei Dai, Andre Manoel, Xavier Fernandes, Anant Kharkar, Christopher Kruegel, Giovanni Vigna, David Evans, Ben Zorn, and Robert Sim. 2024. TrojanPuzzle: Covertly Poisoning Code-Suggestion Models . In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1122–1140. doi:10.1109/SP54263.2024.00140

[2] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2012. On the (im)possibility of obfuscating programs. *J. ACM* 59, 2, Article 6 (May 2012), 48 pages. doi:10.1145/2160158.2160159

[3] Elette Boyle, Yuval Ishai, Pierre Meyer, Robert Robere, and Gal Yehuda. 2023. On Low-End Obfuscation and Learning. In *Information Technology Convergence and Services*. https://api.semanticscholar.org/CorpusID:256504838

[4] Guoqiang Chen, Xin Jin, and Zhiqiang Lin. 2025. JsDeObsBench: Measuring and Benchmarking LLMs for JavaScript Deobfuscation. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei, Taiwan) *(CCS '25)*. Association for Computing Machinery, New York, NY, USA, 36–50. doi:10.1145/3719027.3744871

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[6] Paul Christiano, Jacob Hilton, Victor Lecomte, and Mark Xu. 2025. Backdoor Defense, Learnability and Obfuscation. *LIPIcs, Volume 325, ITCS 2025* 325, 38:1–38:21. doi:10.4230/LIPICS.ITCS.2025.38

[7] Google Threat Intelligence Group. 2025. AI Threat Tracker: Advances in Threat Actor Usage of AI Tools. Google Cloud Blog. https://cloud.google.com/blog/topics/threat-intelligence/threat-actor-usage-of-ai-tools

[8] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 837–847.

[9] Hanxun Huang, Xingjun Ma, Sarah Monazam Erfani, James Bailey, and Yisen Wang. 2021. Unlearnable Examples: Making Personal Data Unexploitable. arXiv:2101.04898 [cs.LG] https://arxiv.org/abs/2101.04898

[10] Zhenlan Ji, Pingchuan Ma, and Shuai Wang. 2022. Unlearnable Examples: Protecting Open-Source Software from Unauthorized Neural Code Learning. 525–530. doi:10.18293/SEKE2022-066

[11] Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: a deobfuscation pre-training objective for programming languages. In *Proceedings of the 35th International Conference on Neural Information Processing Systems (NIPS '21)*. Curran Associates Inc., Red Hook, NY, USA, Article 1147, 13 pages.

[12] Penghui Li, Songchen Yao, Josef Sarfati Korich, Changhua Luo, Jianjia Yu, Yinzhi Cao, and Junfeng Yang. 2025. Automated Static Vulnerability Detection via a Holistic Neuro-symbolic Approach. arXiv:2504.16057 [cs.CR] https://arxiv.org/abs/2504.16057

[13] Yihe Li, Ruijie Meng, and Gregory J. Duck. 2025. Large Language Model Powered Symbolic Execution. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 385 (Oct. 2025), 29 pages. doi:10.1145/3763163

[14] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=9LdJDU7E91

[15] Serge Lionel Nikiema, Jordan Samhi, Abdoul Kader Kaboré, Jacques Klein, and Tegawendé F. Bissyandé. 2025. The Code Barrier: What LLMs Actually Understand? arXiv:2504.10557 [cs.SE] https://arxiv.org/abs/2504.10557

[16] OpenAI. 2024. GPT-4o System Card. OpenAI Technical Report.

[17] OpenAI. 2024. Learning to Reason with LLMs. OpenAI Blog.

[18] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2021. Loki: Hardening Code Obfuscation Against Automated Attacks. doi:10.48550/arXiv.2106.08913

[19] Brian Singer, Keane Lucas, Lakshmi Adiga, Meghna Jain, Lujo Bauer, and Vyas Sekar. 2025. Incalmo: An Autonomous LLM-assisted System for Red Teaming Multi-Host Networks. arXiv:2501.16466 [cs.CR] https://arxiv.org/abs/2501.16466

[20] Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. 2022. CoProtector: Protect Open-Source Code against Unauthorized Training Usage with Data Poisoning. In *Proceedings of the ACM Web Conference 2022* (Virtual Event, Lyon, France) *(WWW '22)*. Association for Computing Machinery, New York, NY, USA, 652–660. doi:10.1145/3485447.3512225

[21] Yuanliang Zhang, Yifan Xie, Shanshan Lit, Ke Liu, Chong Wang, Zhouyang Jia, Xiangbing Huang, Jie Song, Chaopeng Luo, Zhizheng Zheng, Rulin Xu, Yitong Liu, Si Zheng, and Xiang-Ke Liao. 2025. Unseen Horizons: Unveiling the Real Capability of LLM Code Generation Beyond the Familiar. 604–615. doi:10.1109/ICSE55347.2025.00082