MITIGATING VULNERABILITY LEAKAGE FROM LLMS FOR SECURE CODE ANALYSIS

by
BENGÜ GÜLAY

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
July 2025

MITIGATING VULNERABILITY LEAKAGE FROM LLMS FOR SECURE CODE ANALYSIS

Approved by:

Prof. CEMAL YILMAZ . . . .
(Thesis Advisor)

Asst. Prof. DİLARA KEKÜLLÜOĞLU . . . .

Assoc. Prof. ALİ FURKAN KAMANLI . . . .

Date of Approval: July 18, 2025

Bengü Gülay 2025 ©

All Rights Reserved

ABSTRACT

MITIGATING VULNERABILITY LEAKAGE FROM LLMS FOR SECURE CODE ANALYSIS

BENGÜ GÜLAY

Computer Science and Engineering, M.Sc. Thesis, July 2025

Thesis Supervisor: Prof. Cemal Yılmaz

Keywords: vulnerability detection, information leakage, obfuscation, honeypots, code privacy

Large Language Models (LLMs) are increasingly integrated into software development workflows, offering powerful capabilities for code analysis, debugging, and vulnerability detection. However, their ability to infer and expose vulnerabilities in source code raises security concerns, particularly regarding unintended information leakage when sensitive code is shared with these models. This thesis investigates defense strategies to mitigate such leakage: traditional obfuscation techniques and a novel deception-based approach involving honeypot vulnerabilities. We constructed a dataset of 400 C and Python code snippets spanning 51 CWE categories and evaluated their vulnerability detection performance across three state-of-the-art LLMs: GPT-4o, GPT-4o-mini, and LLaMA-4. Firstly, we applied obfuscation methods—including comment removal, identifier renaming, control/data flow transformations, dead code insertion, full encoding, and LLM-based rewriting—and measured their impact on LLM detection accuracy and functionality retention. Dead code insertion and control flow obfuscation proved most effective in suppressing vulnerability leakage, though aggressive techniques like encoding impaired functionality comprehension. Secondly, we introduced honeypot vulnerabilities combined with misleading strategies that were proven effective earlier—such as control flow obfuscation, data flow obfuscation, and identifier renaming—and additional techniques like cyclomatic complexity increases and misleading comments. Honeypots significantly reduced vulnerability detection accuracy by over 60 percentage points in some cases, while maintaining high functional clarity, with LLM-generated similarity scores consistently above 4.1 on a 5-point scale. Misleading comments emerged as a lightweight yet robust defense across all models. These findings underscore the need to balance security and usability in AI-assisted development and highlight ethical considerations, as similar techniques could potentially be misused to conceal malicious flaws from automated audits.

ÖZET

GÜVENLI KOD ANALIZI İÇIN BÜYÜK DIL MODELLERINDEN KAYNAKLANAN ZAFIYET SIZINTISININ AZALTILMASI

BENGÜ GÜLAY

Bilgisayar Bilimi ve Mühendisliği, Yüksek Lisans Tezi, Temmuz 2025

Tez Danışmanı: Prof. Dr. Cemal Yılmaz

Anahtar Kelimeler: güvenlik açığı tespiti, bilgi sızıntısı, karartma, bal küpü, kod gizliliği

Büyük Dil Modelleri (LLM'ler), yazılım geliştirme süreçlerinde giderek daha fazla kullanılarak kod analizi, hata ayıklama ve güvenlik açığı tespiti gibi alanlarda oldukça fazla destek sağlamaktadır. Ancak özellikle hassas kodların bu sistemlerle paylaşılması durumunda, bu modellerin kaynak kodlardaki güvenlik açıklarını ifşa etme ihtimali ve istenmeyen bilgi sızıntılarına yol açabileceği endişeleri göz ardı edilmemelidir. Bu tez, söz konusu bilgi sızıntısını azaltmaya yönelik savunma stratejilerini sunmaktadır: geleneksel obfüskasyon (karmaşıklaştırma) teknikleri ve bal küpü (honeypot) güvenlik açıklarını içeren yenilikçi, aldatmaya dayalı bir yaklaşım. Çalışma kapsamında, 51 farklı CWE kategorisini kapsayan 400 C ve Python kod parçasından oluşan bir veri kümesi oluşturulmuş ve üç güncel LLM (GPT-4o, GPT-4o-mini ve LLaMA-4)'in güvenlik açığı tespit performansları bu kodların üzerinde değerlendirilmiştir. İlk olarak sekiz farklı obfüskasyon yöntemi uygulanarak, bunların LLM'lerin tespit doğruluğu ve işlevselliğin korunması üzerindeki etkileri ölçülmüştür. Sonuçlar, ölü kod ekleme ve kontrol akışı obfüskasyonunun güvenlik açığı sızıntısını engellemede en etkili yöntemler olduğunu; ancak tam şifreleme gibi agresif tekniklerin, kodun işlevselliğinin anlaşılmasını olumsuz etkilediğini göstermiştir. İlk aşamada elde edilen sonuçlar doğrultusunda, başarılı olan obfüskasyonlar diğer yanıltıcı stratejilerle birleştirilerek, bal küpü güvenlik açıkları kodlara eklenmiştir. Bu aşamada birinci fazda etkili olduğu kanıtlanan kontrol akışı obfüskasyonu, veri akışı obfüskasyonu ve tanımlayıcı yeniden adlandırma tekniklerinin yanı sıra, siklomatik karmaşıklığın artırılması ve yanıltıcı yorumlar gibi yeni yöntemler de uygulanmıştır. Elde edilen bulgular, bal küplerinin bazı durumlarda güvenlik açığı tespit doğruluğunu yüzde 60'tan fazla azalttığını; buna karşın kodun işlevselliğinin büyük ölçüde korunduğunu ortaya koymuştur. Ayrıca, yanıltıcı yorumlar tüm modellerde hafif fakat etkili bir savunma yöntemi olarak öne çıkmıştır. Bu bulgular, yapay zekâ destekli yazılım geliştirme süreçlerinde güvenlik ve kullanılabilirlik arasında bir denge kurulması gerekliliğini vurgulamakta; ayrıca benzer tekniklerin otomatik denetimlerden kötü niyetli açıkları gizlemek amacıyla kötüye kullanılabileceğine dair etik kaygılara dikkat çekmektedir.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Prof. Dr. Cemal Yılmaz, for his exceptional guidance and support throughout my thesis. Despite his demanding schedule, he always found the time to provide insightful feedback and valuable advice, which greatly contributed to the development and completion of this work.

I am also thankful to my thesis committee members, Assist. Prof. Dr. Dilara Keküllüoğlu and Assoc. Prof. Dr. Ali Furkan Kamanlı, for their time, thoughtful comments, and constructive suggestions that have improved the quality of this thesis.

I am deeply grateful to my family for their unwavering support, patience, and encouragement during my studies. Their love and trust have been my greatest strength. I would also like to thank someone whose quiet yet constant support and steadfast belief in me have been invaluable during the most challenging moments. Finally, I would like to thank my close friends, whose understanding and motivation have helped me navigate this journey with resilience.

I am grateful to all who have contributed to this important chapter of my academic journey.

To those I hold dear

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Large Language Models (LLMs) have become an important tool in modern software development, helping developers write, debug, refactor, and optimize code across various programming languages. These models provide simplified workflows and reduced development time, often rivaling traditional static analyzers in speed and breadth. Their role in static analysis, vulnerability detection, and code understanding is growing rapidly (Nam, Macvean, Hellendoorn, Vasilescu & Myers, 2024). However, the very power of these models also raises significant security and privacy concerns: once code is shared with LLMs, whether for assistance, review, or integration, they may reveal sensitive logic or vulnerabilities that developers never intended to disclose (Sallou, Durieux & Panichella, 2024). This introduces a new and underexplored form of leakage risk, especially in proprietary or security-critical code.

Although LLMs are highly capable of identifying security flaws in source code, they are not designed with security in mind. This means they can unintentionally expose critical information during analysis (Zhao, Li & Wang, 2022). A major concern is whether LLMs can infer and reveal security vulnerabilities that the user may not have been aware of or did not intend to disclose (Carlini, Tramèr, Wallace, Jagielski, Herbert-Voss, Lee, Roberts, Brown, Song, Erlingsson, Oprea & Raffel, 2021). This raises important questions about information leakage and the effectiveness of different mitigation strategies. In this thesis, we use the term information leakage to describe situations where an LLM reveals security vulnerabilities in code, which we consider sensitive, particularly when the code is proprietary or shared with external tools. While this differs from traditional privacy-related leakage, it still represents a form of unintended information exposure during code analysis.

Recent incidents have highlighted the real-world risks of sharing sensitive code or data with public LLMs. In one high-profile case, Samsung engineers inadvertently leaked confidential source code and meeting notes by submitting them to ChatGPT for assistance with chip testing and documentation (Paganini, 2023). The data, now stored on OpenAI's servers, creates long-term exposure risks. As a result, Samsung banned employees from using external LLMs and began developing internal alternatives. Similarly, Apple, Amazon, JPMorgan, and other corporations have prohibited or restricted the internal use of ChatGPT and GitHub Copilot to prevent leakage of proprietary logic and intellectual property (Wodecki, 2023). India's Finance Ministry and several European institutions have issued similar advisories, reflecting a growing concern over LLM-driven data exposure (Singh & Ohri, 2025). Moreover, technical vulnerabilities—such as Microsoft's EchoLeak zero-click exploit (Lakshmanan, 2025) and Google Vertex AI's model exfiltration flaw (Montalbano, 2024)—demonstrate how attackers may extract sensitive data even without user intent. These events underscore a common theme: code shared with LLMs, whether intentionally or indirectly, can become a liability—visible to external providers, exploitable through prompt injection, or stored without clear retention policies.

These real-world events reinforce a key distinction at the heart of this thesis: information leakage from LLMs does not only arise from training-time memorization of datasets (i.e., data leakage), but also during inference—when proprietary code is submitted and analyzed. In this thesis, we use the term information leakage to describe situations where an LLM reveals security vulnerabilities present in the submitted code to the LLM's servers, which means sensitive details about proprietary systems may be stored or inferred by external service providers. If such information is later exposed—intentionally or through a breach—it could provide attackers with a roadmap to exploit specific organizations. This concern is amplified in high-stakes domains like finance, defense, or infrastructure, where even the inference of a vulnerability represents a risk.

We distinguish between data leakage, where models reveal memorized training data, and information leakage, where models infer latent security weaknesses from previously unseen code during analysis (OWASP, 2024). Our focus is on mitigating this second type, which requires not just the submission of data, but also the model's ability to reason about that data and extract implicit meaning. While all code shared with an LLM constitutes data, not all such data leads to information leakage. Information leakage arises when the model interprets the code, identifies underlying security flaws, and surfaces knowledge that was not explicitly disclosed by the user. This interpretive capability transforms seemingly neutral input into a potential source of sensitive insights. In the context of proprietary or security-critical systems, this distinction becomes crucial, as the unintended inference of vulnerabilities poses serious risks—even when no private training data is involved.

Existing defenses against information leakage often trade interpretability or maintainability for limited gains in confidentiality. Furthermore, they do not fully address the unique capabilities of LLMs, which can reason through many syntactic disguises and extract high-level patterns. This highlights a fundamental tension: how can developers retain the benefits of LLM-powered tools while reducing the risk of leaking critical vulnerabilities? Addressing this question is essential for the safe adoption of AI-assisted development workflows.

This thesis investigates approaches for mitigating vulnerability leakage from LLMs: traditional obfuscation techniques and a novel deception-based strategy involving honeypot vulnerabilities.

In the first phase of our study, we aim to quantify how much sensitive information can be leaked when using LLMs to analyze source code and how obfuscation techniques affect this leakage. Specifically, we investigate how many security vulnerabilities an LLM can detect in a given code and how different obfuscation techniques impact this detection rate. To address this, we built a dataset of 400 C and Python code snippets, each containing security vulnerabilities classified into 51 distinct Common Weakness Enumeration (CWE) categories. Using an automated process, we ran these snippets through the ChatGPT-4o-mini API (OpenAI, 2022) and measured how accurately it could detect vulnerabilities. We then applied several obfuscation techniques—including comment, string, identifier, control flow, data flow obfuscations, dead code insertion, full encoding, and LLM-based obfuscation—to assess their effectiveness in reducing information leakage. At the same time, we evaluated whether the obfuscated code still functioned correctly to maintain the utility of LLMs for legitimate tasks.

Our results from Phase 1 reveal a nuanced trade-off between security and functionality. While techniques such as dead code insertion and advanced control flow transformations were particularly effective in reducing vulnerability detection accuracy, they also caused moderate disruption to the LLM's ability to understand functionality. Simpler obfuscations like comment and identifier obfuscation had a smaller impact, suggesting that LLMs rely on structural patterns of code rather than mere textual cues. Encoding the entire codebase, although highly disruptive, was found to severely impair both vulnerability detection and functionality comprehension, rendering it impractical for real-world applications. These findings emphasize the importance of selecting obfuscation strategies that balance protection against information leakage with the need to preserve the code's semantic clarity.

Building on these insights, we decided to incorporate honeypots into the obfuscation techniques that had yielded successful results. The second phase of our study explores a novel and practical approach to misleading LLMs by introducing honeypot vulnerabilities—synthetic, intentionally misleading code segments designed to distract and confuse automated analyses. Our core hypothesis is that by embedding believable but irrelevant weaknesses into a program, one can reduce an LLM's ability to identify real, exploitable flaws, without compromising the functional clarity of the code. To investigate this, we evaluated the effectiveness of honeypots in combination with code transformations such as misleading comments, identifier renaming, and control or data flow obfuscation.

In Phase 2, we tested these deception strategies against three state-of-the-art LLMs: GPT-4o, GPT-4o-mini, and LLaMA-4. Our evaluation focused on two critical dimensions: vulnerability leakage (i.e., how accurately the LLM could detect real CWEs) and functionality retention (i.e., whether the LLM could still correctly interpret what the code does). We show that LLM-generated honeypots, combined with additional misleading strategies, significantly reduced vulnerability detection accuracy—by over 60 percentage points in some cases—while maintaining a strong understanding of code functionality, with models scoring above 4.1 on a 5-point scale.

Among these strategies, misleading comments emerged as a particularly robust and model-agnostic method. This approach exploits the language-sensitive reasoning of LLMs and requires minimal code modification, making it a lightweight yet powerful defense. Similarly, increasing control flow complexity in tandem with honeypots proved effective against models like GPT-4o-mini and LLaMA. However, GPT-4o exhibited greater resistance to honeypots, suggesting that larger and more advanced models may require more sophisticated or tailored deception techniques.

Finally, we systematically explored eight hybrid strategies combining honeypots with various obfuscation methods. This analysis revealed that certain combinations—such as honeypots with misleading comments or moderate cyclomatic complexity increases—achieve near-parity with LLM-generated honeypots, enabling more controllable and interpretable defenses.

This thesis makes several key contributions: (1) it introduces a high-diversity dataset spanning 51 CWE types for evaluating LLM behavior; (2) it empirically demonstrates the susceptibility of state-of-the-art LLMs to both obfuscation and deception-based defenses; (3) it proposes and validates a novel defense mechanism based on LLM-generated honeypot vulnerabilities; and (4) it provides a structured evaluation framework for functionality retention using LLM-generated explanations.

In summary, this work highlights a critical but underexplored risk in AI-assisted software development: the automatic inference and exposure of latent vulnerabilities

from seemingly benign code shared with LLMs. Our findings reveal that while LLMs are powerful tools for detecting vulnerabilities, they can be systematically misled through lightweight code transformations that preserve functional semantics. These insights offer actionable guidance for developers and security practitioners seeking to balance usability, security, and resilience in AI-supported workflows, while also underscoring the need for future research into deception-resilient LLM architectures and standardized evaluation benchmarks.

2. RELATED WORK

LLMs have emerged as a transformative technology in software engineering, providing developers with unprecedented capabilities in code generation, debugging, optimization, and vulnerability detection (Nam et al., 2024). Models such as OpenAI's GPT series, Meta's LLaMA family, and open-source efforts have demonstrated that LLMs can analyze and produce source code across diverse programming languages, rivaling or surpassing traditional static analyzers in some security tasks (Mohamed, Assi & Guizani, 2025). Their integration into development pipelines promises faster delivery cycles and improved code quality.

However, this power comes with significant security and privacy concerns, particularly regarding information leakage when sensitive or proprietary code is shared with these models. Zhou et al. (Zhou, Weyssow, Widyasari, Zhang, He, Lyu, Chang, Zhang, Huang & Lo, 2025) highlighted that LLMs trained on large-scale public datasets are prone to unintentional memorization of sensitive training data, resulting in leakage when these models are queried. It is important to note that this leakage arises from pre-training exposure rather than prompt-based interactions, as LLMs do not retrain on prompt inputs unless fine-tuned explicitly. Their empirical analysis across 83 software engineering benchmarks revealed leakage rates of 4.8% (Python), 2.8% (Java), and 0.7% (C/C++), with certain datasets showing extreme leakage ratios of 100% and 55.7%, respectively. Similarly, Sallou et al. (Sallou et al., 2024) demonstrated that implicit data leakage from LLMs can inflate benchmark performance and compromise security when previously seen code is regurgitated during inference. These findings suggest that while LLMs offer powerful tools for vulnerability detection and code assistance, they also introduce risks of exposing latent vulnerabilities in proprietary codebases, even without explicit prompting.

This dual nature of LLMs being both effective detectors and potential leakers has motivated research into defensive techniques aimed at safeguarding code privacy. Two prominent directions have emerged: code obfuscation to hinder LLM comprehension and deception strategies, such as honeypot vulnerabilities, to mislead automated analyses. Our work builds on these foundations by systematically evaluating both approaches in mitigating vulnerability leakage while preserving functional clarity.

2.1 LLMs for Vulnerability Detection

Several studies have demonstrated that LLMs can rival or surpass traditional static analysis tools in vulnerability detection tasks. A study conducted in 2023 investigates ChatGPT for vulnerability detection in Python source code and compares its results with static application security testing tools (Bakhshandeh, Keramatfar, Norouzi & Chekidehkhoun, 2023). The study has four different types of experiments, and its results showed that ChatGPT reduces the false positive and false negative rates. A more recent and comprehensive study on vulnerability detection with LLM compared the vulnerability detection capabilities of four LLMs (ChatGPT-3.5, ChatGPT-4, LLaMA-2, and Bard-Gemini) against ten traditional static analysis tools (Yıldırım, Aydın & Çetin, 2024). Their results showed that LLMs, particularly ChatGPT-4, outperformed traditional static analyzers with a much higher accuracy rate.

In addition to these studies, another related work compared the performances of six general-purpose LLMs and six open-source LLMs which are specifically trained for vulnerability detection for further investigation (Guo, Patsakis, Hu, Tang & Casino, 2024). Their findings showed that, even though both models were successful, fine-tuned models performed better in their specific areas. However, the general-purpose models had more stable accuracy across different datasets. As the LLMs' success in detecting vulnerabilities is proven in many studies, several tools have emerged leveraging LLMs for automated vulnerability detection, including prompt engineering frameworks and hybrid static-dynamic analysis systems (Boi, Esposito & Lee, 2024; Ding, Liu, Piao, Song & Ji, 2025; Lu, Ju, Chen, Pei & Cai, 2024). As one of the tools, Zhang et al. (Yan & Li, 2021) proposed combining LLMs with static analyzers for Java code vulnerability detection (Bench-Java dataset). Their evaluation across CWE categories such as CWE-22, CWE-78, CWE-79, and CWE-94 showed improved detection coverage but highlighted persistent false positives. These results align with observations in our study, where even state-of-the-art models occasionally misclassify obfuscated or deceptive inputs.

Despite their strengths, LLMs exhibit limitations in vulnerability detection. Their reliance on surface-level patterns makes them vulnerable to misdirection through lexical changes, semantic-preserving transformations, or adversarial inputs Li, Dutta & Naik (2025). False positives remain a significant challenge, especially in complex or obfuscated codebases. Moreover, recent studies have shown that many evaluations of LLMs are context-deprived, often assessing models on isolated code snippets rather than full execution or data-flow contexts. This leads to inaccurate conclusions and flawed rationales, as LLMs either misidentify vulnerabilities or justify correct predictions for the wrong reasons (Li, Li, Wu, Xu, Zhang, Cheng, Xu & Zhong, 2025). Even state-of-the-art models like GPT-4 and DeepSeek struggle to generalize to real-world scenarios, frequently exhibiting reasoning errors and overthinking biases. Moreover, Steenhoek et al. (Steenhoek, Rahman, Roy, Alam, Tong, Das, Barr & Le, 2025) demonstrate that LLMs consistently fail at multi-step semantic reasoning tasks critical for detecting vulnerabilities, with performance rarely exceeding random guessing across diverse prompts and architectures. These errors stem not only from a lack of training on execution-specific data but also from limitations in understanding semantic subtleties such as bounds and NULL checks. Scaling models or fine-tuning on more code has shown limited benefits, indicating that improving vulnerability detection may require fundamentally new architectures or training paradigms.

2.2 Obfuscation Techniques and Their Impact on LLMs

Alongside vulnerability detection, code obfuscation to prevent leaking sensitive information has also been a key focus of research on LLMs. It has long been used in software protection to obscure code logic from reverse engineering or automated analysis. Previous works have demonstrated the effectiveness of prompt obfuscation in preventing information leakage (Li, Wen & Jin, 2024; Pape, Mavali, Eisenhofer & Schönherr, 2025). In the code obfuscation area, Lin et al. introduced CodeCipher, a technique that obfuscates code at the token level by perturbing embeddings within an LLM's matrix (Lin, Wan, Fang & Gu, 2024). Their work demonstrated that altering token embeddings could hinder LLM-based code completion, translation, and summarization tasks while preserving functionality. Metrics such as Pass@K (code completion success rate), BLEU-4, ROUGE-L, and METEOR (code summarization accuracy) were used to measure obfuscation performance. While their study focused on obfuscation's impact on LLM-assisted programming tasks, our work evaluates obfuscation in the context of vulnerability detection, investigating how different techniques affect information leakage.

Mai et al. proposed ConfusionPrompt, a method for private inference that obfuscates user queries to LLMs (Mai, Yang, Yan, Ye & Pang, 2024). Their technique generates pseudo-prompts alongside genuine queries, making it difficult for adversaries to extract sensitive information. Although their research was primarily focused on protecting prompt privacy, the underlying principles of query obfuscation are relevant to our study, particularly in understanding how obfuscation disrupts an LLM's ability to extract meaningful security insights.

Another study by Fan and Li investigated the challenges LLMs face when summarizing "lexically confusing code" (Li et al., 2025). They introduced Variational Eroded Code Summarization (VECOS), which replaces meaningful identifiers with generic symbols before reconstructing functionally accurate summaries. Their approach forced models to rely on deeper code structures rather than superficial lexical cues. This research aligns with our investigation into identifier obfuscation and its impact on vulnerability detection.

2.3 Honeypot Strategies for Misleading LLMs

While prior work has explored honeypot strategies involving LLMs, these efforts primarily focus on using the models to simulate realistic interactive environments for trapping human attackers. For example, Otal et al. proposed an SSH-based system in which LLMs act as conversational agents to deceive adversaries by mimicking legitimate shell behavior in real time (Otal & Canbaz, 2024). Similarly, Sladić et al. introduced shelLM, a generative honeypot system that leverages LLMs to dynamically engage users during intrusion attempts (Sladić, Valeros, Catania & Garcia, 2024). Another approach is using LLMs to simulate interactive honeypots or analyze attack logs, but none have treated the LLM itself as the target of deception through embedded code-level honeypots (Lanka, Gupta & Varol, 2024). While these studies apply LLMs to support or automate honeypot defense systems, our work takes the inverse approach: we treat the LLM itself as the target of deception by embedding honeypot vulnerabilities within code, aiming to mislead the model and reduce vulnerability leakage.

There are also several recent efforts that specifically aim to mislead or subvert large language models through targeted adversarial inputs. Geiping et al. (Geiping, Stein, Shu, Saifullah, Wen & Goldstein, 2024) systematically categorize and execute a wide range of adversarial attacks such as misdirection, role hacking, and glitch token exploitation that coerce LLMs into producing unintended outputs, including misinformation, system prompt leaks, or even contradicting hard-coded safety policies. While their focus is on prompt-level manipulation to violate alignment, our work shares the broader goal of misleading LLMs but does so through embedded code-level deceptions rather than linguistic attacks. Similarly, Rajeev et al (Rajeev, Ramamurthy, Trivedi, Yadav, Bamgbose, Madhusudan, Zou & Rajani, 2025) introduce CatAttack, a method for generating query-agnostic adversarial triggers that significantly increase the likelihood of incorrect answers from reasoning-focused LLMs. These triggers—short, irrelevant phrases like trivia or vague numerical suggestions—can be appended to any math problem and still degrade model accuracy. Like our approach, CatAttack emphasizes minimal perturbations that preserve human interpretability while misleading the model, further reinforcing the feasibility of subtle adversarial manipulations across diverse domains.

2.4 Summary of Contributions

This thesis advances the understanding of LLM vulnerability analysis by systematically exploring both obfuscation and deception-based defenses. In Phase 1, we differentiate our work from prior studies by directly bridging two previously separate areas: LLM-based vulnerability detection and code obfuscation. While earlier research focused on LLM performance or obfuscation for general privacy, our study uniquely evaluates how various obfuscation techniques—including comment removal, identifier renaming, control and data flow transformations, dead code insertion, full encoding, and LLM-based rewriting—impact the ability of LLMs to detect vulnerabilities in source code. By applying these transformations to a diverse dataset of vulnerable C and Python code snippets, we provide a comprehensive analysis of information leakage in obfuscated code and highlight the trade-offs between reducing detectability and preserving functionality.

In Phase 2, we introduce a novel, deception-based strategy that goes beyond traditional obfuscation or prompt-level manipulations. Our approach embeds honeypot vulnerabilities—realistic yet irrelevant flaws—directly into source code to mislead LLM-based vulnerability detectors. We systematically assess the effectiveness of these honeypots when combined with lightweight misleading techniques such as misleading comments, identifier renaming, and cyclomatic complexity increases. Unlike prior work that treats LLMs as tools for building honeypots or focuses on adversarial prompt engineering, our study treats the LLM itself as the target of deception. This allows us to identify which combinations of techniques are most effective at suppressing vulnerability leakage across multiple state-of-the-art models, all while ensuring functional clarity through LLM-based functionality assessments.

Together, these contributions position our work as the first to deliver a fine-grained, model-aware defense framework that integrates both obfuscation and deception strategies. By providing empirical evidence on the effectiveness of these techniques, we offer practical guidance for developers seeking to mitigate unintended vulnerability exposure in AI-assisted software workflows.

3. PRELIMINARIES

In this section, we introduce the foundational concepts that underpin our study, including LLMs that are used in the study, the use of honeypot vulnerabilities in code, the attention mechanism of LLMs, Mean Reciprocal Rank, and the cyclomatic complexity metric. These concepts provide the theoretical basis for our methodology and help interpret the impact of misleading patterns on LLM-based vulnerability detection.

3.1 Model Overview

This study compares the vulnerability detection behaviors of three LLMs: GPT-4o, GPT-4o-mini, and LLaMA-4 Scout 17B 16E Instruct.

GPT-4o and GPT-4o-mini are proprietary models developed by OpenAI (OpenAI, 2022). GPT-4o is their latest flagship multi-modal model with advanced reasoning capabilities and an extended context window, while GPT-4o-mini is a lighter-weight variant optimized for lower latency and cost. Although parameter counts are not publicly disclosed, GPT-4o is significantly larger and more capable, whereas GPT-4o-mini represents a trade-off between performance and efficiency with an estimate of around 10–20B parameters.

LLaMA-4 Scout 17B 16E Instruct, released by Meta AI, is a fine-tuned, instruction-following model from the LLaMA-4 family (MetaAI, 2025). It utilizes a Mixture-of-Experts (MoE) architecture with 17 billion parameters and 16 experts, of which only a subset is activated per forward pass. This structure allows for computational efficiency without sacrificing representational power. The model has been instruction-tuned for general-purpose use and supports extremely long contexts, up to 10 million tokens, making it well-suited for code analysis tasks.

Together, these models offer a diverse range of capabilities and serve as a robust basis for evaluating the effectiveness of honeypots and other misleading techniques across different LLM architectures and sizes.

3.2 Honeypot Vulnerabilities

Honeypots are intentionally crafted artifacts designed to attract or mislead automated systems or adversaries. Originally developed in cybersecurity to detect or trap malicious actors by simulating vulnerable systems or data, honeypots function as decoys, drawing attention away from critical assets while collecting behavioral data from attackers (Spitzner, 2002).

In the context of source code, honeypot vulnerabilities are artificial or irrelevant weaknesses inserted into software with the goal of distracting automated analysis tools or models. These vulnerabilities often resemble real-world patterns (such as buffer overflows or improper input validation), but are either unreachable, harmless, or logically benign. When indistinguishable from genuine vulnerabilities, honeypots can serve as effective false positives, thereby reducing the likelihood that actual flaws are identified.

In this work, we adapt the concept of honeypots to the domain of LLM-based vulnerability leakage. Instead of targeting human attackers, our honeypots are designed to mislead language models into focusing on these synthetic vulnerabilities rather than the true security flaws present in the code. This technique allows us to systematically evaluate the susceptibility of LLMs to deception and test whether the model's attention can be redirected through targeted code manipulations.

3.3 Attention Mechanism in LLMs

The attention mechanism is a core component of modern LLMs, such as those based on the Transformer architecture. It enables the model to weigh the importance of different parts of the input when generating outputs. Instead of processing tokens in isolation or fixed order, attention allows the model to dynamically focus on the most relevant segments of the input context during each step of inference (Niu, Zhong & Yu, 2021).

In practice, this mechanism assigns scalar attention weights to tokens or token groups, indicating how much influence they should have on the model's understanding or prediction. These weights are computed based on the relationships between tokens, enabling the model to capture long-range dependencies, semantic structure, and contextual cues. The self-attention layers within the model aggregate these weighted inputs to build a comprehensive representation of the data (Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser & Polosukhin, 2017).

In the context of code analysis, this means that LLMs will allocate more cognitive "focus" to code regions that appear syntactically complex, semantically meaningful, or potentially vulnerable. However, this same mechanism can be exploited. By inserting misleading or artificial code fragments that mimic the appearance of real vulnerabilities, it is possible to divert the model's attention away from genuine security flaws.

In our work, we leverage this property of LLMs by obfuscations and strategically embedding honeypot vulnerabilities designed to appear important to the model. By placing these decoy patterns in structurally and semantically plausible locations, we aim to attract a disproportionate share of the model's attention, thereby reducing its ability to detect real vulnerabilities elsewhere in the code.

3.4 Mean Reciprocal Rank (MRR)

Mean Reciprocal Rank (MRR) is an evaluation metric commonly used in information retrieval and recommendation systems to measure the effectiveness of models that return a ranked list of results (Radev, Qi, Wu & Fan, 2002; Voorhees, 1999). It provides a quantitative assessment of how high the correct answer appears in the ranking.

Formally, MRR is defined as the average of the reciprocal ranks of the first relevant result for a set of queries. The reciprocal rank for a query is calculated as $1/rank$, where $rank$ is the position of the first correct result in the model's output. For instance, if the correct vulnerability is ranked first, the reciprocal rank is 1; if it appears third, the reciprocal rank is 1/3. The MRR is computed as follows:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{3.1}$$

where $|Q|$ is the total number of queries and $rank_i$ is the position of the first correct result for the i^{th} query.

In this study, MRR is particularly useful because the LLMs often return multiple possible vulnerabilities as ranked lists. Using MRR allows us to assess how effectively the models prioritize real vulnerabilities by placing them earlier in their outputs. A higher MRR reflects better ranking quality, which is especially important in scenarios where only the top results are reviewed during vulnerability assessment.

3.5 Cyclomatic Complexity

Cyclomatic complexity is a software metric used to quantify the complexity of a program's control flow. It measures the number of linearly independent paths through a program, which is determined by the presence of decision points such as conditionals (if, switch), loops (for, while), and function calls that introduce branching logic (McCabe, 1976). A higher cyclomatic complexity generally indicates a more intricate and potentially harder-to-understand control structure (Esposito, Janes, Kilamo & Lenarduzzi, 2024).

McCabe (McCabe Jr., 2008) proposed interpreting CC values in relation to software security risk: 1–10 (simple, low risk), 11–20 (moderate complexity, moderate risk), 21–50 (high complexity, high risk), and >50 (very high risk, often untestable). He argued that increased complexity inherently reduces security because complex systems are harder to design, implement, test, and use securely, leading to more security vulnerabilities.

This metric is widely used in software engineering to assess code maintainability, test coverage requirements, and potential fault-proneness. In the context of automated analysis and LLM-based tools, more complex code structures may introduce additional cognitive load, possibly affecting the accuracy of vulnerability detection (Sheng, Chen, Gu, Huang, Gu & Huang, 2025).

In our work, we use cyclomatic complexity as an analytical dimension to examine whether structural complexity correlates with LLM performance. After inserting honeypots into our code samples, we calculate the cyclomatic complexity of the original, pre-honeypotted code using a static analysis tool. This ensures that the complexity scores reflect only the inherent logic of the real program, not the misleading elements added later. We then compare complexity scores between correctly and incorrectly classified samples to explore whether more complex control flows hinder the model's ability to detect genuine vulnerabilities.

4. METHODOLOGY

This chapter outlines the experimental methodology employed to evaluate how effectively obfuscation and honeypot techniques mitigate information leakage from LLMs. Our methodology is structured around three key components: the dataset, the two-phase defense strategy, and functionality evaluation. First, we describe the dataset of 400 C and Python code snippets covering 51 different vulnerabilities. Then, we detail two sequential phases: Phase 1 focuses on traditional obfuscation techniques applied individually to the dataset, while Phase 2 introduces honeypot vulnerabilities and their combinations with obfuscation methods. Finally, we present our approach for assessing functionality retention, ensuring that applied transformations do not compromise the semantic integrity of the code.

4.1 Datasets

To evaluate how much information could be leaked through LLMs and the effectiveness of various obfuscation techniques and honeypots, we created a dataset of 400 code snippets, each containing at least one known vulnerability. The dataset includes 300 C code snippets and 100 Python snippets with a total of 51 distinct Common Weakness Enumeration (CWE) categories. While there are existing public datasets for vulnerability detection, most are either limited in size, containing only small numbers of vulnerable examples, or focus on a very narrow set of CWE types, which significantly restricts their coverage. Our dataset addresses this gap by offering both variety and quantity, enabling a more comprehensive evaluation of LLM behavior across diverse vulnerability types. We achieve this by combining multiple datasets and supplementing them with additional code samples from GitHub repositories and other relevant sources, thereby expanding both the breadth and diversity of vulnerabilities included. Specifically, the dataset includes code snippets from the following four sources:

- CWE Top 25 Most Dangerous Software Weaknesses (2024 Edition): CWE Top 25 is an annual list published by the MITRE Corporation, identifying the most critical and prevalent software security weaknesses based on real-world exploit data (MITRE, 2024). In our study, we specifically targeted C and Python implementations corresponding to the 2024 CWE Top 25 entries. Due to limitations in available public examples and the need for clarity in vulnerability expression within these two languages, we ultimately incorporated code samples representing 12 of the top 25 CWEs. These samples were carefully selected to ensure they exhibit clear, self-contained vulnerabilities suitable for automated analysis and LLM-based detection.

- Open Source Security Foundation's Secure Coding Guide for Python: This guide, developed by the Open Source Security Foundation (OpenSSF) (Open Source Security Foundation (OpenSSF), 2025), provides well-documented Python code snippets annotated with specific CWE identifiers. These examples are primarily educational, designed to help developers—particularly those new to secure coding—recognize and avoid common software vulnerabilities. Each code sample is accompanied by a brief explanation of the underlying weakness, making it a useful resource for both learning and structured analysis. In our study, we selected a subset of these Python examples to supplement the real-world vulnerabilities gathered from other sources. These samples contributed to the balance between educational clarity and real-world complexity in our dataset.

- CVEfixes Dataset: The CVEfixes dataset originates from the study "CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software" (Bhandari, Naseer & Moonen, 2021), which provides a curated set of real-world code vulnerabilities and their corresponding fixes. The dataset focuses primarily on C and C++ programs, offering concrete examples of software weaknesses as they appeared in actual projects. For this study, we extracted and used only the samples that belonged to four clearly defined CWE types. We excluded the generic "CWE-Other" category to maintain consistency and interpretability in our evaluation. The selected samples provide concrete, traceable instances of vulnerable code that complement the more educational and illustrative examples in the rest of our dataset.

- GitHub Repositories with CWE-Labeled Code: To enhance the diversity of our dataset, we manually collected additional code samples from open-source GitHub repositories where vulnerabilities were explicitly labeled with CWE identifiers. In selecting these samples, we considered several factors, including the size of the vulnerable code fragment, the programming language used, and the presence of supporting feedback such as issue discussions or commit messages that reinforced the assigned CWE classification. These examples reflect a broad spectrum of coding styles, development contexts, and vulnerability representations found in real-world software projects.

By collecting code samples from these reputable sources, we ensured a diverse and representative dataset for our analysis. Each snippet in the dataset is labeled with its corresponding CWE identifiers, enabling structured analysis and reproducibility. The dataset provides a solid foundation for both vulnerability detection experiments and the subsequent evaluation of honeypot and obfuscation strategies introduced in this work.

4.2 Phase 1: Mitigating with Obfuscation

In the first phase of our study, we systematically applied a suite of traditional obfuscation techniques to the dataset to examine their ability to hinder LLM-based vulnerability detection. This phase serves as a baseline, exploring how techniques such as comment removal, string encoding, identifier renaming, control and data flow alterations, dead code insertion, and LLM-based transformations affect information leakage. Each obfuscation method was tested independently to isolate its impact on the detection performance of the LLM and its ability to preserve code functionality. The results from this phase provided critical insights into the strengths and limitations of obfuscation as a standalone defense mechanism, informing the design of more advanced strategies in Phase 2.

4.2.1 Obfuscations

To assess the impact of code obfuscation on information leakage and vulnerability detection by LLMs, we applied several obfuscation techniques to the dataset one by one. These techniques were designed to obscure various aspects of the code while preserving the comprehensibility of the functionality.

4.2.1.1 Comment Obfuscation

Comments often provide insights into the code's purpose and logic. To hinder understanding and analysis, all comments were identified and removed from the code snippets.

4.2.1.2 String Obfuscation

Strings can reveal key information, such as function parameters, error messages, or credentials. Obfuscating strings can make the code harder to interpret. Each string in the code was XOR-encoded using the hexadecimal value 0x42, rendering it unreadable until decoded at runtime.

4.2.1.3 Identifier Obfuscation

Function and variable names often provide clues about a program's behavior. All function and variable names were replaced with randomly generated names. Each identifier was consistently replaced with its corresponding random name to ensure accuracy.

4.2.1.4 Encoding the Entire Code

The entire code snippet was transformed using Base64 encoding, making it more difficult to analyze directly.

4.2.1.5 LLM-Based Obfuscation

The original code was processed through a different LLM which is local and privacy-preserving. For LLM-based obfuscation, we used a locally hosted CodeLlama-7B model, chosen for its proven effectiveness in prior code obfuscation studies and its ability to run efficiently on resource-limited hardware (Lin et al., 2024; Mohseni, Mohammadi, Tilwani, Saxena, Ndawula, Vema, Raff & Gaur, 2025). The obfuscation prompt was standardized across all samples: "Obfuscate the following code so that it is functionally equivalent but harder for a human to read or understand. Avoid changing core logic." This prompt was applied uniformly to each code snippet using an automated script to ensure consistency. However, it's important to note that different LLMs can produce varying obfuscation styles and results, depending on their architecture and training data.

4.2.1.6 Control Flow Obfuscation

This obfuscation manipulates the logical structure of a program. We applied two sub-techniques: Loop and Conditional Alteration and Opaque Predicates. For loop and conditional alteration, loops and conditional statements were restructured, sometimes by adding dummy conditions or extra loops that do not affect the loop semantically. For opaque predicates, statements that always evaluate to a fixed value (e.g., if (2 + 2 == 4)) were inserted to create the appearance of complex logic. Both techniques were applied in varying forms and combinations across the dataset, but without a strict pattern, resulting in natural variation that reflects realistic obfuscation use cases.

4.2.1.7 Data Flow Obfuscation

It alters how variables interact, making it harder to track dependencies and understand the program's behavior. For split variables, some variables were divided into multiple parts that were combined only when needed. For redundant computations, extra calculations were introduced that did not impact the semantics but increased complexity. Also, some dummy variables were added to clutter the code.

4.2.1.8 Dead Code Insertion

Inserting non-functional code increases overall complexity and misleads analysis. We inserted useless instructions that have no effect on the program's execution (e.g., x = 0; if(x) doSomething();), and dummy variables/functions in order to increase ambiguity. We inserted both useless instructions and dummy variables/functions randomized across samples, that help reduce the chance that the LLM could recognize a single repeated pattern across all obfuscated samples.

Each of these obfuscation techniques was applied individually to the dataset, ensuring that only one obfuscation method was tested at a time. This approach allowed us to evaluate the effectiveness of each technique in isolation, measuring its impact on information leakage and functionality retention without the influence of combined obfuscations.

4.2.2 Leakage Detection Setup

To systematically assess how much information can be extracted from code snippets, we implemented an automated leakage detection process using a structured approach. We utilized the gpt-4o-mini-2024-07-18 version of ChatGPT-4o mini API for vulnerability detection, ensuring consistency across all experiments. A Python-based script was developed to send obfuscated code snippets to the LLM, capture the model's responses, and store the detected vulnerabilities in a structured database for further analysis. All results were stored in a database, allowing direct comparison between the original vulnerabilities and vulnerabilities detected in the code snippets by the LLM. The API was accessed through predefined requests, ensuring standardized interactions with the model.

Each code snippet was sent to LLM using new clients with erased history and cache to ensure unbiased evaluation. Two different prompts were used: one directly asked for security issues, while the other limited the number of outputs.

Prompt without output limitation:

> "You are a software security expert. Check the following code for security issues. Only list issues that are actually in the code. If you list extra CWE codes, you will be punished. If there are no issues, say 'no'. If

there are issues, list the relevant CWE numbers."

Prompt with output limitation: This prompt was tested with different limits of 1, 3, 5, 7, and 10 CWE codes. Below is an example with a maximum of 10 CWE codes:

> "You are a software security expert. Check the following code for security issues. Only list issues that are actually in the code. If you list extra CWE codes, you will be punished. If there are no issues, say 'no'. If there are issues, list the relevant CWE numbers in order of likelihood, from most to least probable with a maximum of 10."

The LLM's response was directly stored in the database without modification, and CWE codes were extracted from the response for analysis. The same procedure was also applied to obfuscated versions of the code snippets. Each obfuscation technique was tested independently, ensuring that the impact of individual obfuscation methods could be assessed in isolation. The detected vulnerabilities were compared against the known CWE classifications assigned to each code snippet in the database.

To evaluate the effectiveness of the LLM in detecting vulnerabilities from both original and obfuscated code, we used accuracy, F-score, and Mean Reciprocal Rank (MRR) as our primary metrics, which are commonly used evaluation metrics in vulnerability detections (Adam, Bulut, Sow, Ocepek, Bedell & Ngweta, 2022; Brama, Dery & Grinshpoun, 2022). Accuracy provided a straightforward measure of how well the model detected vulnerabilities, while F-score indicated the overall detection performance. For code snippets with more than one vulnerability, we assigned an accuracy of 1 if the model detected at least one of them. For the F-score, we compared the results against all the vulnerabilities present in the code. MRR is an evaluation metric used in information retrieval and recommendation systems, and is particularly valuable in situations where the model provides multiple possible vulnerabilities, as it rewards models that rank the correct issues higher (Radev et al., 2002). It checks the ranking of the correct result that the user is looking for. For each query, the reverse order of the correct result, reciprocal rank, is taken and the average is calculated for all queries. In other words, the earlier the correct answer is reached, the higher the MRR. In our project, MRR was used to capture the ranking quality of the vulnerabilities listed by the model. These evaluation metrics allowed us to quantify the extent of information leakage and determine the effectiveness of different obfuscation techniques in reducing detectability while preserving the LLM's ability to extract functionality.

4.3 Phase 2: Mitigating with Honeypots

Honeypots are deliberately inserted elements meant to mislead or divert the attention of automated systems, traditionally used in cybersecurity to detect or trap malicious actors (Spitzner, 2002). In the context of this work, we adapt the concept of honeypots to the domain of code analysis with LLMs. Instead of deceiving human attackers, our goal is to distract the LLM from real vulnerabilities by embedding artificial or misleading vulnerabilities into the source code, along with additional techniques, to divert attention away from actual security risks.

This approach leverages the attention-based architecture of LLMs, which assigns higher weights to tokens or code regions it deems important during inference (Niu et al., 2021). By injecting carefully designed, realistic-looking but ultimately irrelevant vulnerable patterns (honeypots), we aim to draw the model's attention away from the actual weaknesses present in the code.

Concretely, we implement honeypots by inserting real, functioning vulnerabilities, additional CWE instances not originally present, into vulnerable code snippets as decoys. These honeypots are designed to appear exploitable to LLMs but are semantically irrelevant or isolated from critical code paths, ensuring they act as distractions rather than introducing actual risk. For example, an original snippet with a buffer overflow vulnerability (CWE-120) may be augmented with a format string vulnerability (CWE-134) or integer overflow (CWE-190), embedded in separate code regions. These additional vulnerabilities are syntactically valid and semantically exploitable, ensuring they represent genuine weaknesses rather than superficial artifacts.

The honeypotted code samples were generated by prompting an LLM to preserve the original vulnerabilities while automatically injecting additional CWE-labeled vulnerabilities. To further mislead the attention mechanism of the model, we incorporated several complementary obfuscation strategies, including misleading comments, control-flow and data-flow obfuscation, identifier renaming, dead code insertion, which showed successful results for decreasing the vulnerability leakage in Phase 1 section 4.2, and increased complexity. These transformations were generated by the LLM and manually verified for correctness and functionality.

To evaluate the effectiveness of this honeypot approach, we tested three different LLMs (GPT-4o, GPT-4o-mini, and LLaMA-4) by prompting them with both the original and honeypotted code snippets. Each model was asked to identify actual vulnerabilities with a new, separate client with cleared history and cache with the same prompt used in Phase 1:

> "You are a software security expert. Check the following code for security issues. Only list issues that are actually in the code. If you list extra CWE codes, you will be punished. If there are no issues, say 'no'. If there are issues, list the relevant CWE numbers."

We measured accuracy by checking if the models correctly identified the original vulnerability, comparing results across all 400 samples to quantify the impact of honeypots in distracting the models and reducing vulnerability leakage.

Details of each obfuscation technique used in conjunction with honeypots are described in the following section 4.3.1.

4.3.1 Techniques Applied with the Honeypots

To enhance the effectiveness of our honeypot strategy and further reduce vulnerability leakage, we applied a set of other techniques like obfuscation alongside the injection of extra CWE vulnerabilities. These techniques were designed to guide the LLM's attention away from the original vulnerability by altering the code's surface structure, semantic clarity, and token importance, without breaking functionality. The selection of these specific obfuscation methods was informed by findings from Phase 1 section 4.2.1, where several of them were shown to meaningfully reduce LLM detection rates. All transformations were performed by LLMs in a semi-automated fashion and manually verified to ensure the code remained executable and contained both the original and the injected vulnerabilities.

- Misleading Comments: For each injected honeypot vulnerability, we added explicitly misleading comments intended to draw attention to the artificial flaws. These comments directly describe the extra vulnerabilities (e.g., "// possible format string issue here") to make them more salient to the LLM.

Our goal was to prime the model to focus on the added CWE rather than the original vulnerability. Since prior work shows LLMs are heavily influenced by code comments, this proved to be a lightweight but powerful method for shifting their focus.

- Identifier Obfuscation: We applied identifier renaming to selectively obfuscate function and variable names associated with the original vulnerability, while keeping the identifiers related to the honeypotted code clear and human-readable. Unlike the previous phase, which used randomly generated names, this study employed strategically neutral but ambiguous names to preserve readability while masking semantic cues. As shown in earlier experiments, identifier obfuscation effectively reduces vulnerability detection performance in LLMs, likely due to interference with learned token associations.

- Control Flow Obfuscation: To obscure the logical execution path, we employed classic control-flow obfuscation strategies, including loop restructuring, nested conditionals, and opaque predicates. For example, we added dummy loops, swapped conditional branches, and inserted always-true or always-false statements such as if (2 + 2 == 4) to give the illusion of deeper logic. These changes increase the syntactic distance between the original vulnerability and the rest of the code, potentially diffusing model attention.

- Data Flow Obfuscation: We disrupted variable interactions through data-flow obfuscation. This included the use of split variables, redundant calculations, dummy intermediate variables, and unnecessary assignments. These modifications altered how data propagated through the code, obfuscating the semantic context around the vulnerable operations and reducing the LLM's ability to infer the original flaw from variable behavior or value dependencies.

- Cyclomatic Complexity Increase: As part of the broader obfuscation strategy, we also increased the cyclomatic complexity of the honeypotted versions. This was achieved by introducing additional branching, nesting, and redundant control structures. The goal was to make the code structurally more complex, increasing the cognitive load on the model and introducing noise that could reduce its ability to accurately detect the original vulnerabilities.

4.3.2 Cyclomatic Complexity

To investigate whether the structural complexity of source code affects the ability of LLMs to detect real vulnerabilities in the presence of honeypots, we conducted a dedicated analysis using cyclomatic complexity as a quantitative measure.

Cyclomatic complexity represents the number of independent execution paths in a program and is influenced by control flow constructs such as conditionals, loops, and function calls. Rather than measuring the complexity of the honeypotted versions, which may be artificially inflated due to inserted misleading structures, we computed the cyclomatic complexity of the original, pre-honeypotted code. This ensured that the metric reflected the inherent logical complexity of the functional code.

The complexity scores were calculated using a static analysis tool prior to any LLM-generated manipulation. Once the honeypots were added and detection results were obtained, we divided the honeypotted samples into two groups: those where the LLM successfully identified the original vulnerability and those where it failed to do so. We then compared the cyclomatic complexity distributions between these groups.

This analysis aimed to reveal whether higher structural complexity correlates with increased detection failure in LLMs, suggesting that models may be more susceptible to distraction when processing code with intricate control flow.

4.3.3 Evaluating Technique Combinations with Honeypots

To better understand which techniques are most effective at diverting LLM attention when combined with honeypot vulnerabilities, we conducted a targeted evaluation of selected strategies. While LLM-generated honeypots applied many transformations simultaneously—often producing effective but overly complex and unrealistic samples—our goal here was to disentangle this process and analyze the contribution of each technique, both in isolation and in combination.

All variants in this analysis were based on code snippets with a single inserted honeypot vulnerability. These honeypots were added manually to maintain consistency across samples and to avoid LLM prompt artifacts. To simulate realistic and controlled scenarios, we then selectively applied a subset of techniques previously described (e.g., misleading comments, identifier renaming, control and data flow obfuscation, and structural complexity increases).

The following configurations were tested, where honeypot represents the additional vulnerabilities:

- Honeypot and Misleading Comment
- Honeypot and Identifier Renaming
- Honeypot and Control Flow Obfuscation
- Honeypot and Data Flow Obfuscation
- Honeypot and Cyclomatic Complexity Increase
- Honeypot, Cyclomatic Complexity Increase and Misleading Comment
- Honeypot, Misleading Comment and Control Flow Obfuscation
- Honeypot, Cyclomatic Complexity Increase, Misleading Comment and Control Flow Obfuscation

For each configuration, we measured LLM vulnerability detection accuracy across 400 code samples, using the same prompt, considerations, and scoring method as in earlier stages. The same three LLM models GPT-4o, GPT-4o-mini, and LLaMA-4 were used for evaluation. This allowed us to compare the performance impact of each technique or combination when used alongside honeypots.

Through this analysis, we aimed to identify the most efficient and minimally invasive ways to mislead LLM-based vulnerability detectors—without relying on heavy, automated rewriting.

4.4 Functionality Understanding Evaluation

To assess whether our interventions interfered with the LLM's ability to understand code behavior, we conducted a functionality understanding evaluation using the following three-step method after each phase.

This evaluation measures how well the LLM preserves functional comprehension after transformations such as obfuscation, honeypot insertion, and structural modification. Although our techniques aim to obscure vulnerabilities, it is important that they do not disrupt the functional semantics of the code as understood by the model.

The evaluation was carried out as follows:

1.1 Original Explanation Generation: Each original (unmodified) code snippet was submitted to the LLM with a prompt requesting a functional explanation. This explanation was saved as the reference for comparison.

1.2 Obfuscated/Honeypotted Explanation Generation: The transformed version of the same code was sent to a fresh LLM session—ensuring no shared context with the first step—and the same prompt was used to generate an explanation of the modified code's functionality.

1.3 Similarity Scoring: In a new, independent session, the LLM was presented with both explanations (original and modified) and asked to rate how well the original code's functionality is preserved in the modified code's explanation on a scale from 0 to 5. A score of 5 indicated that the functional description of the modified code closely matched that of the original, while a score of 0 indicated no meaningful similarity.

This scoring reflects how much the applied transformations disrupted the LLM's understanding of the code's intended functionality. High similarity scores suggest strong functionality retention, whereas lower scores point to potential semantic distortion introduced by the transformations.

By applying this process consistently across our dataset, we were able to evaluate the trade-off between vulnerability obfuscation and semantic clarity—an essential aspect for scenarios where the LLM is expected to reason about program behavior post-transformation.

5. RESULTS

5.1 Phase 1: Mitigating with Obfuscation

5.1.1 Vulnerability Leakage

The vulnerability leakage of the LLM in detecting vulnerabilities was measured across different obfuscation techniques using various metrics with ChatGPT-4o mini. The results indicate that obfuscation techniques generally reduce the model's ability to detect vulnerabilities, though the degree of reduction varies depending on the type of obfuscation and the limitations placed on the model's output.

5.1.1.1 Accuracy without Output Limitation

The accuracy of the LLM in detecting vulnerabilities without any output limitations was calculated for each obfuscation technique. As expected, no obfuscation resulted in the highest accuracy of 0.6363 which is consistent with previous work Yıldırım et al. (2024). Obfuscation techniques such as comment, string, and identifier obfuscation showed a minor reduction in detection capability, indicating moderate effectiveness in reducing detectability. More complex obfuscation methods like encoding, control flow, and data flow obfuscation led to a significant reduction in accuracy. Among all the techniques, dead code obfuscation was the most effective in reducing vulnerability detection, resulting in the lowest accuracy of 0.5189.

Table 5.1 Accuracy of vulnerability detection for each obfuscation technique

Obfuscations	**Accuracy**
No Obfuscation	0.6363
Comment Obfuscation	0.6032
String Obfuscation	0.6030
Identifier Obfuscation	0.5992
Encoded	0.5449
Obfuscated by LLM	0.6032
Control Flow Obfuscation	0.5598
Data Flow Obfuscation	0.5516
Dead Code Insertion	0.5189

After evaluating accuracy, we further analyzed the F-score to gain a deeper understanding of the LLM's performance in detecting vulnerabilities. Unlike accuracy, which only considers correct predictions, the F-score accounts for both precision and recall, making it particularly relevant given that the LLM often identified more CWEs than necessary, sometimes detecting multiple potential vulnerabilities in a single instance. As shown in Table 5.2, the F-scores were relatively low across all obfuscation techniques, reinforcing our observation that the model produced a significant number of false positives. While no obfuscation yielded the highest F-score (0.2065), obfuscation techniques like encoding (0.1640) and control flow obfuscation (0.1786) significantly lowered the model's effectiveness. Dead code insertion, which had the most substantial impact on accuracy, also resulted in one of the lower F-scores (0.1799), further emphasizing its effectiveness in concealing vulnerabilities. These results highlighted a critical limitation in the LLM's detection capability: while it could identify vulnerabilities, it often did so imprecisely, leading us to explore methods to refine its outputs. Given the prevalence of false positives, we decided to impose output limitations, restricting the number of reported vulnerabilities to improve precision and overall detection quality.

Table 5.2 F-score for vulnerability detection across different obfuscation techniques

Obfuscation	**F-score**
No Obfuscation	0.2065
Comment Obfuscation	0.2058
String Obfuscation	0.2017
Identifier Obfuscation	0.1883
Encoded	0.1640
Obfuscated by LLM	0.1973
Control Flow Obfuscation	0.1786
Data Flow Obfuscation	0.1823
Dead Code Insertion	0.1799

5.1.1.2 Accuracy with Output Limitations

The accuracy of vulnerability detection was also evaluated with different output limitations, specifically limiting the number of detected vulnerabilities to 1, 3, 5, 7, and 10. The results show that as the output limit increased, the accuracy generally improved. For example, in the case of no obfuscation, the accuracy increased from 0.4867 for a 1-list limit to 0.7199 for a 10-list limit. Obfuscation techniques, such as comment and string obfuscation, resulted in similar trends, with increased accuracy when the output limitation was higher. However, dead code obfuscation still showed the lowest accuracy across all limits, emphasizing its effectiveness in reducing vulnerability detection.

Table 5.3 Accuracy of vulnerability detection with different prompt limitations

Obfuscation	**Max 1**	**Max 3**	**Max 5**	**Max 7**	**Max 10**
No Obfuscation	0.4867	0.6329	0.6443	0.6978	0.7199
Comment Obf	0.4732	0.5988	0.6201	0.6452	0.6783
String Obf	0.4511	0.6025	0.6092	0.6782	0.6782
Identifier Obf	0.4322	0.6191	0.6223	0.6554	0.6691
Encoded	0.4016	0.5133	0.5559	0.5957	0.6543
Obf by LLM	0.4415	0.6250	0.6064	0.6489	0.6941
Control Flow Obf	0.3830	0.5479	0.5957	0.6064	0.6729
Data Flow Obf	0.4016	0.5399	0.5532	0.6090	0.6543
Dead Code Ins	0.3245	0.5186	0.5452	0.5824	0.6197

5.1.1.3 Mean Reciprocal Rank (MRR)

To further evaluate the LLM's ability to rank vulnerabilities accurately, we calculated the MRR for each obfuscation technique with different output limitations. The MRR values were lower across the board compared to accuracy, reflecting the reduced performance of the LLM in ranking the vulnerabilities. However, as with accuracy, the MRR generally improved as the output limit increased. For instance, no obfuscation resulted in an MRR of 0.4867 for the 1-list limit, improving to 0.5017 for the 10-list limit. Again, dead code obfuscation showed the lowest MRR, indicating its effectiveness in obfuscating vulnerabilities.

The results for both accuracy and MRR highlight that obfuscation techniques generally reduce the LLM's ability to detect and rank vulnerabilities, with dead code

Table 5.4 Mean Reciprocal Rank (MRR) for vulnerability detection with different prompt limitations

Obfuscation	**Max 1**	**Max 3**	**Max 5**	**Max 7**	**Max 10**
No Obfuscation	0.4867	0.5532	0.5199	0.5296	0.5017
Comment Obf	0.4732	0.5271	0.4991	0.4912	0.4880
String Obf	0.4511	0.5405	0.4961	0.5083	0.4817
Identifier Obf	0.4322	0.4867	0.4950	0.4806	0.4983
Encoded	0.4016	0.4581	0.4435	0.4433	0.4710
Obf by LLM	0.4415	0.5335	0.4945	0.4987	0.4806
Control Flow Obf	0.3830	0.4741	0.4970	0.4673	0.4604
Data Flow Obf	0.4016	0.4756	**0.4336**	0.4636	0.4378
Dead Code Ins	**0.3245**	**0.4446**	0.4426	**0.4267**	**0.4358**

insertion being the most effective technique in reducing detectability. However, limiting the number of vulnerabilities detected by the LLM generally leads to an improvement in performance, particularly for techniques that would otherwise be harder to detect.

5.1.2 Functionality Retention

To assess how well obfuscation techniques preserve the LLM's ability to extract functionality, we evaluated the functional similarity between original and obfuscated code. The results indicate that while some techniques maintain high comprehensibility of the functionality, others significantly alter the code's behavior.

Table 5.5 Functionality retention scores for each obfuscation technique

Obfuscations	**Score**
No Obfuscation	-
Comment Obfuscation	4.8840
String Obfuscation	4.5634
Identifier Obfuscation	4.7609
Encoded	1.2553
Obfuscated by LLM	3.3165
Control Flow Obfuscation	4.0984
Data Flow Obfuscation	3.8431
Dead Code Insertion	3.5957

Comment obfuscation and identifier obfuscation showed minimal impact and preserved comprehensibility of functionality well. String obfuscation resulted in a slightly lower but still high score. Control Flow and Data Flow Obfuscation had moderate effects. Dead Code Insertion further reduced functionality retention. Obfuscated by LLM impacted comprehensibility of functionality more significantly. Encoding had the most severe effect, resulting in the lowest comprehensibility of functionality retention.

5.2 Phase 2: Mitigating with Honeypots

5.2.1 Effectiveness of LLM-Generated Honeypots

To evaluate the impact of LLM-generated honeypots on vulnerability leakage, we compared the accuracy of LLMs in identifying known CWEs before and after honeypot injection. To establish a baseline, we first evaluated the ability of GPT-4o-mini to correctly identify vulnerabilities in the original code samples in Phase 1. In Phase 2, we extended this evaluation to include GPT-4o and LLaMA-4, using the same methodology. The detection accuracy was measured as the proportion of true CWE matches.

The results for the baseline are presented in Table 1 under the "Accuracy (Original)" column. LLaMA achieved the highest baseline accuracy of 0.678, closely followed by GPT-4o, while GPT-4o-mini showed a lower detection capability at 0.636. These scores indicate that in the absence of obfuscation or honeypots, LLMs can effectively detect known vulnerabilities with moderate to high accuracy.

The results also show that inserting LLM-generated honeypots led to a substantial drop in detection performance across all models. For instance, accuracy for GPT-4o-mini dropped from 0.636 to 0.027, and for GPT-4o from 0.668 to 0.149. Similarly, LLaMA's accuracy decreased from baseline levels, 0.678 to 0.205 after honeypot insertion.

These results confirm that LLM-generated honeypots are highly effective at misleading models, often causing them to misidentify artificial vulnerabilities while overlooking real ones. Notably, GPT-4o-mini was the most impacted, with detection accuracy dropping by over 60 % points, while GPT-4o and LLaMA retained

Table 5.6 Accuracy comparison of vulnerability leakage

Model	**Original Code**	**Honeypot Injected Code**
GPT-4o-mini	0.636	0.027
GPT-4o	0.668	0.149
LLaMa	0.678	0.205

slightly higher performance under honeypot conditions.

5.2.2 Functionality Retention of LLM-Generated Honeypots

To evaluate how well LLMs understand the functional behavior of honeypotted code, we conducted a functionality understanding assessment. Each model was given both the original and honeypotted versions of a code snippet and asked whether the two versions perform the same task. The responses were rated on a scale from 0 (completely incorrect) to 5 (fully accurate functional equivalence).

The results, presented in Table 5.7, show that all three models generally maintained a strong understanding of code functionality despite the presence of honeypots. GPT-4o-mini and GPT-4o achieved average scores of 4.13 and 4.10, respectively. LLaMA performed the best, with an average score of 4.61. These scores indicate that the added honeypots did not significantly impair the LLMs' ability to comprehend what the code is doing.

This suggests that while honeypots reduced vulnerability leakage, they did not cause confusion about the overall functionality for most LLMs.

Table 5.7 Functionality retention scores for honeypotted codes

Model	Mean Functionality Score
GPT-4o-mini	4.13
GPT-4o	4.10
LLaMa	4.61

These scores indicate that the LLM-generated honeypotted versions largely retained the intended behavior of the original code.

5.2.3 Cyclomatic Complexity and Effectiveness of Honeypot Obfuscation

To examine whether the structural complexity of code influences the effectiveness of honeypot-based defenses, we conducted a cyclomatic complexity analysis across all tested models. Cyclomatic complexity serves as a proxy for logical branching and control flow intricacy, both of which can affect an LLM's ability to parse and reason about code behavior. For this analysis, we calculated the average cyclomatic complexity of code samples that were successfully obfuscated by honeypots (i.e., the model failed to detect real vulnerabilities, Accuracy = 0) and those where the model remained accurate (Accuracy = 1).

Table 5.8 Cyclomatic Complexity means according to accuracy

Accuracy	GPT-4o-mini	GPT-4o	LLaMa-4
accuracy 0	6.19	6.14	6.32
accuracy 1	4.71	6.35	5.72

The analysis reveals a compelling model-dependent relationship between code complexity and LLM vulnerability detection performance. For GPT-4o-mini and LLaMA, there is a clear pattern: the average cyclomatic complexity of samples where honeypots succeeded is notably higher than that of correctly identified cases. Specifically, GPT-4o-mini exhibits a mean complexity of 6.19 for failed detections and 4.71 for successful ones, while LLaMA follows a similar trend (6.32 vs. 5.72). These results suggest that higher structural complexity may play a role in amplifying the distracting effects of honeypots, potentially overwhelming the LLM's internal reasoning pathways or reducing its confidence in identifying genuine vulnerabilities.

In contrast, GPT-4o shows no such sensitivity. The difference in average complexity between misclassified and correctly classified samples is minimal (6.14 vs. 6.35), implying that this model maintains a stable detection performance regardless of the underlying control flow intricacy. This robustness may be attributed to stronger internal representations, enhanced attention mechanisms, or improved pretraining on complex code. The implication is important: while complexity-boosting obfuscation strategies may work against smaller or less robust models, they are less effective when facing top-tier LLMs like GPT-4o.

To better visualize these dynamics, we present boxplots (Figure 5.1, Figure 5.2, and Figure 5.3) illustrating the spread and distribution of cyclomatic complexity numbers (CCN) across both detection categories (still leaked: accuracy=1; not leaked:

accuracy=0) for each model. These visualizations reinforce the trends observed in the numerical averages. For both GPT-4o-mini and LLaMA, the boxplots show a skew toward higher complexity values in the failed detection group, with wider interquartile ranges and more upper outliers. GPT-4o's boxplot, in contrast, shows a relatively balanced distribution across both groups, supporting the hypothesis that its vulnerability detection is less sensitive to structural obfuscation.

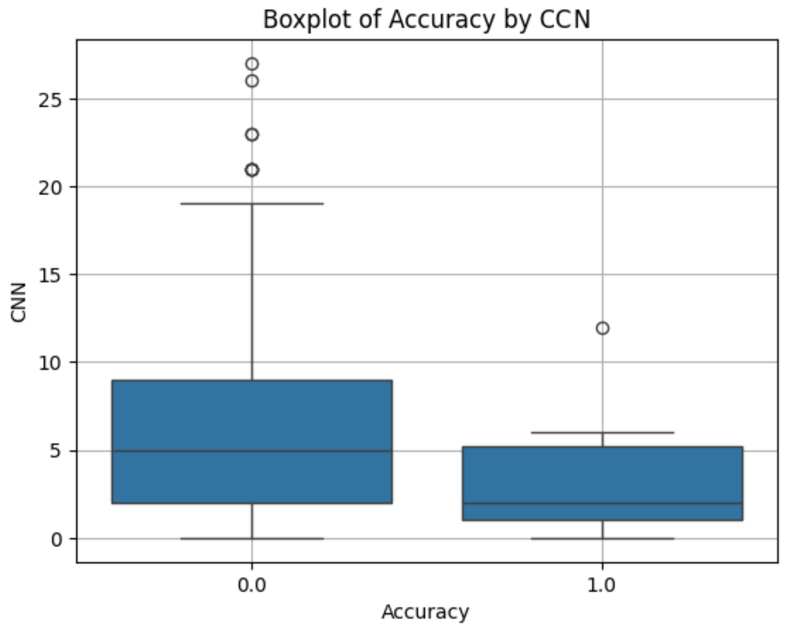

Figure 5.1 Boxplot for ChatGPT-4o-mini

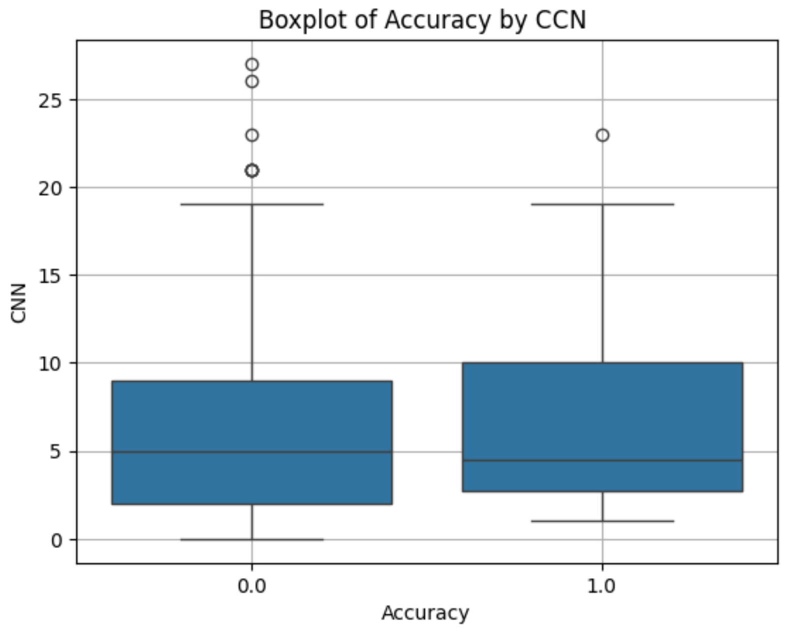

Figure 5.2 Boxplot for ChatGPT-4o

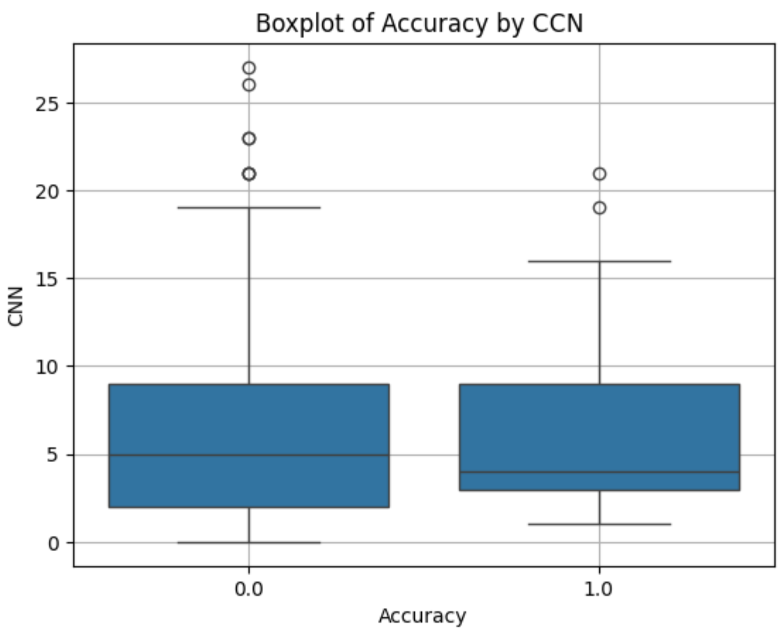

Figure 5.3 Boxplot for LLaMa-4

Collectively, this analysis indicates that cyclomatic complexity can be an effective factor in misleading certain LLMs, particularly when combined with honeypot injection. However, the results also caution against one-size-fits-all defenses: LLMs vary in their sensitivity to control-flow intricacy, and defenses effective against one model may be less effective on another. These findings motivate further work into model-specific obfuscation tuning and suggest that complexity-aware defenses should be strategically deployed depending on the target LLM's architecture and capabilities.

5.2.4 Evaluation Of Technique Combinations with Honeypots

To determine which technique combinations most effectively reduce vulnerability leakage, we evaluated the performance of three LLMs on eight manually crafted honeypot variants. Each variant applies our core honeypots with a specific set of complementary techniques, such as misleading comments, identifier changes, or control flow alterations (see Section 4.3.1).

Table5.9 reports the mean vulnerability leakage accuracy across 400 samples for each combination. The results are interpreted relative to the baseline established by LLM-generated honeypots (Table5.6).

For GPT-4o-mini, the combinations Hp + Misleading Comment and Hp + Complex-

Table 5.9 Accuracy of vulnerability leakage for technical combinations with honeypot

Model	GPT-4o-mini	GPT-4o	LLaMa
Hp + Misleading Comment	0.029	**0.143**	0.112
Hp + Identifier Renaming	0.035	0.212	0.162
Hp + Control Flow Obf	0.032	0.21	0.194
Hp + Data Flow Obf	0.043	0.170	0.212
Hp + Cyclomatic Complexity Increase	0.029	0.154	0.146
Hp + Cyclomatic Complexity Increase + Misleading Comment	**0.027**	0.176	**0.096**
Hp + Misleading Comment + Control Flow Obf	0.032	0.152	0.162
Hp + Cyclomatic Complexity Increase + Misleading Comment + Control Flow Obf	0.045	0.172	0.148

ity + Comment reduced leakage to 0.028 and 0.030 respectively, nearly matching the LLM-generated benchmark of 0.0266. In contrast, other combinations—such as Hp + Data Flow Obf or Hp + Control Flow—were less effective for this model, yielding higher leakage.

For ChatGPT-4o, the addition of honeypots significantly reduced vulnerability leakage across all combinations. Among these, the most effective techniques were relatively simple: pairing the honeypot with misleading comments (0.143), cyclomatic complexity increase (0.154), and misleading comments combined with control flow obfuscation (0.152). These combinations achieved leakage rates comparable to the LLM-generated honeypots (0.1489), indicating that well-targeted distractions—like misleading narrative cues or slight complexity boosts—can effectively impair the model's ability to extract vulnerabilities.

In the case of LLaMA, we observed a similar trend. The LLM-generated honeypot resulted in a leakage score of 0.2048, but multiple combinations achieved even better results. Notably, honeypot with misleading comments (0.112) and honeypot with cyclomatic complexity increase plus misleading comments (0.096) led to the greatest reduction in leakage. These findings suggest that LLaMA is particularly susceptible to semantically disruptive elements like contradictory or misleading annotations, especially when combined with moderate structural changes.

These findings highlight that selective combinations of techniques, particularly those involving misleading comments or added structural complexity, can approximate the effectiveness of LLM-generated honeypots for some models. This enables a more controlled and interpretable design of defensive code transformations.

5.2.5 Functionality Retention of Technique Combinations with Honeypots

To determine whether combining honeypots with additional obfuscation techniques affects LLMs' understanding of program behavior, we conducted the functionality assessment that was described earlier in Section 4.4. Each LLM was asked whether an obfuscated and honeypotted version of a code snippet performed the same task as its original version, and responses were scored on a 0–5 scale for the accuracy of functional understanding.

Table 5.10 Functionality retention scores for honeypotted codes with each technical combination

Model	GPT-4o-mini	GPT-4o	LLaMa-4
Hp + Misleading Comment	4.75	4.56	4.96
Hp + Identifier Renaming	4.44	4.36	4.81
Hp + Control Flow Obf	4.22	4.29	4.60
Hp + Data Flow Obf	4.27	4.31	4.62
Hp + Cyclomatic Complexity Increase	4.28	4.24	4.73
Hp + Cyclomatic Complexity Increase + Mis-leading Comment	4.56	4.53	4.75
Hp + Misleading Comment + Control Flow Obf	4.42	4.34	4.64
Hp + Cyclomatic Complexity Increase + Mis-leading Comment + Control Flow Obf	4.32	4.31	4.79

As shown in Table 5.10, functionality retention remained consistently high across all combinations and models. For GPT-4o-mini, the highest-scoring combination was Hp + Misleading Comment (4.75), followed closely by Hp + Cyclomatic Complexity Increase + Misleading Comment (4.56). GPT-4o exhibited similarly strong performance, with top scores for the same combinations (4.56 and 4.53, respectively). LLaMA once again outperformed the other models, achieving near-perfect functionality scores; Hp + Misleading Comment reached 4.96, and most combinations exceeded 4.6.

These results indicate that even when honeypots are combined with more complex transformations such as control flow or data flow obfuscation, LLMs largely retain the ability to correctly interpret the code's functionality. In particular, combinations that include misleading comments or increased cyclomatic complexity appear to be especially effective at preserving functional clarity while still offering privacy protection through reduced vulnerability leakage.

6. DISCUSSION

Our findings reveal both the potential risks and limitations of LLM-based vulnerability detection and the effectiveness of mitigation strategies across two phases of experimentation. While LLMs like GPT-4o-mini, GPT-4o, and LLaMA demonstrate considerable capacity to identify security vulnerabilities in source code, their accuracy is significantly impacted by both traditional obfuscation techniques and the introduction of honeypot vulnerabilities combined with misleading transformations.

In Phase 1, basic obfuscation methods such as comment, string, and identifier obfuscation showed only a moderate effect on reducing information leakage. These techniques slightly lowered detection accuracy, suggesting that LLMs still rely on the structural patterns of code rather than just textual cues. More sophisticated obfuscation techniques, such as control flow and data flow obfuscation, were more effective in hindering vulnerability detection. By altering how code executes while keeping its functionality intact, these methods disrupted the model's ability to recognize known vulnerability patterns. However, the most effective technique in reducing vulnerability detection was dead code insertion, which significantly decreased accuracy. This likely resulted from the introduction of misleading logic that confused the model's pattern recognition.

Encoding the entire code in Base64 proved to be highly disruptive to vulnerability detection, as expected. This aligns with the known limitations of LLMs (Wei, Haghtalab & Steinhardt, 2023) when processing non-natural or encoded input formats, which inhibit their ability to parse and semantically interpret code. However, Base64 encoding also severely impacted functionality retention, making it impractical for real-world use. Similarly, LLM-based obfuscation showed mixed results—sometimes improving security but occasionally making vulnerabilities more recognizable. This suggests that LLMs may have an inherent understanding of certain obfuscation patterns, especially those generated by other models.

Beyond detection accuracy, functionality retention was a critical factor in Phase 1. While simple transformations like comment and identifier obfuscation preserved the

LLM's ability to extract functionality almost entirely, techniques such as encoding and deep structural obfuscations degraded semantic understanding. These results emphasize the delicate trade-off between obfuscation for security and maintaining the code's original behavior.

Building on these insights, Phase 2 introduced honeypot vulnerabilities combined with multiple misleading strategies including misleading comments, identifier renaming, control flow obfuscation, data flow obfuscation which showed successful results in phase 1, and intentional increases in cyclomatic complexity to systematically mislead LLMs. Honeypots were highly effective across all tested models. By embedding decoy vulnerabilities that mimic real-world patterns, we succeeded in diverting the models' attention away from actual weaknesses. The effectiveness of these strategies was particularly notable for GPT-4o-mini and LLaMA, where accuracy dropped by over 60 percentage points. However, GPT-4o demonstrated greater resistance to honeypots, suggesting that larger and more advanced models may require more sophisticated or tailored deception strategies.

Among all Phase 2 techniques, misleading comments placed around honeypots emerged as the most consistently successful approach. This technique leverages LLMs' sensitivity to natural language context, subtly steering attention without altering code behavior. Similarly, pairing honeypots with increased cyclomatic complexity proved highly effective against models like GPT-4o-mini and LLaMA. These findings suggest that combining semantic (comment-based) and structural (complexity-based) distractions creates more convincing decoys.

The success of these approaches likely stems from how LLMs parse and prioritize code context. Misleading comments exploit the models' weighting of natural language cues, guiding them toward irrelevant sections of code. Honeypots introduce familiar patterns of vulnerability, making it harder for models to distinguish between noise and signal. Increasing cyclomatic complexity further disrupts reasoning, potentially overwhelming internal representations or reducing model confidence in identifying genuine flaws.

Functionality retention remained strong throughout Phase 2. Even for structurally disruptive methods like control flow modification, most honeypotted samples preserved semantic fidelity as assessed through LLM-generated functional explanations. This indicates that our deceptive techniques mislead models without compromising intended program behavior. Nonetheless, since functionality evaluation relied on LLM-based summaries rather than dynamic testing, subtle behavioral changes may have gone undetected.

6.1 Limitations and Ethical Implications

Despite promising results, several limitations should be acknowledged. Our experiments relied exclusively on static LLM analysis without ground-truth execution results. Functionality assessments based on model-generated explanations, while scalable, may miss edge cases or runtime-specific behaviors. Future work incorporating dynamic or hybrid analysis would strengthen the validity of these findings.

Moreover, the models evaluated in this study—GPT-4o, GPT-4o-mini, and LLaMA—are rapidly evolving. Newer architectures or fine-tuned systems may exhibit different responses to obfuscation and honeypot techniques. We also did not explore jailbreak prompts, adversarial training, or interactive use cases, which could further influence LLM behavior in real-world environments.

There are also important ethical considerations. While developers may adopt honeypots and obfuscations to guard sensitive code against unintended disclosure through LLMs, there is a dual-use risk. Malicious actors could use similar techniques to hide vulnerabilities from AI-assisted auditing tools. As LLMs become increasingly integrated into software development workflows, it is critical for tool developers and security researchers to address these potential blind spots through transparency, robust auditing practices, and the development of deception-resilient models.

7. CONCLUSION

As LLMs become increasingly integrated into secure software development and vulnerability auditing workflows, understanding both their capabilities and their blind spots is critical. This thesis provides a comprehensive examination of how LLMs can inadvertently expose vulnerabilities in source code and how such leakage can be mitigated through two complementary defense strategies: traditional obfuscation and deception-based honeypot techniques.

In Phase 1, we demonstrated that LLMs are capable of identifying security flaws in source code without explicit prompting, highlighting a novel form of information leakage. By systematically applying a range of obfuscation techniques—including comment removal, string encoding, identifier renaming, control and data flow transformations, and dead code insertion—to a diverse dataset of 400 vulnerable code snippets spanning 51 CWE types, we evaluated which methods most effectively hinder the LLM's ability to infer security weaknesses. Our results showed that dead code insertion and advanced control flow obfuscation were among the most effective defenses. However, extreme techniques such as full encoding, while highly disruptive to vulnerability detection, severely impaired the LLM's ability to extract and understand functionality, rendering them impractical for real-world use. This highlights a crucial trade-off: obfuscation can help reduce unintended vulnerability exposure, but overly aggressive transformations risk compromising the legitimate utility of LLMs for development tasks such as refactoring and explanation.

Building on these insights, Phase 2 introduced a novel defense paradigm by embedding honeypot vulnerabilities—carefully crafted decoy weaknesses—into source code to mislead LLMs and distract their attention from genuine flaws. When combined with additional misleading strategies which demonstrated their success in Phase 1, such as misleading comments, control flow obfuscation, and deliberate increases in cyclomatic complexity, honeypots significantly reduced vulnerability detection accuracy across all tested models (GPT-4o, GPT-4o-mini, and LLaMA). Among these strategies, misleading comments emerged as particularly robust and model-agnostic, exploiting the language-sensitive reasoning of LLMs with minimal code modification.

Our findings also underscore the role of code complexity as a modulating factor: very simple code may resist deception, while more complex structures can amplify the effectiveness of misleading signals.

Importantly, both phases of this study emphasized functionality preservation. While obfuscation and honeypot techniques altered code structure and semantics to varying degrees, functionality retention evaluations suggest that the transformed code maintained its intended behavior in most cases. Nonetheless, we acknowledge that our functionality assessments were based on LLM-generated summaries, and further validation through dynamic testing remains an important area for future work.

These findings offer valuable insights for developers seeking to protect sensitive code from unintended disclosure when using AI-assisted tools. Lightweight obfuscation and strategically placed honeypots could be incorporated into integrated development environments (IDEs) or continuous integration (CI) pipelines as pre-processing steps, safeguarding software against over-disclosure of security weaknesses to third-party LLM systems.

In terms of scientific contribution, this thesis introduces a high-diversity dataset spanning 51 CWE types, evaluates LLM behavior across multiple obfuscation and deception techniques, and provides empirical evidence on the effectiveness of these methods in reducing vulnerability leakage. By highlighting both the strengths and weaknesses of current LLMs, this work offers actionable insights for developers, security practitioners, and future model designers seeking to balance usability, security, and resilience in AI-supported software development.

BIBLIOGRAPHY

Adam, C., Bulut, M. F., Sow, D., Ocepek, S., Bedell, C., & Ngweta, L. (2022). Attack techniques and threat identification for vulnerabilities.

Bakhshandeh, A., Keramatfar, A., Norouzi, A., & Chekidehkhoun, M. (2023). Using chatgpt as a static application security testing tool.

Bhandari, G., Naseer, A., & Moonen, L. (2021). Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. (pp. 30–39).

Boi, B., Esposito, C., & Lee, S. (2024). Smart contract vulnerability detection: The role of large language model (llm). *24*(2), 19–29.

Brama, H., Dery, L., & Grinshpoun, T. (2022). Evaluation of neural networks defenses and attacks using ndcg and reciprocal rank metrics. *Int. J. Inf. Secur.*, *22*(2), 525–540.

Carlini, N., Tramèr, F., Wallace, E., Jagielski, M., Herbert-Voss, A., Lee, K., Roberts, A., Brown, T., Song, D., Erlingsson, Ú., Oprea, A., & Raffel, C. (2021). Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, (pp. 2633–2650). USENIX Association.

Ding, H., Liu, Y., Piao, X., Song, H., & Ji, Z. (2025). Smartguard: An llm-enhanced framework for smart contract vulnerability detection. *Expert Systems with Applications*, *269*, 126479.

Esposito, M., Janes, A., Kilamo, T., & Lenarduzzi, V. (2024). Early career developers' perceptions of code understandability. a study of complexity metrics.

Geiping, J., Stein, A., Shu, M., Saifullah, K., Wen, Y., & Goldstein, T. (2024). Coercing llms to do and reveal (almost) anything.

Guo, Y., Patsakis, C., Hu, Q., Tang, Q., & Casino, F. (2024). Outside the comfort zone: Analysing llm capabilities in software vulnerability detection. In Garcia-Alfaro, J., Kozik, R., Choraś, M., & Katsikas, S. (Eds.), *Computer Security – ESORICS 2024*, (pp. 271–289)., Cham. Springer Nature Switzerland.

Lakshmanan, R. (2025). Zero-click ai vulnerability exposes microsoft 365 copilot data without user interaction. *The Hacker News*.

Lanka, P., Gupta, K., & Varol, C. (2024). Intelligent threat detection—ai-driven analysis of honeypot data to counter cyber threats. *Electronics*, *13*(13).

Li, Q., Wen, J., & Jin, H. (2024). Governing open vocabulary data leaks using an edge llm through programming by example. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, *8*(4).

Li, Y., Li, X., Wu, H., Xu, M., Zhang, Y., Cheng, X., Xu, F., & Zhong, S. (2025). Everything you wanted to know about llm-based vulnerability detection but were afraid to ask.

Li, Z., Dutta, S., & Naik, M. (2025). IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In *The Thirteenth International Conference on Learning Representations*.

Lin, Y., Wan, C., Fang, Y., & Gu, X. (2024). Codecipher: Learning to obfuscate source code against llms.

Lu, G., Ju, X., Chen, X., Pei, W., & Cai, Z. (2024). Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning.

Journal of Systems and Software, *212*, 112031.

Mai, P., Yang, Y., Yan, R., Ye, R., & Pang, Y. (2024). Confusionprompt: Practical private inference for online large language models.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320.

McCabe Jr., T. (2008). Software quality metrics to identify risk. Department of Homeland Security Software Assurance Working Group. Presented on January 31, 2008; Last edited for content in Nov. 2008.

MetaAI (2025). Meta-llama LLaMA-4-Scout-17B-16E-Instruct. `https://huggingface.co/meta-llama/Llama-4-Scout-17B-16E-Instruct`. Hugging Face. Accessed: June 17, 2025.

MITRE (2024). 2024 cwe top 25 most dangerous software weaknesses. `https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html`. Common Weakness Enumeration (CWE). [Online]. Accessed: July 12, 2025.

Mohamed, A., Assi, M., & Guizani, M. (2025). The impact of llm-assistants on software developer productivity: A systematic literature review.

Mohseni, S., Mohammadi, S., Tilwani, D., Saxena, Y., Ndawula, G., Vema, S., Raff, E., & Gaur, M. (2025). Can llms obfuscate code? a systematic analysis of large language models into assembly code obfuscation. *Proceedings of the AAAI Conference on Artificial Intelligence*, *39*, 24893–24901.

Montalbano, E. (2024). Google ai platform bugs leak proprietary enterprise llms. *Dark Reading*.

Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., & Myers, B. (2024). Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Niu, Z., Zhong, G., & Yu, H. (2021). A review on the attention mechanism of deep learning. *Neurocomputing*, *452*, 48–62.

Open Source Security Foundation (OpenSSF) (2025). The best practices for oss developers. `https://github.com/ossf/wg-best-practices-os-developers`. [Online]. Accessed: July 12, 2025.

OpenAI (2022). Chatgpt. `https://platform.openai.com/docs/api-reference/streaming`. OpenAI API Reference. [Online].

Otal, H. T. & Canbaz, M. A. (2024). Llm honeypot: Leveraging large language models as advanced interactive honeypot systems. In *2024 IEEE Conference on Communications and Network Security (CNS)*, (pp. 1–6). IEEE.

OWASP (2024). Data leakage - owasp top 10 for large language model applications. `https://owasp.org/www-project-top-10-for-large-language-model-applications/Archive/0_1_vulns/Data_Leakage.html`.

Paganini, P. (2023). Samsung data leak via chatgpt — employees shared confidential info.

Pape, D., Mavali, S., Eisenhofer, T., & Schönherr, L. (2025). Prompt obfuscation for large language models.

Radev, D. R., Qi, H., Wu, H., & Fan, W. (2002). Evaluating web-based question answering systems. In González Rodríguez, M. & Suarez Araujo, C. P. (Eds.), *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC'02)*, Las Palmas, Canary Islands - Spain. European Language Resources Association (ELRA).

Rajeev, M., Ramamurthy, R., Trivedi, P., Yadav, V., Bamgbose, O., Madhusudan, S. T., Zou, J., & Rajani, N. (2025). Cats confuse reasoning llm: Query agnostic adversarial triggers for reasoning models.

Sallou, J., Durieux, T., & Panichella, A. (2024). Breaking the silence: the threats of using llms in software engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, (pp. 102–106)., New York, NY, USA. Association for Computing Machinery.

Sheng, Z., Chen, Z., Gu, S., Huang, H., Gu, G., & Huang, J. (2025). Llms in software security: A survey of vulnerability detection techniques and insights.

Singh, S. C. & Ohri, N. (2025). India's finance ministry asks employees to avoid ai tools like chatgpt, deepseek. *Reuters*.

Sladić, M., Valeros, V., Catania, C., & Garcia, S. (2024). Llm in the shell: Generative honeypots. In *2024 IEEE European Symposium on Security and Privacy Workshops (EuroSamp;amp;PW)*, (pp. 430–435). IEEE.

Spitzner, L. (2002). *Honeypots: Tracking Hackers*. USA: Addison-Wesley Longman Publishing Co., Inc.

Steenhoek, B., Rahman, M. M., Roy, M. K., Alam, M. S., Tong, H., Das, S., Barr, E. T., & Le, W. (2025). To err is machine: Vulnerability detection challenges llm reasoning.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., & Garnett, R. (Eds.), *Advances in Neural Information Processing Systems*, volume 30, Long Beach, CA, USA. Curran Associates, Inc.

Voorhees, E. M. (1999). The trec-8 question answering track report. Technical Report NIST Special Publication 500-246, National Institute of Standards and Technology, Gaithersburg, MD, USA.

Wei, A., Haghtalab, N., & Steinhardt, J. (2023). Jailbroken: How does llm safety training fail?

Wodecki, B. (2023). Jpmorgan joins other companies in banning chatgpt. *AI Business*.

Yan, F. & Li, M. (2021). Towards generating summaries for lexically confusing code through code erosion. In Zhou, Z.-H. (Ed.), *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, (pp. 3721–3727). International Joint Conferences on Artificial Intelligence Organization. Main Track.

Yıldırım, R., Aydın, K., & Çetin, O. (2024). Evaluating the impact of conventional code analysis against large language models in api vulnerability detection. In *Proceedings of the 2024 European Interdisciplinary Cybersecurity Conference*, EICC '24, (pp. 57–64)., New York, NY, USA. Association for Computing Machinery.

Zhao, X., Li, L., & Wang, Y.-X. (2022). Provably confidential language modelling. In Carpuat, M., de Marneffe, M.-C., & Meza Ruiz, I. V. (Eds.), *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, (pp. 943–955)., Seattle, United States. Association for Computational Linguistics.

Zhou, X., Weyssow, M., Widyasari, R., Zhang, T., He, J., Lyu, Y., Chang, J., Zhang, B., Huang, D., & Lo, D. (2025). Lessleak-bench: A first investigation of data leakage in llms across 83 software engineering benchmarks.