Chapter 5
Are Textual Prompts in Large Language Models Sufficient for Vulnerability Detection?

Puya Pakshad
https://orcid.org/0000-0003-2834-2769
Illinois Institute of Technology, USA

Sajad Aqanasiri
https://orcid.org/0009-0007-8169-3476
Shahid Beheshti University, Iran

ABSTRACT

Large Language Models (LLMs) have gained traction in domains from software development to cybersecurity, particularly for detecting vulnerabilities in program source code. Their ability to analyze large codebases and identify security weaknesses makes them valuable in software security analysis. However, their effectiveness declines in the absence of intermediate representations such as Abstract Syntax Trees (AST), Control Flow Graphs (CFG), and Data Flow Graphs (DFG), or even tokenized forms of code. In this research study, we assess the performance of LLMs in detecting vulnerabilities directly from raw source code, without structural representations. By designing context-specific prompts, we aim to enhance the model's understanding of code semantics. Our findings show that LLMs can partially identify vulnerabilities from raw code alone, reaching up to 43% accuracy. This indicate both the potential and current limitations of prompt-based LLMs for static vulnerability detection.

DOI: 10.4018/979-8-3373-4862-9.ch005

INTRODUCTION

Vulnerability detection serves as a foundational step in securing modern software, which aims to identify flaws in program source code and software behavior to prevent potential exploitation by malicious adversaries (Ghaffarian & Shahriari, 2017). Recent industry evidence indicates that over 60% of data breaches are caused by unpatched software vulnerabilities, leading to significant business consequences such as data loss, reputational harm, and supply chain disruptions (Ponemon Institute, 2025; Palo Alto Networks Unit 42, 2025). For example, a 2024 analysis by CrowdStrike shows a high-impact software supply chain attack, where an exploited vulnerability in a vendor's source code led to widespread breaches across multiple partner organizations, resulting in multi-million-dollar financial losses and operational disruptions (CrowdStrike, 2024). From a technical perspective, the FBI has reported that over 30% of security incidents in recent years were linked to storage-related security vulnerabilities that allowed unauthorized access to sensitive organizational data (Karie, Sahri, & Yang, 2022). Moreover, global cybercrime damages are projected to reach $10.5 trillion annually by 2025, which represent severe risks to large-size businesses (Gavou, Iliya, & Ihuoma, 2024). These security vulnerabilities often originate in the software development phase, where insecure coding practices by developers introduce exploitable flaws (Hughes & Turner, 2023). These pieces of evidence demonstrate how software vulnerabilities, particularly those left unpatched, serve as primary attack vector with direct and measurable impacts on organizational security and business continuity, which emphasize the critical importance of early vulnerability detection in program source code (IBM Security, 2024).

Numerous techniques have been proposed for software vulnerability detection, typically classified into static, dynamic, and hybrid analysis techniques (Shereen et al., 2024). Traditional methods, such as static code analyzers and runtime behavioral monitors, have laid the groundwork for modern vulnerability detection tools. Over time, these traditional approaches have increasingly been integrated with artificial intelligence, particularly machine learning and deep learning models, which results in higher precision and lower false positive rates (Sheng et al., 2025).

Recently, large language models (LLMs) have emerged as powerful tools in the smart detection of software vulnerabilities, which leverage vast contextual understanding to identify security flaws with high semantic awareness (Shao & Ding, 2024; Khare et al., 2023; Mao et al., 2024). Modern software vulnerability detection technqiues frequently are based on intermediate representations of source code to better convey its structural and semantic properties to LLMs. These include representations such as the Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG), and Code Property Graph (CPG), which are often fused into Composite Joint Graphs (CJGs) to provide rich input embeddings (Shao & Ding,