Large Language Models in the Software Supply Chain: Challenges and Opportunities

Giacomo Benedetti[1], Luca Caviglione[1], Carmela Comito[2], Daniela Gallo[2,3], Alberto Falcone[2], Massimo Guarascio[2], Angelica Liguori[2,*], Giuseppe Manco[2], Francesco Sergio Pisani[2], Ettore Ritacco[4] and Antonino Rullo[2]

[1]*Institute for Applied Mathematics and Information Technologies, Via de Marini, 6, Genova, 16149, Italy*
[2]*Institute for High Performance Computing and Networking, Via P. Bucci, 8-9/C, Rende, 87036, Italy*
[3]*University of Salento, Piazza Tancredi, 7, Lecce, 73100, Italy*
[4]*University of Udine, Via Palladio, 8, Udine, 33100, Italy*

Abstract
Large Language Models (LLMs) are increasingly integrated within the software development lifecycle, for instance, to generate code and documentation or to support debugging. As a result, LLMs are expected to become a relevant tool for improving the security posture of modern software supply chains, especially for detecting weaknesses or vulnerabilities, patching code in an automated manner, and explaining runtime behaviors. While LLMs offer substantial productivity benefits, their widespread adoption introduces new security risks. Adversaries can exploit LLM-generated code to introduce or propagate vulnerabilities, potentially compromising the integrity of critical systems. In this paper, we explore how software security can be enhanced by taking advantage of LLMs. Specifically, we outline key research directions focusing on source code attribution, malware variant generation for robust detection, and the use of LLMs to support secure coding practices. To showcase part of our ongoing research, we briefly assess the explainability capabilities of LLMs and their potential to bridge the gap between code and comprehension in modern development workflows.

Keywords
Software Security, Large Language Models, Code analysis, Generative AI

1. Introduction

Large Language Models (LLMs) have profoundly transformed the way software is developed, maintained, and secured. Their ability to generate human-like code, explain complex logic, and assist with debugging has led to a paradigm shift in modern software engineering [1]. This transformation is especially visible in the emergence of *Vibe Coding*, a concept introduced by OpenAI in early 2025. In essence, it envisions a deeply integrated, real-time, conversational programming experience where developers engage with LLMs as dynamic collaborators. Unlike traditional development approaches, this emerging coding paradigm emphasizes fluid interaction, continuous feedback, and a blending of natural and programming languages to accelerate the software creation process and make it more accessible [2].

Beyond development tasks, LLMs are increasingly embedded in a wide array of systems across different domains. They are used to summarize emails, refine legal or technical documents, assist in customer service, prepare test cases, and even manage infrastructure-as-code in DevOps workflows [3, 4]. Even if the pervasive integration of LLMs into critical systems enhances productivity, it also significantly enlarges the attack surface for potential adversaries. For instance, attackers may exploit LLMs through

Ital-IA 2025: 5th National Conference on Artificial Intelligence, organized by CINI, June 23-24, 2025, Trieste, Italy
*Corresponding author.
giacomorenatobenedetti@cnr.it (G. Benedetti); luca.caviglione@cnr.it (L. Caviglione); carmela.comito@cnr.it (C. Comito); daniela.gallo@cnr.it (D. Gallo); alberto.falcone@cnr.it (A. Falcone); massimo.guarascio@cnr.it (M. Guarascio); angelica.liguori@cnr.it (A. Liguori); giuseppe-manco@cnr.it (G. Manco); francescosergio.pisani@cnr.it (F. S. Pisani); ettore.ritacco@uniud.it (E. Ritacco); antonino.rullo@cnr.it (A. Rullo)
0000-0003-2609-6787 (G. Benedetti); 0000-0001-6466-3354 (L. Caviglione); 0000-0001-9116-4323 (C. Comito); 0009-0009-3245-7738 (D. Gallo); 0000-0002-2660-1432 (A. Falcone); 0000-0001-7711-9833 (M. Guarascio); 0000-0001-9402-7375 (A. Liguori); 0000-0001-9672-3833 (G. Manco); 0000-0003-2922-0835 (F. S. Pisani); 0000-0003-3978-9291 (E. Ritacco); 0000-0002-6030-0027 (A. Rullo)

prompt injection, model manipulation, or by forcing the generation of insecure code patterns, which cannot be detected via conventional countermeasures (see, e.g., [5] and the references therein).

Despite this, LLMs are demonstrating their ability to enhance security across modern software supply chains [6]. To this aim, one of the most pressing concerns lies in the data used for training. If models are trained on code repositories that contain weaknesses or poorly-written code they may inherit and reproduce vulnerable traits. This can result in code suggestions that include unsafe functions, insecure authentication flows, or outdated cryptographic practices. Moreover, data used by LLMs could need some form of policing, for instance, to prevent the uncontrolled use of copyrighted software. Thus, deploying LLMs in production-quality scenarios requires some form of tracking and **attribution**. At the same time, the ability of LLMs of generating "seen" programming prose could be turned into an advantage for **anticipating mutations** in malicious code and prepare suitable datasets to make countermeasures more robust. Several LLMs can assist in identifying vulnerabilities, suggesting patches, explaining insecure logic, and even reasoning about the root causes of security issues [7]. An emerging trend then takes advantage of their natural language capabilities for aiding non-expert developers in understanding and fixing security flaws, bridging the gap between code and comprehension. For instance, LLMs can be used to score vulnerabilities or to **provide explainability** in complex analysis pipelines, e.g., those aimed at reversing code or patching software [8].

This work outlines the main challenges that the ICAR - IMATI research group is facing to advance in software supply chain security via LLMs, within the framework of the two funded projects SERICS (SEcurity and RIghts In the CyberSpace) and WHAM! (Watermarking Hazards and novel perspectives in Adversarial Machine learning). Specifically, it presents our key research ideas on the work to be done to tame the ingestion of source code and leverage LLMs to improve security aspects. It also briefly showcases the explainability gaps to be filled for their use in real settings by showing two simple examples of C prose analyzed through two general-purpose models, i.e., Google Gemini and ChatGPT.

The rest of the paper is structured as follows. Section 2 provides ideas on how to enforce code attribution, Section 3 deals with the generation of software for counteracting malware, and Section 4 outlines some pros and cons of using LLMs for explain code. Lastly, Section 5 concludes the paper.

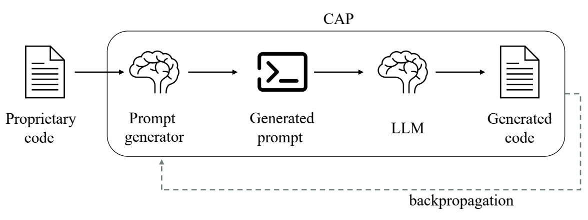
2. Automatic Code Attribution

Advancements in LLMs for improving the development pipeline (e.g., to generate code or support the understanding of complex programs) require specific datasets for both training and fine-tuning phases, which often contain huge volumes of code fragments or complete implementations [9]. Moreover, datasets used to feed AI tools could exploit information or software that have been "scraped" without consent or by violating copyright constraints. In this vein, being able to understand whether specific slices of source code have been ingested to enhance the performances of AI tools for coding tasks has rapidly become a relevant concern. Specifically, we consider prompt-based generative models that produce suitable outputs in response to given input prompts. A key question is whether a given LLM has been trained using proprietary code, i.e., code whose usage is subject to licensing restrictions.

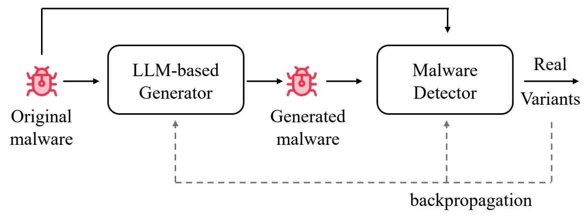
As a possible solution, Membership Inference Attack (MIA) approaches [10] appear to be the most suited. Their goal is to determine whether a specific sample was included in the training set of an AI model. Here, the core idea is that if a generative model produces data resembling existing sources, it likely indicates that the model was exposed to those sources during training. This behavior reflects a known "weakness" in generative models: their tendency to memorize and reproduce training data [11].

In the case of LLMs tailored for code generation tasks, when the model yields an output that closely resembles our target code, this suggests that it may have encountered the original code during training. Existing MIA approaches require that the individual suspecting that an LLM has been trained on its code must manually create targeted prompts to potentially obtain a positive feedback. However, this manual process becomes impractical when applied to large-scale datasets.

To overcome such limitation, our future research is aimed at expanding the basic idea of CAP (Copyright Audit via Prompts generation) [12]. In essence, CAP provides an automated MIA method



(a) CAP framework adapted for code attribution.



(b) AI-Based pipeline for malware variant generation.

Figure 1: LLM-based frameworks designed for our research.

able to automatically generate prompts used to induce a black-box generative model suspected of having been trained on proprietary data to potentially reveal copyright infringement. The objective of our research is to investigate whether this approach can be adapted to our scenario to detect unauthorized usage of proprietary code, as illustrated in Figure 1a. Another possible utilization of CAP is the automatic recognition of code plagued with some form of security hazards. For instance, it can be used to check whether the LLM has been exposed during the training phase to a dataset containing fragments with malicious routines or known vulnerabilities.

3. Defensive Malicious Code Generation

The most recent AI-based detection methods can be used to spot a wide-array of malicious software [13]. Moreover, the increasing availability of samples for training allows to mitigate the impact of obfuscation techniques, which typically impair static analysis tools [14]. Unfortunately, threat actors are starting to deploy mechanisms to reduce the effectiveness of countermeasures taking advantage of AI. A major approach concerns the creation of variants that slightly differ from a given offensive payload. As a result, the training phase is burdened and a malware variant may share commonalities with benign software, thus leading to several false alarms.

Hence, LLMs could be adopted to anticipate attacks by generating in an automatic manner “mutations” of malicious software to make detector more robust. In this perspective, the adversarial learning framework offers a natural solution to address this issue as it can be employed to simultaneously generate realistic variants of malware (potentially) capable of evading detection systems.

The objective is hence to strengthen the detector by exposing it to diverse, evasive samples that it learns to correctly identify. A possible solution is to use a generative model to synthesize realistic malware variants that preserve the original malicious functionality while exhibiting minimal structural perturbations designed to evade detection. In a Generative Adversarial Network-based setting, the generator could be an LLM model designed to generate such malware variants, while the discriminator, i.e., the malware detector, learns to discriminate between real and generated variants. In a second stage, the generator can be utilized to produce realistic malware variants, which are then used to train the final model responsible for discriminating between benign and malicious samples.

Figure 1b illustrates the pipeline described above, with an emphasis on the learning process of the generator. The generation of malware variants is utilized by the final model to facilitate the extraction of generalizable features. This, in turn, enhances its ability to detect zero-day threats, i.e., previously unseen or emerging malware.

4. LLMs For Explainability

Several works have demonstrated the ability of LLMs to support both human and automated frameworks for software analysis. For instance, they can be used to deobfuscate code [15] or transform malicious scripts into a more suitable form for static analysis [8, 16]. Since improving the security posture of complex software supply chains requires a deep understanding of code, LLMs can be used to identify harmful coding practices or map “weak” implementations against potential vulnerabilities.

```c
1  int processMessageFromSocket(int socket) {
2    int success;
3    char buffer[BUFFER_SIZE];
4    char message[MESSAGE_SIZE];
5    if (getMessage(socket, buffer, BUFFER_SIZE) > 0) {
6      ExMessage *msg = recastBuffer(buffer);
7      int index;
8      for (index = 0; index < msg->msgLength; index++)
            {
9        message[index] = msg->msgBody[index];
10     }
11     message[index] = '\0';
12     success = processMessage(message);
13   }
14   return success;
15 }
```

Listing (1) C fragment affected by CWE-606.

```c
#include <stdio.h>
#include <string.h>
#define MAXLEN 1024

int main() {

  char inputbuf[MAXLEN];
  char pathbuf[MAXLEN];

  read(cfgfile,inputbuf,MAXLEN);
  strcpy(pathbuf,inputbuf);

  return (0);


}
```

Listing (2) C fragment affected by CWE-170.

Figure 2: Fragments of C code used for the case study.

Alas, a longitudinal study on whether pre-trained models can explain security weaknesses plaguing code is still missing. Thus, part of our ongoing research is devoted to assessing the most popular models when handling software containing a Common Weakness Enumeration (CWE). As a preliminary example, we showcase how ChatGPT (GPT-3.5) and Gemini (2.0 Flash and 2.5 Pro) behave when used against C/C++ code. To this end, we selected two code fragments (Figure 2) containing weaknesses. To analyze the code, we provided ChatGPT and Gemini a prompt that asked for the identification of specific CWEs and an explanation of the weaknesses affecting the code, partially borrowed from [16].

Example 1 - CWE 606 The first fragment in Listing 1 contains an instance of CWE-606 (Unchecked Input for Loop Condition) according to the MITRE framework. The function `processMessageFromSocket()` fails to check the length of the received message. Consequently, `msgLength` may exceed the capacity of the `message` array accessed in a `for` loop (see Line 5).

Analyzing the code with LLMs provided the following results. ChatGPT identifies the code as affected by CWE-119 (Improper Restriction of Operations within Memory Buffer Bounds), CWE-121 (Stack-Based Buffer Overflow), and CWE-843 (Access of Resource Using Incompatible Type). Both CWE-119 and CWE-121 relate to CWE-606, even though they outline the issue from different perspectives. Alas, their more general nature does not allow them to pinpoint the part of the code to be fixed precisely. They fail to attribute the weakness to the unchecked loop termination condition in Line 8, where `msg->msgLength` is used without validation. For CWE-843, ChatGPT warns about potentially malformed user input at Line 6, which can cause unexpected behavior. This is a relevant concern, but it is not precise enough to identify the weak part of the code. Instead, Gemini 2.0 Flash identifies CWE-120 (Buffer Copy without Checking Size of Input) and CWE-131 (Incorrect Calculation of Buffer Size). The CWE-120 indicates a "traditional" buffer overflow where a miscalculated message length could lead to stack writing. Alas, MITRE advises using this weakness with caution, as it is too general. Instead, the CWE-131 refers to an incorrect size computation within `malloc` operations. The LLM correctly reasons that "*the loop assumes the message size within the recast buffer will fit within* `MESSAGE_SIZE` *without explicit checks.*". However, despite the sound reasoning, Gemini 2.0 Flash does not point at the unchecked loop condition that should be fixed. Lastly, Gemini 2.5 Pro identifies CWE-121 and CWE-125 (Out-of-bounds Read), both attributing the possible hazard to stack overwriting when `msg->msgLength` exceeds the actual message body size or `MESSAGE_SIZE`. First, it outlines that the problem begins at Line 6, where the (re)cast is done without validation. Then, it correctly hints that the issue will happen at Line 8, where the `for` loop could potentially cause a write beyond the bounds of the `message` array. Even if Gemini 2.5 Pro does not explicitly target CWE-606, its accurate reasoning can guide developers in fixing the code through appropriate bounds checking before executing the loop at Line 8.

Table 1
ChatGPT and Gemini answers on the provided code fragments.

Model	Lst. 1 - CWE ID	Lst. 2 - CWE ID	Lst. 1 - Eval.	Lst. 2 - Eval.
ChatGPT 3.5	119, 121, 843	120, 252, 362, 676	◑	◑
Gemini 2.0 Flash	120, 131	120	○	○
Gemini 2.5 Pro	121, 125	120, 125, 170, 252	◑	●

Example 2 - CWE 170 Listing 2 contains code that reads data from a `cfgfile` into `inputbuf` at Line 10. Then, it copies the input from `inputbuf` to `pathbuf` by using the `strcpy()` function at Line 11. Since `strcpy` stops copying when it finds a NULL terminator, then an attacker can manipulate `cfgfile` to provide a string without proper termination to make the function reading from the stack until an arbitrary NULL terminator is found. The MITRE framework categorizes this weakness in the CWE-170. The code analysis through LLMs provided the following results. ChatGPT identifies CWE-120, CWE-252 (Unchecked Return Value), CWE-362 (Concurrent Execution using Shared Resource with Improper Synchronization), and CWE-676 (Use of Potentially Dangerous Function). The model uses CWE-120 and CWE-676 to describe the potential buffer overflow resulting from the lack of NULL termination rather than relying on CWE-170. Then, it uses CWE-252 to highlight the lack of return value check for the `read` function (see, Line 10). While unchecked return values are generally poor practice, this issue is not security-critical in this specific context. Gemini 2.0 Flash only identifies CWE-120. As with Listing 1, this classification is overly generic and may be triggered merely by the presence of term *buf* in the `inputbuf` and `pathbuf` variables. Gemini 2.5 Pro detects CWE-120, CWE-125, CWE-170, and CWE-252. While CWE-120 and CWE-252 remain generic and somewhat tangential to the main issue, the combination of CWE-125 (Out-of-bounds Read) and CWE-170 (Improper Null Termination) aligns with our ground truth. It precisely captures the core security weakness in the code.

Discussion Table 1 reports all the CWEs identified by the considered LLMs as well as a condensed indicator of their answers. Along the line of [17], we refined and mapped the collected outputs into three categories: correct answer and correct reasoning (●), incorrect answer but correct reasoning (◑), and incorrect answer and incorrect reasoning (○). As shown, Gemini 2.5 Pro exhibits the best performance, as it provides good reasoning without oversimplifying the security problem to be addressed. The behavior of ChatGPT is similar, but it tends to focus on "side" issues for the main weakness. Lastly, Gemini 2.0 Flash appears to be ineffective in explaining the CWEs, as it primarily provides generic and unfocused answers. This behavior can be attributed to its simpler nature, particularly in terms of parameters, compared to ChatGPT and the 2.5 Pro counterparts. As a general remark, this sample investigation and our ongoing research suggest that general-purpose LLMs provide simple analyses of CWEs. We point out that this further reinforces similar findings on the use of commercial LLMs for solving security-oriented tasks [8, 17].

5. Conclusions

In this work, we have presented the primary research directions that the ICAR – IMATI joint group is pursuing to strengthen software supply chain security through the use of LLMs. Specifically, we explored how LLMs can be leveraged to support source code attribution, generate realistic malware variants for robust detection, and assist developers in adopting secure coding practices.

Part of our ongoing research is devoted to consider LLMs as the final stage of a pipeline for de-obfuscating and explaining malicious JavaScript code. We plan to advance in the outlined research directions, mainly to reduce the impact of uncontrolled training/tuning operations and to generate malware mutations to make detection less prone to evasion attacks. Our long-term goals are towards the development of practical methodologies that enable the secure and effective use of LLMs in real-world software engineering scenarios.

Acknowledgments

This work was partially supported by the following projects: SERICS - SEcurity and RIghts in the CyberSpace (PE00000014), and WHAM! - Watermarking Hazards and novel perspectives in Adversarial Machine learning (B53D23013340006).

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] X. Chen, C. Gao, C. Chen, et al., An Empirical Study on Challenges for LLM Application Developers, ACM Transactions on Software Engineering and Methodology (2025).
[2] P. P. Ray, A Review on Vibe Coding: Fundamentals, State-of-the-art, Challenges and Future Directions, TechRxiv (2025).
[3] S. Abedu, A. Abdellatif, E. Shihab, LLM-Based Chatbots for Mining Software Repositories: Challenges and Opportunities, in: 28th International Conference on Evaluation and Assessment in Software Engineering, 2024, pp. 201–210.
[4] C. C. Polisiani, M. Zuppelli, M. C. Calzarossa, et al., Mitigation of Covert Communications in MQTT Topics Through Small Language Models, in: 32nd International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2024, pp. 1–6.
[5] J. Zhang, H. Bu, H. Wen, Y. Liu, et al., When LLMs meet cybersecurity: A Systematic Literature Review, Cybersecurity 8 (2025) 1–41.
[6] L. Williams, G. Benedetti, S. Hamer, et al., Research Directions in Software Supply Chain Security, ACM Transactions on Software Engineering and Methodology (2025).
[7] S. Wang, Y. Zhao, X. Hou, et al., Large Language Model Supply Chain: A Research Agenda, ACM Transactions on Software Engineering and Methodology (2024).
[8] S. L. Mirtaheri, A. Pugliese, N. Movahed, et al., A Comparative Analysis on Using GPT and BERT for Automated Vulnerability Scoring, Intelligent Systems with Applications 26 (2025).
[9] M. Rahman, S. Khatoonabadi, E. Shihab, A Large-scale Class-level Benchmark Dataset for Code Generation with LLMs, arXiv (2025).
[10] H. Hu, Z. Salcic, L. Sun, et al., Membership Inference Attacks on Machine Learning: A Survey, ACM Computing Surveys 54 (2022) 1–37.
[11] Y. Liu, J. Huang, Y. Li, et al., Generative AI Model Privacy: A Survey, Artificial Intelligence Review 58 (2025).
[12] D. Gallo, A. Liguori, E. Ritacco, et al., CAP: Detecting Unauthorized Data Usage in Generative Models via Prompt Generation, arXiv (2024).
[13] A. H. Salem, S. M. Azzam, O. Emam, et al., Advancing Cybersecurity: A Comprehensive Review of AI-driven Detection Techniques, Journal of Big Data 11 (2024) 105.
[14] M. G. Gaber, M. Ahmed, H. Janicke, Malware Detection with Artificial Intelligence: A Systematic Literature Review, ACM Computing Surveys 56 (2024) 1–33.
[15] B. Choi, H. Jin, D. H. Lee, et al., ChatDEOB: An Effective Deobfuscation Method Based on Large Language Model, in: International Conference on Information Security Applications, Springer, 2024, pp. 151–163.
[16] C. Patsakis, F. Casino, N. Lykousas, Assessing LLMs in malicious code deobfuscation of real-world malware campaigns, Expert Systems with Applications 256 (2024) 124912.
[17] S. Ullah, M. Han, S. Pujar, et al., LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks, in: 2024 IEEE Symposium on Security and Privacy, IEEE, 2024, pp. 862–880.