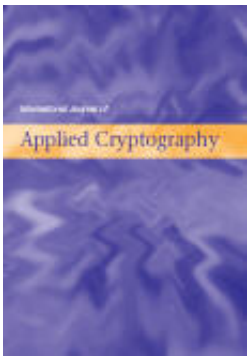
INDERSCIENCE PUBLISHERS
Linking academia, business and industry through research



International Journal of Applied Cryptography

ISSN online: 1753-0571 - ISSN print: 1753-0563
https://www.inderscience.com/ijact

Large language models for vulnerability detection: a multi-use case comparative study

Vasileios Kouliaridis, Georgios Karopoulos, Georgios Kambourakis

DOI: 10.1504/IJACT.2026.10079612

Article History:

Received:	13 August 2025
Last revised:	05 February 2026
Accepted:	07 March 2026
Published online:	07 July 2026

Large language models for vulnerability detection: a multi-use case comparative study

Vasileios Kouliaridis and Georgios Karopoulos*

European Commission,
Joint Research Centre (JRC),
21027, Ispra, Italy
Email: vasileios.kouliaridis@ec.europa.eu
Email: georgios.karopoulos@ec.europa.eu
*Corresponding author

Georgios Kambourakis

Department of Information and Communication Systems Engineering,
University of the Aegean,
Karlovasi, 83200, Samos, Greece
Email: gkamb@aegean.gr

Abstract: Large language models (LLMs) have shown promising capabilities in assisting researchers and developers in different fields of cybersecurity. This work investigates whether 11 state-of-the-art LLMs can be used for source code vulnerability analysis across three different use cases and four publicly available benchmark datasets. More specifically, we examined Android, smart contract and IoT source code, containing vulnerabilities from Open Worldwide Application Security Project (OWASP) Mobile Top 10, Common Weakness Enumeration (CWE) databases, and smart contract related vulnerabilities. Moreover, we explored whether LLMs could detect potentially privacy-invasive actions and if retrieval-augmented generation (RAG) could improve the performance of LLMs in vulnerability detection. Our results reveal that no single LLM is consistently better-performing compared to others across all use cases and datasets, whereas different models are the best performers in different use cases and datasets. Thus, a careful LLM selection is necessary based on the unique characteristics of each use case.

Keywords: large language models; LLMs; vulnerability detection; vulnerability analysis; code analysis; OWASP; mobile security; Android; IoT; smart contracts; retrieval-augmented generation; RAG.

Reference to this paper should be made as follows: Kouliaridis, V., Karopoulos, G. and Kambourakis, G. (2026) 'Large language models for vulnerability detection: a multi-use case comparative study', *Int. J. Applied Cryptography*, Vol. 5, No. 6, pp.1–17.

Biographical notes: Vasileios Kouliaridis received his PhD from the Department of Information and Communication Systems Engineering, University of the Aegean, Greece. He is currently working as a scientific officer for the Joint Research Center of the European Commission. His research interests are in the fields of mobile and network security, privacy and machine learning. He is a frequent author and reviewer in conferences and scientific journals on the above fields.

Georgios Karopoulos is a senior researcher at the Joint Research Centre of the European Commission. He holds a PhD in Computer Network Security from the University of the Aegean, Greece. In the past, he was a Marie Curie Fellow Researcher at the University of Athens, Greece, and an ERCIM fellow at IIT-CNR, Italy. His research interests are in the areas of network security, smart grid security and critical infrastructure protection. He has published and he is a frequent reviewer in conferences and scientific journals in the above areas.

Georgios Kambourakis is a Professor at the University of the Aegean, where he previously served as the Head of the Department and Director of the Info-Sec-Lab. He also worked at the European Commission's Joint Research Centre from 2019 to 2022 and was a Visiting Scholar at George Mason University in 2017. His research focuses on security and privacy in mobile and wireless networks, IoT, VoIP, and DNS, and he has over 180 publications. His work has led to the discovery of more than 30 new CVEs, half of which were of high or critical severity. He is an editor for several journals, including *IEEE Communications Surveys and Tutorials*, and has served on the program committees for numerous international conferences, including ACM CCS, IEEE ICC, IEEE CNS, IFIP SEC, and IFIP DBSec. He has also been involved in numerous national and EU-funded R&D projects

This paper is a revised and expanded version of a paper entitled 'Assessing the effectiveness of LLMs in Android application vulnerability analysis' presented at The 7th International Conference on Attacks and Defenses for Internet-of-Things (ADIoT 2024), Hangzhou, China, 26–27 December 2024.

1 Introduction

Recent studies on software vulnerabilities demonstrate an ever-increasing trend both in the discovery of new vulnerabilities and their exploitation. More specifically, a 61% increase in discovered vulnerabilities compared to the previous year was observed in 2024; according to the same study, the number of exploited vulnerabilities rose by 96% during the same period (Action1, 2025). Furthermore, software supply chain attacks have seen a yearly increase of 742% between 2019 and 2022 (Sonatype, 2023). Even though there are diverse approaches in detecting software vulnerabilities in different domains, such as mobile apps (Kouliaridis et al., 2023), the above surge demonstrates that they are not enough. All these statistics underline the need for more effective defence methods to detect software vulnerabilities early enough and prevent their exploitation.

Lately, large language models (LLMs) have emerged as a disruptive technology that can assist in many different domains, including cybersecurity (Abendroth Dias et al., 2025). In the context of cybersecurity, LLMs have been used for code analysis which can be useful in detecting source code vulnerabilities. The origins of using LLMs for code analysis can be traced back to the proposal of Word2Vec (Church, 2017) in 2013. Word2Vec is a shallow neural network that captures the relationships between words by representing a word as a vector with many dimensions. Its successor, bidirectional encoder representations from transformers (BERT) (Devlin et al., 2019), was introduced in 2018 and it was by design able to be trained bidirectionally. This way, it could better understand the context by learning from both sides of a given text during training.

One of the first models capable of code analysis was GPT-2 (Solaiman et al., 2019), which was trained on a large corpus of source code data. It was fully released in Nov. 2019 and it could understand the structure and context of a given piece of source code, as well as propose the most probable code to follow a given input, thus assisting in code completion. The next version, GPT-3 (Brown et al., 2020), was released in 2020 and showed increased capabilities compared to GPT-2. Being a significantly larger model with 175 B parameters, it could generate human-like text and generate code based on a textual description of the task to be performed. Recent studies have also demonstrated the ability of LLMs to analyse and understand code (Wan et al., 2022; Liu et al., 2023a). However, to the best of our knowledge, the existing literature does not cover an extensive comparison of different models in detecting source code vulnerabilities in different contexts.

The objective of this paper is to fill the above gap in related work by comparing 11 different state-of-the-art and more established LLMs to detect source code vulnerabilities in three distinct use cases: mobile code, smart contracts, and IoT. A previous version of this work by the same authors analysed the efficacy of LLMs in detecting Android code vulnerabilities listed in the Open Worldwide Application Security Project (OWASP) Mobile Top 10 2024 (Kouliaridis et al., 2025). Here, a two-dimension approach is followed:

1 in the horizontal dimension, two LLMs were used to evaluate vulnerability detection across four different datasets, comprising source code for three distinct use cases

2 in the vertical dimension, the mobile device use case was analysed more deeply using nine LLMs on a dataset representing the mobile code use case.

This way, we obtain a comprehensive overview of the efficacy of diverse LLMs in detecting vulnerable code in different settings.

In more detail, we evaluate each model regarding its ability to identify code vulnerabilities in different use cases and datasets containing snippets of vulnerable source code. To assess the LLMs, we use both automated and manual evaluation methods. We also analyse the pros and cons of each model, and discuss the reasons behind their observed performance. Overall, our analysis provides a broad perspective in the ability of open-source and proprietary LLMs in detecting a wide range of heterogeneous software vulnerabilities, including the main smart contract vulnerabilities as well as those mentioned in OWASP Mobile Top 10 and Common Weakness Enumeration (CWE) databases. Our work can provide valuable assistance to software developers, integrators and security evaluators in the aforementioned domains, by automating at least part of their tasks, including secure software development, third-party software assessment, and security audits. On top of the contributions of our previous work (Kouliaridis et al., 2025), this paper advances research in the field in the following ways:

- We analyse the capabilities and performance of 11 open-source and proprietary LLMs, i.e., Quen 2.5, GPT 3.5, GPT 4, GPT 4 Turbo, Llama 2, Llama 3.3, Zephyr Alpha, Zephyr Beta, Nous Hermes Mixtral, MistralOrca, and Code Llama, in identifying software vulnerabilities. We provide detailed information regarding our experiments, such as benchmark datasets and prompts used, to ease reproducibility. Our results provide evidence on the capabilities and limitations of each LLM on different use cases and datasets. Furthermore, we provide a detailed analysis of these results and the limitations of each model.

- To understand if there is any difference in the efficacy of each LLM depending on the type of vulnerabilities, we expand our experiments across three representative use cases: mobile code, smart contracts, and IoT applications. These three scenarios include software vulnerabilities from OWASP Mobile Top 10 lists, CWE databases and smart contract vulnerabilities.
- We further deepen our research by investigating whether the results of different datasets belonging to the same use case are aligned. For this reason, we use two completely different datasets for the mobile code use case. This way, we are able to check the consistency of results and get a complete picture on LLM performance by combining a macroscopic view over all use cases with a deep dive in one of them.

The rest of the paper is organised as follows. The next section presents the existing literature on using LLMs for detecting code vulnerabilities. Section 3 gives a summary of the overall methodology we followed. Section 4 describes the four datasets we employed, Section 5 outlines the types of vulnerabilities contained in the datasets, and Section 6 provides details on the LLMs used. Then, the setup of our experiments and the results are presented in Section 7, whereas a discussion of the results is provided in Section 8. The last section concludes the paper and proposes some lines for future research.

2 Related work

The application of LLMs in cybersecurity has gained considerable interest in recent years, particularly in areas such as vulnerability detection, penetration testing, and security analysis. Several studies, including Al-Hawawreh et al. (2023), Yao et al. (2024), Gupta et al. (2023) and Motlagh et al. (2024), have provided a thorough examination of the current state and potential future uses of these models in cybersecurity. These works investigate the limitations, applications, and potential areas of further research to fully leverage the capabilities of these models to improve cybersecurity. The rest of this section will concentrate on research related to software vulnerability analysis using LLMs, analysing both peer-reviewed and recently published, self-archived works to provide a comprehensive understanding of the topic.

Thapa et al. (2022) assessed the effectiveness of transformer-based LLMs, including BERT, DistilBERT, CodeBERT, GPT-2, and Megatron, in detecting code vulnerabilities. Their evaluation was performed on C/C++ source code snippets from two publicly available datasets, and the results indicated that these models performed well in identifying software vulnerabilities. Notably, the GPT-2 model achieved an F1-score exceeding 95% in all tests, being the top performer in this task. In the field of software engineering, Liu et al. (2023c) explored the use of in-context learning to improve the ability of LLMs to detect software vulnerabilities. This approach involves using code retrieval to identify code snippets similar to the code being scrutinised, and then providing these snippets as input to the model along with the examined code and its analysis. The experimental results showed that this method outperformed the original GPT model, highlighting the adaptability of LLMs to learn from context-specific examples and improve their vulnerability detection capabilities. In our experiments we also observed a notable increase in the detection rate when employing retrieval-augmented generation (RAG) for Code Llama: detection on M2 rose from 30% to 100%, confirming the trend reported by Liu et al. (2023c).

Another line of research has focused on incorporating verification into the vulnerability detection process using LLMs. Purba et al. (2023) performed an empirical study on the use of LLMs for vulnerability assessment in software. They utilised four pre-trained models, including GPT-3.5, Davinci, and CodeGen, to identify vulnerabilities in two labelled datasets, code gadgets and CVEfixes, with static analysis serving as a reference point. The study concentrated on two types of vulnerabilities: SQL injections and buffer overflows. The results revealed that LLMs struggled to detect vulnerabilities accurately, showing high false-positive rates. However, the study suggested that these models could potentially complement and improve traditional static analysis techniques. Our results show that the detection capabilities of different LLM models vary significantly, and currently no LLM is able to detect different kinds of vulnerabilities in a horizontal manner. This aligns with Purba et al. (2023) observation that currently LLMs can play a complementary role to static analysis. In a related study, Sandoval et al. (2023) investigated the safe use of code assistants, focusing on concerns about the potential risks associated with their use. In this study, LLMs were used to generate code, which was then manually reviewed and analysed using static analysis tools. The research provided valuable insights into how developers interact with LLMs, highlighting the importance of user awareness in mitigating security risks related to assisted code generation.

A new framework, namely LLM4VFD, has been proposed in Yang et al. (2025) to identify vulnerabilities in open-source code, addressing the limitations of existing tools such as VulFixMiner, CoLeFunDa, and Vulcurator. This framework utilises LLMs enhanced with chain-of-thought reasoning and in-context learning to improve detection accuracy. Additionally, it provides a detailed analysis and explanation to help security experts' understanding of the decision-making process. The LLM4VFD framework consists of three components that examine code change intentions, development artifacts, and historical vulnerabilities. The authors evaluated their framework using their own dataset called BigVulFix, which comprises 1,689 vulnerability fix commits. The evaluation involved four LLMs: Qwen 2, Llama 3.1, Deepseek-Coder-V2, and CodeBERT, and the results showed significant code improvements, ranging from 68.1% to 145.4%, compared to existing methods.

The work in Guo et al. (2024), evaluates the performance of 12 LLMs in total: six open-source trained for vulnerability detection, three general purpose, and three

fine-tuned general purpose. The above LLMs were tested in six datasets on their ability to classify code into vulnerable or non-vulnerable. The results show that while LLMs can detect vulnerabilities, their performance has significant variations across datasets, and fine-tuning can improve their accuracy. Also, the study raises concerns about the quality of existing datasets, including mislabelling issues, which can impact the training and performance of LLMs. The authors conclude that there is a need for high-quality datasets, improved model generalisation, and explainability to ensure the reliability and robustness of LLMs in software vulnerability detection.

Recent works have explored the potential of LLMs in static binary taint analysis. For instance, Liu et al. (2023d) have demonstrated the utility of LLMs in identifying potential vulnerabilities in binary code. Their approach involves disassembling and decompiling the binary, followed by the application of an LLM to identify security-critical functions and potential dangerous flows. The LLM then synthesises this information to generate a comprehensive vulnerability report for the binary under examination. Similarly, Wang et al. (2023) have proposed a novel vulnerability detection framework, namely DefectHunter, which integrates multiple technologies, including LLMs. The architecture of DefectHunter comprises three primary components: a tool for extracting structural code features, a pre-trained LLM for generating semantic code representations, and a conformer-based mechanism for identifying vulnerabilities from the combined structural and semantic data.

Cheshkov et al. (2023) assessed the effectiveness of ChatGPT and GPT-3 in identifying CWE vulnerabilities in code. Using a custom dataset comprised of Java files from open GitHub repositories, they found that the vulnerability detection capabilities of these models are limited. In a separate study, Noever (2023) investigated the potential of LLMs in detecting software vulnerabilities. They evaluated 129 code samples written in eight different programming languages from various GitHub repositories. The results showed that GPT-4 was able to identify approximately four times more vulnerabilities than static code analysis tools that rely on predefined rules. Furthermore, the authors asked various LLMs, including GPT-3 and GPT-4, to provide fixes for the identified vulnerabilities.

The application of LLMs has also been extended to the detection of vulnerabilities in smart contracts, a critical area of concern in blockchain security. Sun et al. (2024) introduced LLM4Vuln, a novel evaluation framework designed to assess the vulnerability detection capabilities of LLMs in the context of smart contracts. Notably, this framework differs from existing approaches by focusing on the evaluation of each model's vulnerability reasoning capabilities, rather than measuring their performance in detecting vulnerabilities. In a related effort, GPTLens (Hu et al., 2023) is a framework that uses LLMs to detect vulnerabilities in smart contracts. GPTLens adopts a two-stage approach, which differs from traditional one-stage detection methods, with the aim of reducing false positives. In the first stage, the LLM acts as an auditor, generating a long list of potential vulnerabilities for the contract under examination. In the second stage, the LLM assumes the role of a critic, verifying the validity of the vulnerabilities identified in the initial stage. The experimental results show that GPTLens outperforms single-stage vulnerability detection approaches.

3 Methodology

In this section, we describe the methodology we followed for assessing the potential of LLMs in identifying source code vulnerabilities in different use cases. In a high-level view, we followed a two-fold approach, comprising a horizontal and a vertical dimension; this way, we could grasp a better overview of the efficacy of LLMs in detecting vulnerable code in different settings. In the horizontal dimension, we used two models to evaluate LLMs in vulnerability detection across four different datasets, containing source code for three distinct use cases. Then, in the vertical dimension, we delved deeper into the mobile device use case using nine LLMs on one of the mobile source code datasets, representing one of the above use cases. The reason for not running the full set of 11 models on the three other datasets is that it would have required significantly more inference time and cloud-compute costs. With reference to Subsection 7.1, these requirements stem from:

1 the limitations of the *GPT@JRC* platform used for running most of the models

2 the hardware which run the Code Llama model locally

3 the large size of these datasets (LVDAndro $>$ 13k, IR-Fuzz $>$ 4k, IoTvulCode $>$ 66k).

A schematic representation of the methodology we followed is depicted in Figure 1.

The first step was to select diverse use cases that cover key sectors where code vulnerabilities would have a significant impact due to their extensive deployment and potential for widespread exploitation. Specifically, we focused on three key use cases: mobile devices, smart contracts, and IoT. Then, we scoped out appropriate source code datasets that contained software vulnerabilities. Among the existing ones, we selected datasets that are labelled and relatively recent; we also considered the size of the datasets, favouring the largest ones, in order to have more representative results. Note that the LVDAndro mobile code dataset contains older vulnerabilities from the OWASP Mobile Top-10 2016 list, whereas our own-developed dataset (Vulcorpus) covers OWASP Mobile Top-10 vulnerabilities of 2024.

Regarding the LLM models employed, we used two recent models for the horizontal dimension: Qwen 2.5 and Llama 3.3. We selected these models because of their open-source nature, popularity and widespread recognition in the community, which makes them ideal for detecting application vulnerabilities. Additionally, both Qwen 2.5

and Llama 3.3 are relatively new models, which allowed us to assess the capabilities of state-of-the-art LLMs in this domain. For the vertical dimension, we used nine well-established models. In the horizontal evaluation, we test whether a model can discover if there is a vulnerability in a given source code sample, as well as if it can correctly identify which specific vulnerability it is. Apart from these two tests, in the vertical evaluation we additionally request the LLM to explain and provide a valid solution for the vulnerability. In the following sections, we describe the four datasets we used, the vulnerabilities contained in them and the utilised LLMs.

4 Datasets

This section contains a summary of the datasets used in this paper.

4.1 LVDAndro

LVDAndro (Senanayake et al., 2023) is a series of labelled datasets of Android source code with security vulnerabilities. LVDAndro contains code samples from actual Android applications and the labelling was automatically performed by vulnerability scanning tools, such as Mobile Security Framework (MobSF) (https://github.com/MobSF/Mobile-Security-Framework-MobSF) and Quick Android Review Kit (Qark) (https://github.com/linkedin/qark/). Labelling is mainly based on CWE but it also includes information on OWASP Mobile Top-10 (v. 2016). This dataset was created from real Android applications, therefore it contains non-vulnerable code as well. The dataset has ≈21M of total code samples, out of which ≈15M are vulnerable and ≈7M non-vulnerable, containing 23 different CWE IDs. In our case, we utilised the second dataset from the LVDAndro collection, which consists of three datasets, and conducted our experiments on the 13,750 samples labelled according to the OWASP standard.

4.2 Vulcorpus

Considering that LVDAndro included information from an older version of the OWASP Mobile Top-10 vulnerability list (v. 2016), we also used Vulcorpus (Kouliaridis et al., 2025; Vulcorpus, 2024), which contains Android code samples for each of the OWASP Mobile Top-10 vulnerabilities of 2024. The dataset comprises 100 code samples (ten samples for each vulnerability) that are written in Java, utilising common insecure coding practices, e.g., weak authentication mechanisms or not filtering input/objects. Each sample exhibits maximum two interrelated vulnerabilities, while one or two of these samples per vulnerability category are obfuscated manually using the well-known rename technique - a method that renames variables, functions, and other identifiers in the source code to obscure its original meaning while preserving its execution.

In addition, we investigated the potential of each LLM to detect privacy-invasive code by creating and assessing three samples containing risky actions without asking user confirmation. These three privacy-invasive samples are also available at Vulcorpus (2024) along with Vulcorpus. The actions performed by these samples are as follows:

- Get the precise location of the device through the ‘android.permission.ACCESS_FINE_LOCATION’ permission, and directly share the latitude and longitude over the Internet via API. According to the Android API (Android Developers – Manifest Permissions, 2024), this permission has a ‘dangerous’ protection level, namely it may give the requesting application access to user’s private data, among others.
- Capture an image via the ‘ACTION_IMAGE_CAPTURE’ intent (Android Developers – MediaStore, 2024), and subsequently attempt to share the captured image file via API.
- Open local documents through the ‘ACTION_OPEN_DOCUMENT’ intent (Android Developers – Intent, 2024), and attempt to send them to a remote host via API.

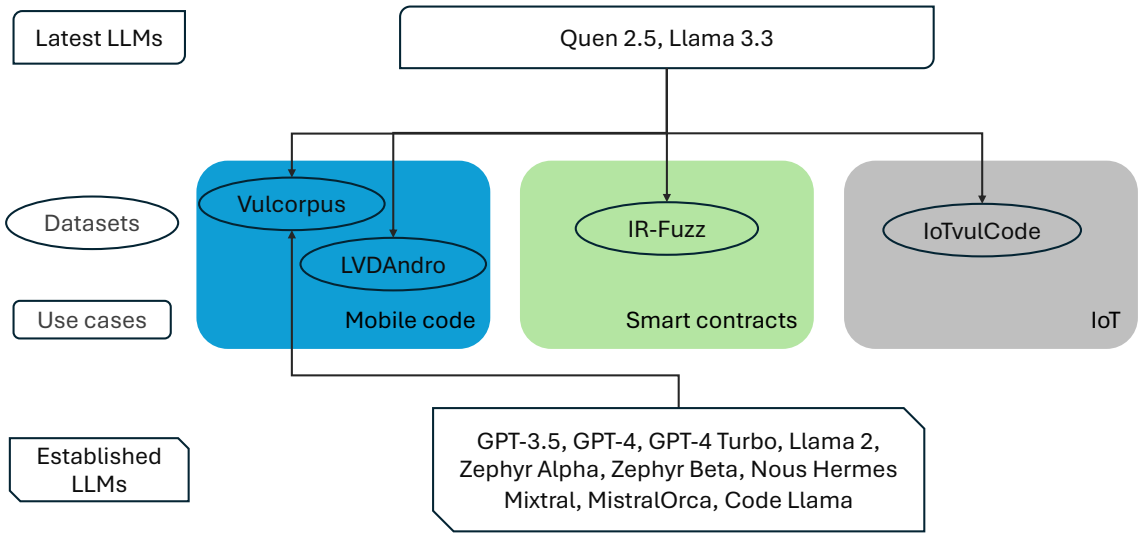
4.3 IR-Fuzz

In order to test the vulnerability detection capabilities of LLMs on smart contracts, we employed the IR-Fuzz (Liu et al., 2023b) dataset. It comprises over 12K real-world Ethereum smart contracts, including inherited contracts, which are organised across approximately 4K files. These files contain the source code of smart contracts that exhibit a wide range of behaviours and potentially security vulnerabilities. More specifically, the dataset contains smart contracts with eight distinct types of vulnerabilities, namely timestamp dependency (TP), block number dependency (BN), dangerous delegatecall (DG), Ether frozen (EF), unchecked external call (UC), reentrancy (RE), integer overflow (OF), and dangerous Ether strict equality (SE). Each smart contract of the dataset was checked manually by the authors of the dataset for vulnerabilities and labelling was performed according to vulnerability-specific patterns for each of the above vulnerabilities.

4.4 IoTvulCode

The IoTvulCode dataset (Bhandari et al., 2024) focuses on identifying source code vulnerabilities in IoT operating systems and applications. The software selected is open-source, written in C/C++, actively maintained and popular according to CVE records. This labelled dataset includes a collection of over 480K benign and 66K vulnerable samples of code functions. The authors utilised three different static analysis tools for labelling and categorised the identified vulnerabilities based on CWE.

Figure 1 Methodology overview (see online version for colours)



5 Vulnerabilities

This section describes the vulnerabilities that each benchmark dataset comprises. The vulnerabilities depend on the dataset and are of three different types:

1 vulnerabilities contained in the OWASP Top-10 list

2 smart contract vulnerabilities

3 vulnerabilities contained in CWE databases.

Subsection 5.1 lists the OWASP Top-10 vulnerabilities that are contained in *Vulcorpus* and *LVDAndro* datasets, Section 5.2 smart contract vulnerabilities that are relevant to *IR-Fuzz*, and Subsection 5.3 CWE vulnerabilities contained in *IoTvulCode*. Note that in our analysis we only consider OWASP Top-10 vulnerabilities for the LVDAndro dataset since CWE vulnerability detection is performed on the IoTvulCode dataset. Table 1 summarises the types of vulnerabilities considered in our analysis for each of the datasets.

Table 1 Type of vulnerabilities analysed for each dataset

Dataset	*OWASP*	*Smart contract*	*CWE*
LVDAndro	✓		
Vulcorpus	✓		
IR-Fuzz		✓	
IoTvulCode			✓

5.1 OWASP

In this subsection, we provide a brief overview of the vulnerabilities included in the OWASP Mobile Top-10. As we consider both the 2016 and 2024 versions, it is worth noting that the two lists differ, with the primary difference being that four vulnerabilities from the 2016 list have been replaced with new ones in the 2024 list; other differences include merging vulnerabilities or moving to a different position in the Top-10. For more details regarding each vulnerability category, the reader is referred to OWASP (2024) for the 2024 list and to OWASP (2016) for the 2016 list. Regarding the analysis of these vulnerabilities, it is important to consider that while some categories, such as M5 of the 2024 list (insecure communication), are relatively straightforward, others may be more complex and challenging for LLMs to comprehend, such as M2 of the 2024 list (inadequate supply chain security). In the rest of this subsection, we describe all the categories included in both lists using the following notation to signify to which list they belong: `<position in the list>-<list version>`; for example, the third category of the 2016 list is cited as M3-2016.

- *Improper credential usage (M1-2024):* Poor credential management can lead to severe security issues, namely, unauthorised users may be able to gain access to sensitive information or administrative functionalities within the mobile app or its back-end systems. This in turn leads to data breaches and fraudulent activities.
- *Improper platform usage (M1-2016):* This vulnerability occurs when a mobile app misuses or fails to use platform security controls, such as Android intents (messages that allow components to communicate) or TouchID (Apple's fingerprint authentication technology). This can expose a web service or API call to insecure coding techniques, allowing an attacker to feed malicious content to the other endpoint and exploit the vulnerability. To prevent this, developers must follow secure coding practices and guidelines, avoiding common risks such as storing sensitive data in insecure locations.
- *Inadequate supply chain security (M2-2024):* By exploiting vulnerabilities in the mobile supply chain, attackers may be able to manipulate application

functionality. For example, they can insert malicious code into the mobile application's codebase or libraries (Ars Technica, 2024), as well as modify the code during the application's build process to introduce backdoors, spyware, or other type of malware. The attacker can also exploit vulnerabilities in third-party software libraries, software development kits (SDKs), or hard-coded credentials to gain access to the mobile app or the back-end servers. Overall, this type of vulnerabilities can lead to unauthorised data access or manipulation, denial of service (DoS), or complete takeover of the mobile application or device.

- *Insecure authentication/authorisation (M4-2016, M6-2016, M3-2024):* Poor authorisation could lead to the destruction of systems or unauthorised access to sensitive information, while poor authentication results in the inability to identify the user making an action request, leading to the inability to log or audit user activity. This situation makes it difficult to detect the source of an attack, understand any underlying exploits, or develop strategies to prevent future attacks. Obviously, authentication failures are tightly coupled to authorisation failures; when authentication controls fail, authorisation cannot be performed. That is, if an attacker can anonymously execute sensitive functionality, it indicates that the underlying code is not verifying the user's permissions, highlighting failures in both authentication and authorisation controls.
- *Insufficient input/output validation (M4-2024):* A mobile application that does not adequately validate and sanitise data from external sources, such as user input or network data, is susceptible to a range of attacks, including SQL injection, command injection, and cross-site scripting. Insufficient output validation can also lead to data corruption or presentation vulnerabilities, possibly allowing the malicious actor to inject harmful code or manipulate sensitive information shown to the users.
- *Insecure communication (M3-2016, M5-2024):* Modern mobile applications typically communicate with one or more remote servers. This renders user data susceptible to interception and modification, if they are transmitted in plaintext or using an outdated encryption protocol.
- *Inadequate privacy controls (M6-2024):* Privacy controls aim to safeguard Personally Identifiable Information (PII), including names and addresses, credit card details, emails, and information related to health, religion, sexuality, and political opinions. This sensitive information can be used to impersonate the victim for fraudulent activities, misuse their payment data, blackmail them with sensitive information, or harm them by destroying or manipulating sensitive data.
- *Insufficient binary protections (M8-2016, M9-2016, M7-2024):* The application's binary may hold valuable information, such as commercial API keys or hard-coded cryptographic secrets. Furthermore, the code within the binary itself could be valuable, for instance, containing critical business logic or pre-trained AI models. In addition to gathering information, attackers may also manipulate app binaries to gain access to paid features for free or bypass other security controls. In the worst-case scenario, popular apps could be altered to include malicious code and then distributed through third-party app stores or under a new name to deceive unsuspecting users.
- *Client code quality (M7-2016):* Attackers can exploit poor code quality vulnerabilities by passing carefully crafted, untrusted inputs to method calls within mobile code, which can lead to security vulnerabilities such as buffer overflows and memory leaks. To prevent this, developers should maintain consistent coding patterns, write readable and well-documented code, and use automation tools to identify buffer overflows and memory leaks.
- *Security misconfiguration (M10-2016, M8-2024):* These occur when security settings, permissions, or controls are improperly configured, leading to vulnerabilities and unauthorised access.
- *Insecure data storage (M2-2016, M9-2024):* Such vulnerabilities may stem from weak encryption, insufficient data protection, insecure data storage mechanisms, and improper handling of user credentials.
- *Insufficient cryptography (M5-2016, M10-2024):* The use of obsolete cryptographic suites, primitives, or cryptographic practices may lead to loss of data confidentiality, integrity, and inability to impose source authentication among others. Typical repercussions include data decryption, manipulation of cryptographic processes, leak of encryption keys, etc.

5.2 Smart contract vulnerabilities

In this subsection, the eight distinct types of vulnerabilities contained in the IR-Fuzz dataset are described.

- *Timestamp dependency (TP):* Smart contracts often use block timestamps for time-based operations, such as fund transfers, auctions and lotteries. However, block timestamps are not completely immutable and can be slightly adjusted by the miner, leading to timestamp manipulation where functions are triggered prematurely or with a delay. For example, in an auction facilitated by smart contracts, a miner who also acts as a bidder could end the auction before the foreseen time when they are the highest bidder, thus winning the auction in an unfair way.

- *Block number dependency (BN):* Similar to TP, using block numbers in smart contracts as a condition to perform a critical operation, such as fund transfers, can create a vulnerability that allows an attacker to exploit the contract. This is feasible because although block numbers are considered a reliable indicator of time in general, they can also be manipulated to some extent by a miner, in a similar way as block timestamps.
- *Dangerous delegatecall (DG):* Delegatecall is a low-level interface that allows a contract A to invoke another contract B. The execution of the call, however, occurs in the environment of contract A, so that an attacker who manipulates the arguments can control the contract and execute arbitrary code.
- *Ether frozen (EF):* In this vulnerability, Ether is at risk of being frozen and become inaccessible if the contract developer does not implement all necessary functions for Ether transfer. For example, if only the receive function of Ether is implemented, Ether can become frozen as it will not be possible to transfer it out.
- *Unchecked external call (UC):* This vulnerability occurs when a smart contract fails to verify the outcome of an external call to another contract or address, potentially leading to incorrect assumptions about the call's success, such as silent failures. This can result in inconsistencies in the contract state, failed transactions, lost funds, and vulnerabilities that attackers can potentially exploit.
- *Reentrancy (RE):* A reentrancy attack takes place when a smart contract makes an external call before updating its own state, allowing a malicious contract to reenter the original function and repeat actions, such as withdrawals. This can enable an attacker to drain a contract's funds or trigger unauthorised function calls, leading to unintended actions.
- *Integer overflow (OF):* When an arithmetic operation results in a value that exceeds the maximum size of a fixed-size integer data type, it causes the value to wrap around to its minimum size. This vulnerability can enable an attacker to manipulate account balances or token amounts.
- *Dangerous Ether strict equality (SE):* Strict equality checks on a contract's balance can cause these checks to fail as the balance will be inconsistent with the total amount of Ether sent by users. This can potentially lock the contract, leading to a DoS attack.

5.3 CWE

The most common CWEs found in IoTvulCode (Bhandari et al., 2024) are: CWE-120, CWE-119, CWE-126, CWE-190, CWE-20, CWE-457, CWE-362, CWE-134, and CWE-367, which account for more than 95% of the reported vulnerabilities. A brief explanation of each of these vulnerabilities is given below:

- *CWE-120 – buffer copy without checking size of input:* A buffer overflow could occur when a program copies data from an input to an output buffer without checking the destination buffer size. If the destination buffer is not large enough to hold the copied data, the excess data will overflow the buffer, causing the writing of data to adjacent memory locations. This can lead to unpredictable behaviour, crashes, or even allow attackers to execute malicious code.
- *CWE-119 – improper restriction of operations within the bounds of a memory buffer:* This weakness occurs when a program reads from or writes to a memory location that is outside the boundaries of the allocated buffer. Such operations can lead to buffer overflows, unexpected results, execution of arbitrary code, crashes, or unauthorised access to sensitive data.
- *CWE-126 – buffer over-read:* A buffer over-read takes place when a program reads data that are located beyond the allocated space of a buffer. As a result, the program may read sensitive data, crash, or experience unexpected results.
- *CWE-190 – integer overflow or wraparound:* When a program performs arithmetic operations on integers, it can produce a value that is too large to fit in the maximum limit of the integer type. This can cause the integer to overflow or wraparound, producing very small or negative numbers and leading to unexpected behaviour, crashes, or security vulnerabilities.
- *CWE-20 – improper input validation:* Before accepting input or data, a program should first validate that this input is correct and safe, otherwise it may end up accepting malicious or malformed data. Properties that can be checked in this context include size, length, data structure position, syntax, input type, business logic, and authenticity.
- *CWE-457 – use of uninitialised variable:* There are some languages, such as C/C++, where variables are not initialised by default and, thus, contain garbage values. The use of uninitialised variables can lead to unpredictable control flow or results, DoS, arbitrary code execution or disclosure of sensitive data.
- *CWE-362 – concurrent execution using shared resource with improper synchronisation ('race condition'):* In multithreaded or concurrent environments, a code sequence might require temporary, exclusive access to a shared resource for a given time period. In this context, a race condition could occur when another code sequence accesses this shared resource during this period, violating exclusivity. This could lead to data corruption, crashes, or security vulnerabilities.
- *CWE-134 – use of externally-controlled format string:* Format strings are used to control the output of data

in a program. When a program uses an externally-controlled format strings, such as user-provided input, it may allow attackers to inject malicious format specifiers, leading to security vulnerabilities or crashes.

- *CWE-367 – time-of-check time-of-use (TOCTOU) race condition:* A program checks the state of a resource or variable before using it. However, an attacker can change this state between the check and the actual use of the resource, leading to a TOCTOU race condition. This can lead to security vulnerabilities, crashes, or unexpected behaviour.

6 Large language models

The selection of LLM models for our experiments was based on a combination of factors, including their implementation type, licensing, and computational requirements; our purpose was to diversify as much as possible the selection of models. Specifically, we chose Qwen 2.5 and Llama 3.3 for the horizontal dimension of our study because they are both open-source models that can be deployed on-premise, allowing for more control over data and computation but at the expense that they may require significant computational resources to deploy and maintain. In contrast, commercial models, such as GPT-3.5 and GPT-4, are cloud-based and proprietary, which provides scalability and ease of use, but also introduces potential limitations related to data privacy, as well as additional costs and dependence on cloud infrastructure. In terms of computational and time complexity, we considered the parameter count and inference time of each model. For example, Llama 3.3 has 70 billion parameters and supports up to 128K tokens of context length, while Qwen 2.5 has a comparable coding capability to GPT-4, but with a smaller parameter count. We also considered the time complexity of each model, with some models like GPT-4 Turbo being optimised for faster inference times. By considering these factors, we aimed to provide a comprehensive evaluation of the strengths and limitations of different LLMs in code vulnerability detection, and to identify the most suitable models for different use cases and deployment scenarios.

For the study of the horizontal dimension, as described in Section 3, two open-source LLMs were selected, i.e., Qwen 2.5 and Llama 3.3, to detect source code vulnerabilities in four diverse datasets. The choice of these models was motivated by their open-source nature, which allows for transparency and reproducibility of the results. Additionally, both Qwen and Llama are widely recognised models in the field of natural language processing (NLP), with a large community of users and contributors. In the following, a short description of these two models is provided.

- *Qwen 2.5 (qwen-coder-2.5-instruct, 32K tokens) (Qwen2.5-Coder, 2025):* It is a state-of-the-art, open-source code model, having comparable coding capabilities with GPT-4o and supporting 92 coding languages. Apart from coding, it also has good general and mathematical skills, supporting long context understanding and generation with up to 128K tokens context length. Its training dataset comprises 5.5 trillion tokens with 70% code, 20% text and 10% math from diverse repositories, such as GitHub and Kaggle.
- *Llama 3.3 (Llama-3.3-70b-instruct-ui, 128K tokens) (Llama-3.3-70B-Instruct, 2025):* The Llama 3.3 model was developed by Meta and released in 6 December 2024, whereas its knowledge cut-off date is December 2023. It uses a 70-billion parameter instruction-tuned architecture and was designed with a focus on multilingual dialogue applications and demonstrates improved performance compared to other open-source and proprietary chat models. It also features improved coding proficiency, assisting developers in code generation and debugging.

For the vertical dimension, i.e., analysing mobile code using the Vulcorpus dataset, nine established LLMs were used: three commercial models, namely, GPT-3.5, GPT-4, and GPT-4 Turbo, and six open source models, namely, Llama 2, Zephyr Alpha, Zephyr Beta, Nous Hermes Mixtral, MistralOrca, and Code Llama. According to their documentation, these models have been pre-trained on large amounts of text data, including code, having demonstrated performance in various software engineering tasks, including code analysis. That is, their ability to understand code syntax and semantics makes them well-suited for identifying vulnerabilities residing in code, while their large size and diverse training data make them less likely to overfit to a specific codebase. A brief description of each LLM is given below. It is important to note that we used the default settings of each model.

- *GPT 3.5 (version 1106) (Brown et al., 2020):* It is a powerful language model that has been pre-trained on a large corpus of text data, including code. It has demonstrated performance in various NLP tasks and has been used for code analysis tasks such as code completion, code search, and code summarisation.
- *GPT 4 (version gpt-4-32k) (OpenAI, 2024a,b):* It is a newer version of GPT, pre-trained on an even larger corpus of text data, including code. It has demonstrated improved performance over GPT 3.5 in various NLP tasks and has been used for code analysis, including code review and repair.
- *GPT 4 Turbo (version turbo-2024-04-09):* It is a variant of GPT 4, specifically designed for tasks that require faster inference times, such as code analysis. It has been pre-trained on the same large corpus of text data as GPT 4, optimised for faster performance.
- *Llama 2 (version Llama-2-70b-chat) (Touvron et al., 2023):* This LLM has been pre-trained on a diverse set of text data, including code. It has demonstrated performance in various NLP tasks, also been

exploited for code analysis, including code summarisation and code search.

- *Zephyr Alpha (version zephyr-7b-alpha) (Tunstall et al., 2023):* It is pre-trained on a huge corpus of text data from diverse sources, including books, articles, and websites. This model has been fine-tuned with a mix of publicly available and synthetic datasets on top of Mistral LLM. Despite its small size (7B parameters), it potentially shows a performance comparable to several models with a number of parameters in the range of 20–30B.
- *Zephyr Beta (version zephyr-7b-beta) (Tunstall et al., 2023):* This model has been fine-tuned with a mix of publicly available and synthetic datasets on top of Mistral LLM. It is the successor of Zephyr Alpha, therefore considered significantly more powerful than its predecessor. Based on its documentation, it is fast and competent, showing a performance comparable to the best open-source models, having around 70B parameters.
- *Nous Hermes Mixtral (version nous-hermes-2-mixtral-8x7b-dpo) (Nous Hermes 2 Mixtral 8x7b DPO, 2024):* It is one of the most powerful open-source models available, comprising a fine-tuned version of Mixtral base model.
- *MistralOrca (version mistral-7b-openorca) (Lian et al., 2023; Mukherjee et al., 2023; Longpre et al., 2023):* It has been fine-tuned with Open-Orca datasets on top of Mistral LLM. Despite its small size, it outperforms Llama 2 13B, showing a performance comparable to several models with a number of parameters in the range of 20-30B.
- *Code Llama (version 7b) (Code Llama, 2024):* It is a special version of Llama 2, tailored specifically for coding applications. This specialised version has been refined through extensive additional training on code-focused data, with prolonged exposure to relevant datasets. The result is a tool with alleged superior coding proficiency that builds upon the foundation of Llama 2. More specifically, Code Llama can generate code and create explanations about code in response to prompts in both programming and natural language. Its capabilities extend to assisting with code completion and troubleshooting code errors. Furthermore, Code Llama is versatile, supporting a broad array of widely-used programming languages, including Python, C++, Java, PHP, JavaScript, C Sharp, and Bash. In this work, we examine the smallest pre-trained model, namely, the 7B version. In addition, for this LLM, in a separate run, we employed LlamaIndex (LlamaIndex, 2024) to improve the detection capabilities of Code Llama. LlamaIndex is a data framework for LLM-based applications, enhancing them with additional contextual data. This context augmentation technique is called retrieval-augmented generation (RAG) and can be used to address the restrictions of LLMs by giving them access to contextual, current data. For the RAG process, we selected the bge-small-en-v1.5 (Baai/bge-small-en-v1.5, 2024) embedding model, developed by the Beijing Academy of Artificial Intelligence. Additionally, we used the 50% of Vulcorpus, i.e., only the samples that contain code comments regarding the specific vulnerability. Android's application quality and security guidelines and code examples (Security Guidelines, 2024) were also added as input to the RAG, along with information on each vulnerability from the OWASP website (OWASP, 2024).

7 Evaluation

This section summarises the evaluation procedure we followed to assess the effectiveness of the selected LLMs in vulnerability detection. First, we give an overview of the experimental setup, including the platform we used to run the models and the actual prompts provided as input to the LLMs. Then, we discuss our findings, providing insight into the outcomes we observed and offering possible reasons for the specific results we obtained, where necessary.

7.1 Experimental setup

In this section, we give an overview of the setup of both the dimensions we described in Section 3. First, we analyse the horizontal dimension that comprises two LLMs, namely Qwen and Llama, and then the vertical one, where we used nine LLMs on the Vulcorpus dataset. Apart from the technical setup, we also provide details on the elements we analysed as well as the prompts submitted to the LLMs.

For the study of the horizontal dimension, we utilised the API interface of the *GPT@JRC* platform (Longueville et al., 2025), a system developed by the European Commission's Joint Research Centre (JRC), which enabled us to test the capabilities of two LLMs, namely, Qwen 2.5 and Llama 3.3, to analyse code samples across four diverse datasets and identify potential vulnerabilities. The datasets used in our experiments are LVDAndro, Vulcorpus, IR-Fuzz, and IoTvulCode, which comprise code samples for the Android, smart contract, and IoT use cases. We compared the results acquired by the LLMs, with those listed in the above labelled dataset to check their capacity in identifying code vulnerabilities in heterogeneous code samples.

The following prompts were used for each of the three use cases.

Android – LVDAndro

Check if the following Android code has any Mobile OWASP Top 10 vulnerability. I want you to only print using the following CSV format with two values: `[vulnerable]`, `[OWASP Mobile Top 10 category]`. The first value is Boolean. The second is of type string and lists the OWASP Mobile Top 10 categories, which is one or more of the following: M1: improper platform usage, M2: insecure data storage, M3: insecure communication, M4: insecure authentication, M5: insufficient cryptography, M6: insecure authorisation, M7: client code quality, M8: code tampering, M9: reverse engineering, M10: extraneous functionality. If a piece of code has more than one vulnerability category, put them in the same variable. That way, you will always print one CSV line. Code: `<input>`

Android – Vulcorpus

Check if the following Android code has any Mobile OWASP Top 10 vulnerability. I want you to only print using the following CSV format with two values: `[vulnerable]`, `[OWASP Mobile Top 10 category]`. The first value is Boolean. The second is of type string and lists the OWASP Mobile Top 10 categories, which is one or more of the following: M1: improper credential usage, M2: inadequate supply chain security, M3: insecure authentication/authorisation, M4: insufficient input/output validation, M5: insecure communication, M6: inadequate privacy controls, M7: insufficient binary protections, M8: security misconfiguration, M9: insecure data storage, M10: insufficient cryptography. If a piece of code has more than one vulnerability category, put them in the same variable. That way, you will always print one CSV line. Code: `<input>`

Smart contracts

Check if the following solidity smart contract code has any vulnerability. I want you to only print using the following CSV format with two values: `[vulnerable]`, `[vulnerability category]`. The first value is boolean. The second is of type string and lists the vulnerability which is one or more of the following: block number dependency (BN), dangerous delegatecall (DE), Ether frozen (EF), Ether strict equality (SE), integer overflow (OF), reentrancy (RE), timestamp dependency (TP), unchecked external call (UC). Use the 2-code letters for each vulnerability and if a piece of code has more than one, put them in the same variable. That way, you will always print 1 CSV line. in the same variable. Code: `<input>`

IoT

Check if the following code has any vulnerability. I want you to only print using the following CSV format with two values: `[vulnerable]`, `[CWE]`. The first value is Boolean and the second is of type string and lists the CWEs. In case you do not know the CWE number, add the CWE category. Code: `<input>`

For studying the vertical dimension, eight of the nine pre-trained LLMs run on the *GPT@JRC* platform. The last one, Code Llama, was run on a local computer with an M2 processor and 16 GB unified memory. All LLMs were tested in the Android use case through the Vulcorpus dataset, assessing their ability in identifying potential vulnerabilities and proposing code improvements. To this end, a simple scoring system was used to present

1 the number of vulnerabilities each LLM was able to detect

2 if the LLM proposed valid suggestions for possibly fixing the vulnerability.

Both these partial scores have a maximum value of 10/10 per vulnerability category, i.e., one point for each piece of vulnerable code the LLM was able to detect and annotate.

As previously mentioned, the LLMs used in this work are pre-trained. This means that the associated libraries, possibly needed by each code sample but not included in the input, cannot be analysed. This mostly affects the analysis regarding the M2 vulnerability. Therefore, to evaluate LLMs against M2, instead of Java code, we used ten libraries with known vulnerabilities as input. These libraries, also included in Vulcorpus for reasons of reproducibility, were published before the training date of each LLM.

At a final stage, as detailed in Section 7.2, the results of each LLM were compared and cross-checked against those produced by two well-known SAST tools, namely Bearer (Bearer, 2024) and MobSFscan (MobSFscan, 2024). Bearer is a static application security testing tool, which uses built-in rules covering the OWASP Top 10 and CWE Top 25. MobSFscan is a static analysis tool that uses MobSF's (MobSF, 2024) security rules and can find insecure code patterns in Android or iOS source code. Finally, we also assessed the performance of each LLM in detecting privacy-invasive behaviours, using the three samples detailed in Section 4. The output was rated using three categories:

1 not privacy-invasive

2 potentially privacy-invasive

3 privacy-invasive.

It is important to note that the prompt given to the LLM has a major effect on its output. Since in the horizontal scenario a detailed prompt was provided, here a brief instruction was given to investigate also this aspect; the prompt was as follows.

Android

Check if there are any security issues in the following code; if there are, explain the issue. Code: `<input>`

7.2 Results

The first set of experiments concerns the horizontal dimension, that is, the detection capabilities of Qwen and Llama across the four datasets. Specifically, as described in Section 3, the first set of experiments checks whether there is a vulnerability in a given source code sample, whereas

the second set checks whether the LLM can correctly identify which specific vulnerability it is. In Table 2, the results of the first set are summarised; in Table 3, we present the accuracy of the two LLMs in describing the specific vulnerabilities detected in the previous step.

Looking at Table 2 and leaving out LVDAndro, the two LLMs showed high performance in detecting whether a vulnerability existed in the tested code samples or not. More specifically, Qwen 2.5 scored 67% in the IoT case and above 96% in smart contracts and mobile code (Vulcorpus). On the other hand, Llama 3.3 scored above 94% in all three use cases. Regarding the low detection rates of LVDAndro, our interpretation is that the code samples of this dataset are too short (usually less than one line) to allow the LLMs to easily identify whether they contain a vulnerability or not. This is further supported by the consistent low rates observed by both LLMs.

Table 2 Vulnerability detection results across all benchmark datasets

Dataset	*Qwen 2.5*		*Llama 3.3*	
	Detected	*Percentage*	*Detected*	*Percentage*
LVDAndro	2,777/13,750	20.2%	4,822/13,750	35.1%
Vulcorpus	100/100	100%	100/100	100%
IR-Fuzz	4,147/4,285	96.8%	4,260/4,285	99.4%
IoTvulCode	44,840/66,699	67.3%	63,294/66,699	94.9%

Table 3 Identifying vulnerabilities across all benchmark datasets

Dataset	*Qwen 2.5*		*Llama 3.3*	
	Detected	*Percentage*	*Detected*	*Percentage*
LVDAndro	3,162/13,750	23%	2,979/13,750	21.7%
Vulcorpus	57/100	57%	57/100	57%
IR-Fuzz	2,385/4,285	55.7%	3,667/4,285	85.6%
IoTvulCode	16,045/66,699	24%	11,628/66,699	17.4%

After detecting whether a vulnerability exists, the two LLMs were asked to identify the specific vulnerability. The results shown in Table 3 reveal a low to medium capability of the two LLMs in determining which exactly is the vulnerability in the examined code samples. In the mobile code (LVDAndro) and the IoT use case (IoTvulCode dataset), all checks had an identification accuracy of no more than 24%. For the other mobile code dataset (Vulcorpus) and the smart contract dataset (IR-Fuzz), the two LLMs scored between 55 and 86%.

The next set of experiments concerns the vertical dimension, namely, the detection capabilities of nine LLMs on the mobile device use case, using the Vulcorpus dataset. Table 4 and Figure 2 recapitulate the results for each LLM on the Vulcorpus dataset. Particularly, in each line of Table 4, letter 'D' indicates that the model *detected* the vulnerability, whereas 'I' denotes whether it explained the situation and provided a valid solution for *improving* the code or not. The importance of 'I' lies in the fact that it is the sole indicator of whether the LLM actually 'perceived' the security issue and can be used to evaluate each LLM and compare them to each other.

Overall, with reference to Table 4, the LLMs that detected the most vulnerabilities in total were: Code Llama (81/100), GPT 4 (67/100), Nous Hermes Mixtral (62/100), Zephyr Beta (54/100), and Zephyr Alpha (53/100), followed by GPT 4 TURBO (50/100), GPT 3.5 (42/100), MistralOrca (37/100), and Llama 2 (30/100). When it comes to total code improvement suggestions, the top performers were: GPT 4 (83/90), GPT 4 Turbo (66/90), Zephyr Alpha (58/90), Zephyr Beta (56/90), and Nous Hermes Mixtral (56/90), followed by Code Llama (44/90), MistralOrca (38/90), GPT 3.5 (37/90), and Llama 2 (31/90). Overall, considering both 'D' and 'I' values, GPT 4 poses as the top performer. On the other hand, LLMs that identify the correct vulnerability, but fail to provide corrections or suggestions regarding the problematic lines of code, such as Code Llama, may indicate an insufficiently trained model for this type of analysis.

Focusing on individual vulnerabilities, GPT 4 discovered all vulnerabilities of categories M1 and M6, MistralOrca of M9, Zephyr alpha of M5 and M10, Zephyr beta of M5 and M9, and Llama 2 and Code Llama of M5. Note that for the rest of this section all these categories concern the OWASP 2024 top-ten list. No LLM achieved a perfect score for the rest of the vulnerabilities, namely, M2, M3, M4, M7, and M8. In these scenarios the best performers were GPT 3.5 (7/10), Zephyr Beta and Code Llama (9/10), Nous Hermes Mixtral (9/10), Code Llama (9/10), and Nous Hermes Mixtral and Code Llama (9/10), respectively. Concerning M2, even though it was tested on 10 vulnerable libraries published before the training date of each model, all LLMs with a few exceptions show low detection performance, as presented in Table 4. This behaviour may imply that LLM training did not include these libraries. The same applies to the 'I' column for M2, which is marked as *N/A* for all LLMs. These limitations can be addresses using context augmentation, as discussed in Section 6. This way, LLMs can be provided with access to recent data and improve their vulnerability detection capabilities.

For the purpose of better understanding which kind of vulnerabilities are hard for LLMs to detect, we averaged the 'D' score per vulnerability for all nine LLMs. Then, we sorted the mean values in ascending order; the results are presented in Table 5. In the same table, we included the best performer in each category, showing which LLM could handle better the respective vulnerability. Our results show that the most difficult vulnerability to detect for an LLM is M2, whereas the easiest is M5. As we also explained above, the low detection capability in the case of M2 (2.1 on average) can be attributed to the lack of up-to-date information at the LLMs' side. Moreover, another factor that could contribute to this low score is that, for this type of vulnerabilities, the LLMs check a list of libraries for known vulnerabilities instead of directly analysing the application's source code. This is also the only vulnerability where no LLM had a perfect (10) or nearly perfect score (9). On the other side of the spectrum, four LLMs achieved a perfect detection score for M5.

Table 4 Vulnerability analysis results for the Vulcorpus dataset

LLM	*M1*		*M2*		*M3*		*M4*		*M5*		*M6*		*M7*		*M8*		*M9*		*M10*		*Mean*	
	D	*I*	*D*	*I*	*D*	*I*	*D*	*I*	*D*	*I*	*D*	*I*	*D*	*I*	*D*	*I*	*D*	*I*	*D*	*I*	*D*	*I*
Llama 2	0	0	0	N/A	0	*10*	4	5	*10*	0	6	6	0	0	0	0	4	4	6	6	3	3.4
GPT-3.5	3	3	*7*	N/A	2	3	2	3	8	6	3	5	5	4	3	5	4	6	5	2	4.2	4.1
GPT-4	*10*	*10*	0	N/A	6	7	6	*8*	5	*10*	*10*	*10*	6	*9*	7	*10*	8	*10*	9	9	6.7	*9.2*
GPT-4 TURBO	4	5	0	N/A	3	5	5	*8*	8	9	6	8	4	4	7	*10*	7	9	6	8	5	7.3
Zephyr Alpha	0	0	3	N/A	6	6	5	5	*10*	*10*	7	8	2	2	7	7	3	*10*	*10*	*10*	5.3	6.4
Zephyr Beta	0	8	0	N/A	*9*	9	8	*8*	*10*	*10*	0	8	3	0	5	4	*10*	0	9	9	5.4	6.2
Nous Hermes Mixtral	1	3	6	N/A	1	3	*9*	5	6	8	8	8	7	3	*9*	9	8	*10*	7	7	6.2	6.2
Mistral Orca	9	9	0	N/A	2	2	4	4	5	5	0	0	3	4	0	1	*10*	*10*	4	3	3.7	4.2
Code Llama*	9	5	3	N/A	*9*	4	8	4	*10*	5	8	5	*9*	4	*9*	4	7	7	9	6	*8.1*	4.9

Notes: 'D' and 'I' stand for the number of vulnerable samples detected and the number of vulnerable samples for which the LLM suggested improvements, respectively; top scores per vulnerability are in italic; the asterisk exhibitor stands for Code Llama without RAG.

Figure 2 Vulnerability detection for the Vulcorpus dataset using nine established LLMs (see online version for colours)

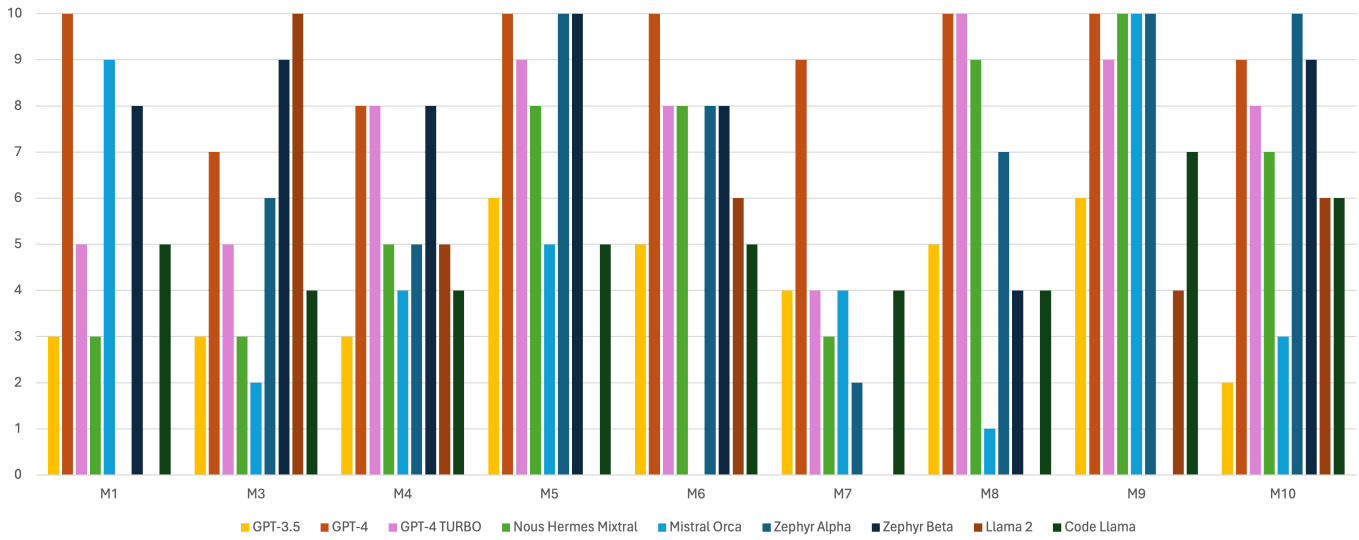


Table 5 Best performing LLM model and average detection score per vulnerability for the Vulcorpus dataset

Vulnerability	*Best performer*	*Mean score*
M1	GPT 4 (10)	4
M2	GPT 3.5 (7)	2.1
M3	Zephyr Beta (9), Code Llama* (9)	4.33
M4	Nous Hermes Mixtral (9)	5.66
M5	Zephyr Alpha (10), Zephyr Beta (10), Llama 2 (10), Code Llama* (10)	8
M6	GPT 4 (10)	5.33
M7	Code Llama* (9)	4.33
M8	Nous Hermes Mixtral (9), Code Llama* (9)	5.22
M9	MistralOrca (10), Zephyr Beta (10)	6.77
M10	Zephyr Alpha (10)	7.22

Notes: The asterisk exhibitor stands for Code Llama without RAG.

In addition to the above, our last set of experiments concerned the detection of privacy-invasive actions, as described in Subsection 4.2; the obtained results are presented in Table 6. Regarding the easiness of detection of each action, six LLMs correctly detected potential privacy-invasive actions for location, eight LLMs for camera, and six for local file sharing. Looking at the performance of each LLM individually, the best performer was Zephyr Alpha, which clearly marked two out of three codes as privacy-invasive and the other as potentially privacy-invasive. On the other hand, MistralOrca did not detect any possible privacy-invasive actions.

The next set of experiments concerned the use of RAG to understand to which degree it can assist LLMs to detect code vulnerabilities more effectively. These experiments were run on Code Llama and the following were used as input for RAG:

1 half of the samples of each vulnerability annotated with comments explaining the vulnerable code

2 text and code examples from Android's app security guidelines (Security Guidelines, 2024)

3 all the CVEs related to the vulnerable libraries used for M2 vulnerability.

Then, we analysed the other half of the code samples that were not annotated. The results, presented in Table 7, show in general increased performance both in vulnerability detection and in improvement suggestion. When excluding M2, in Code Llama without RAG (Table 4), the detection rate ranged from 70 to 100%, with one test scoring 100% and five tests scoring 90%. In Code Llama with RAG the detection rate ranges from 80 to 100%, with six tests scoring 100% and three scoring 80%. In M2, the base model scored 30% in detection, whereas the RAG-assisted scored 100%, an improvement of 233%. For improvement suggestion, Code Llama without RAG had a detection range from 40 to 70%, where the majority of tests (seven out of nine) scored either 40 or 50%. When using the RAG-based model, only one test scored 80% and the rest 100%, showing that RAG covered a wider gap in assisting improvement suggestion to reach a nearly perfect score compared to vulnerability detection where scores were already closer to 100%.

Table 6 Results per LLM regarding privacy-invasive actions

LLM	*Location*	*Camera*	*Files*
Llama 2	P	*Y*	N
GPT 3.5	*Y*	P	P
GPT 4	N	P	N
GPT 4 Turbo	P	*Y*	P
Zephyr Alpha	*Y*	P	*Y*
Zephyr Beta	*Y*	P	N
Nous Hermes Mixtral	*Y*	P	P
MistralOrca	N	N	N
Code Llama	N	N	*Y*
Code Llama + RAG	N	*Y*	P

Notes: N: not privacy-invasive, P: potentially privacy-invasive, Y: privacy-invasive.

Table 7 Evaluation results for Code Llama with RAG for the Vulcorpus dataset

Vulnerability	*D*	*I*
M1	5/5	5/5
M2	10/10	N/A
M3	5/5	5/5
M4	5/5	5/5
M5	5/5	5/5
M6	5/5	5/5
M7	5/5	5/5
M8	4/5	4/5
M9	4/5	5/5
M10	4/5	5/5

Notes: The letters 'D' and 'I' stand for the number of vulnerable samples detected and the number of vulnerable samples for which the LLM suggested improvements, respectively.

Table 8 Results of prominent SASTs on Vulcorpus

SAST	*M1*	*M2*	*M3*	*M4*	*M5*	*M6*	*M7*	*M8*	*M9*	*M10*	*Total*
MobSFscan	2	N/A	0	0	1	1	1	3	0	4	*12*
Bearer	2	N/A	0	1	6	3	3	4	3	7	*29*

To gain insight into whether the performance of LLMs can be considered acceptable or not, we compared them against two reputable SASTs, namely Bearer and MobSFscan. The results of these two tools when used to detect vulnerabilities in the Vulcorpus dataset are presented in Table 8. Overall, MobSFscan detected 12/100 vulnerabilities, whereas Bearer found 29/100 vulnerabilities; as a comparison, the worst scoring LLM (Llama 2) found 30/100 vulnerabilities and the best one (Code Llama wihout RAG) found 81/100. These results suggest that LLMs have in general better performance in detecting code vulnerabilities compared to current, well-known SASTs. When comparing the detection capacity of specific vulnerability types looking at Tables 8 and 5, it is shown that LLMs score significantly higher compared to SASTs in M3, M4 and M9, and to a lesser extent to M1, M6, and M7. Moreover, it is evident that specific types seem easier to detect for both LLMs and SASTs, such as M10 and to a lesser extent M5.

8 Discussion

The discussion in this section will be structured around the following research questions, which aim to provide a comprehensive understanding of our findings:

- What are the performance differences between newer and older LLMs in the context of code vulnerability detection?
- How do various datasets influence the performance of LLMs in detecting code vulnerabilities?
- Is there a particular LLM that consistently outperforms others across different types of tasks related to code vulnerability analysis?
- Which type of LLM model (open-source or commercial) demonstrates superior performance in detecting code vulnerabilities?
- Can the RAG mechanism enhance the performance of LLMs for code vulnerability analysis?

By addressing these research questions, this section summarises our findings into the current state of LLMs in code vulnerability detection, highlighting areas of strength, weakness, and potential for future improvement.

The first observation that emerges from considering both dimensions of our study is that in common tests the performance of newer LLMs has increased. More specifically, in both the horizontal and vertical dimensions we analysed a common dataset, namely, Vulcorpus, with 11 different LLMs in total. In the horizontal case, both LLMs scored 100%, whereas in the vertical scenario the mean detection rate across the nine LLMs was 53%. This increase could be attributed to two main differences:

1 newer models are more efficient and/or trained with more timely data

2 the LLMs of the horizontal scenario were given a more detailed prompt compared to the ones of the vertical scenario.

Focusing on the horizontal dimension, the diversity of detection rates across the different use cases and even among the same use case, show that results are highly dependent on the characteristics of the dataset. Starting from the common use case (mobile devices), we used two different datasets (LVDAndro and Vulcorpus) comprising Android code samples but the results were very dissimilar (20 to 35% for LVDAndro and 100% for Vulcorpus); a similar situation is observed for identifying the vulnerability category. The fact that the detection rates of the two LLMs are close to each other leads to the conclusion that the difference is due to the datasets' characteristics and not the LLM's characteristics. The same pattern is found when considering all the datasets we examined: the results between the two LLMs tend to converge, but at the same time the detection rates between different datasets have a wide range, from 17 to 100%. This observation highlights also the importance of result validation using diverse experiments before arriving to conclusions when investigating LLM effectiveness in code vulnerability detection.

In the vertical dimension, having tested a large number of LLMs (9) across a single dataset (Vulcorpus), more dissimilarities among the LLMs were observed compared to the horizontal case. Overall, LLMs showed a general ability to detect code vulnerabilities but some of them were more appropriate to be used for such a purpose. The top performer LLMs when considering both detection and improvement suggestion were GPT-4 and Code Llama. However, the first one was not as good in detection as it was in improvement suggestions where it had the highest mean score (9.2); in contrast, Code Llama had the top performance in vulnerability detection (8.1), but a rather low score in improvement suggestions. Focusing on vulnerabilities detection, MistralOrca and Zephyr Beta performed exceptionally well for M9, while Zephyr Alpha scored high in M10. Additionally, several LLMs struggled with M1, while most were unable to identify M2. This highlights that there is no single LLM that excels in all cases, indicating the need for selecting an LLM based on the vulnerability category and the type of task that it is expected to perform.

Another interesting question is which type of LLM model is better in detecting code vulnerabilities: open or commercial ones? When comparing open LLM models with commercial ones, we see that the open models were the best performers in seven out of ten categories of vulnerabilities, i.e., M3, M4, M5, M7, M8, M9, M10. On the other hand, considering mean detection and improvements scores, as presented in Table 4, the situation is mixed.

In addition to the above, we evaluated the use of RAG in fine-tuning LLMs for code vulnerability analysis. Our results suggest that RAG can significantly increase both the vulnerability detection and improvement suggestion performance. On the detection of privacy-invasive actions, the results are mixed, showing that they should be interpreted carefully when it comes to privacy analysis in mobile platforms. Zephyr Alpha had the best performance, whereas GPT 3.5, GPT 4.0 Turbo, and Nous Hermes Mixtral indicated all actions as privacy or potentially privacy-invasive; on the other hand, MistralOrca did not identify any potential privacy-invasive actions. Finally, the comparison of LLMs with well-known SASTs for vulnerability detection revealed that the former are more capable in identifying code vulnerabilities.

9 Conclusions

This study provides empirical evidence on the effectiveness of using LLMs for code vulnerability analysis across multiple domains, namely Android, smart contracts, and IoT. Our findings reveal that while some LLMs are capable of detecting code vulnerabilities, their overall performance is still in an early stage and requires further refinement. The observed variability in accuracy in vulnerability detection and improvement suggestion across different LLMs and datasets underlines that no LLM is currently fit to detect different kinds of vulnerabilities in an horizontal manner. Furthermore, the limitations of LLMs, such as training data cutoff and hallucinations, can hinder their ability to detect emerging vulnerabilities, deprecated libraries, and zero-day exploits. This underscores the critical need for continuous model updates and proactive retraining. Therefore, extended testing is needed with different models before taking an informed decision on which one to use for a specific use case and data format. Also, further research is needed by researchers and developers on the improvement of LLMs towards creating more effective and efficient vulnerability detection tools and address the aforementioned limitations. As future work, we plan to further expand our research with other LLMs not used in this study, newer versions of the LLMs we used here, and different use cases and datasets.

Declarations

All authors declare that they have no conflicts of interest.

References

Abendroth Dias, Kulani et al. (2025) *Generative AI Outlook Report*, Tech. Rep. KJ-01-25-309-EN-N (online), KJ-01-25-309-EN-C (print), Luxembourg.

Action1 (2025) *2025 Software Vulnerability Ratings Report* [online] https://www.action1.com/software-vulnerability-ratings-report-2025/ (accessed 10 July 2025).

Al-Hawawreh, M., Aljuhani, A. and Jararweh, Y. (2023) 'ChatGPT for cybersecurity: practical applications, challenges, and future directions', *Cluster Computing*, Vol. 26, No. 6, pp.3421–3436.

Android Developers – Intent (2024) [online] https://developer.android.com/reference/android/content/Intent#ACTION_OPEN_DOCUMENT (accessed 20 August 2024).

Android Developers – Manifest Permissions (2024) [online] https://developer.android.com/reference/android/Manifest.permission#ACCESS_FINE_LOCATION (accessed 20 August 2024).

Android Developers – MediaStore (2024) https://developer.android.com/reference/android/provider/MediaStore#ACTION_IMAGE_CAPTURE (accessed 20 August 2024).

Ars Technica (2024) *What We Know About the XZ Utils Backdoor that Almost Infected the World* [online] https://arstechnica.com/security/2024/04/what-we-know-about-the-xz-utils-backdoor-that-almost-infected-the-world/ (accessed 17 April 2024).

Baai/bge-small-en-v1.5 (2024) [online] https://huggingface.co/BAAI/bge-small-en-v1.5 (accessed 20 August 2024).

Bearer (2024) [online] https://github.com/Bearer/bearer (accessed 20 August 2024).

Bhandari, G.P., Gebremariam, A., Nikola, G., Andrii, S. and Tor-Morten, G. (2024) 'IoTvulCode: AI-enabled vulnerability detection in software products designed for IoT applications', *International Journal of Information Security*, Vol. 23, No. 4, pp.2677–2690.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I. and Amodei, D. (2020) 'Language models are few-shot learners', in Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. and Lin, H. (Eds.): *Advances in Neural Information Processing Systems*, Vol. 33, pp.1877–1901, Curran Associates, Inc., Red Hook, New York, USA.

Cheshkov, A., Zadorozhny, P. and Levichev, R. (2021) *Evaluation of ChatGPT Model for Vulnerability Detection*, arXiv preprint arXiv:2304.07232.

Church, K.W. (2017) 'Word2vec', *Natural Language Engineering*, Vol. 23, No. 1, pp.155–162.

Code Llama (2024) [online] https://ai.meta.com/blog/code-llama-large-language-model-coding/ (accessed 20 August 2024).

Devlin, J., Chang, M-W., Lee, K. and Toutanova, K. (2019) 'BERT: pre-training of deep bidirectional transformers for language understanding', in Burstein, J., Doran, C. and Solorio, T. (Eds.): *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Long and Short Papers, Association for Computational Linguistics*, Minneapolis, Minnesota, Vol. 1, pp.4171–4183, DOI: 10.18653/v1/N19-1423.

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

Gupta, M., Akiri, C., Aryal, K., Parker, E. and Praharaj, L. (2023) 'From ChatGPT to ThreatGPT: impact of generative AI in cybersecurity and privacy', *IEEE Access*, Vol. 11, pp.80218–80245.

Hu, S., Huang, T., Ilhan, F., Tekin, S. and Liu, L. (2023) 'Large language model-powered smart contract vulnerability detection: new perspectives', in *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, IEEE Computer Society, Los Alamitos, CA, USA, November, pp.297–306.

Kouliaridis, V., Karopoulos, G. and Kambourakis, G. (2023) 'Assessing the security and privacy of Android official ID wallet apps', *Information*, Vol. 14, No. 8, p.457.

Kouliaridis, V., Karopoulos, G. and Kambourakis, G. (2025) 'Assessing the effectiveness of LLMs in Android application vulnerability analysis', in Meng, W., Yung, M. and Shao, J. (Eds.): *Attacks and Defenses for the Internet-of-Things, Lecture Notes in Computer Science*, Springer Nature Switzerland, Cham, Vol. 15397, pp.139–154.

Lian, W., Goodson, B., Wang, G., Pentland, E., Cook, A., Vong, C. and Teknium (2023) *Mistralorca: Mistral-7b Model Instruct-Tuned on Filtered OpenOrCAV1 GPT-4 Dataset*, 20 August 2024.

Liu, J., Xia, C.S., Wang, Y. and Zhang, L. (2023a) 'Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation', in Oh, A., Naumann, U., Globerson, A., Saenko, K., Hardt, M. and Levine, S. (Eds.): *Advances in Neural Information Processing Systems*, Curran Associates, Inc., Red Hook, NY, Vol. 36, pp.21558–21572.

Liu, Z., Qian, P., Yang, J., Liu, L., Xu, X., He, Q. and Zhang, X. (2023b) 'Rethinking smart contract fuzzing: fuzzing with invocation ordering and important branch revisiting', *IEEE Transactions on Information Forensics and Security*, Vol. 18, pp.1237–1251.

Liu, Z., Liao, Q., Gu, W. and Gao, C. (2023c) 'Software vulnerability detection with GPT and in-context learning', in *2023 8th International Conference on Data Science in Cyberspace (DSC)*, pp.229–236.

Liu, P., Sun, C., Zheng, Y., Feng, X., Qin, C., Wang, Y., Li, Z. and Sun, L. (2023d) *Harnessing the Power of LLM to Support Binary Taint Analysis*, arXiv preprint arXiv:2310.08275.

Llama-3.3-70B-Instruct (2025) [online] https://huggingface.co/meta-Llama/Llama-3.3-70B-Instruct (accessed 9 June 2025).

LlamaIndex (2024) [online] https://docs.Llamaindex.ai/en/stable/ (accessed 20 August 2024).

Longpre, S., Hou, L., Vu, T., Webson, A., Chung, H.W., Tay, Y., Zhou, D., Le, Q.V., Zoph, B., Wei, J. and Roberts, A. (2023) 'The FLAN collection: designing data and methods for effective instruction tuning', in Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S. and Scarlett, J. (Eds.): *Proceedings of the 40th International Conference on Machine Learning, Proceedings of Machine Learning Research*, PMLR, Honolulu, Hawaii, USA, Vol. 202, pp.22631–22648.

Longueville, B., Sanchez, I., Kazakova, S., Luoni, S., Zaro, F., Daskalaki, K. and Inchingolo, M. (2025) 'The proof is in the eating: lessons learnt from one year of generative AI adoption in a science-for-policy organisation', *AI*, June, Vol. 6, p.128.

MobSF (2024) [online] https://github.com/MobSF/Mobile-Security-Framework-MobSF (accessed 20 August 2024).

MobSFscan (2024) [online] https://github.com/MobSF/MobSFscan (accessed 20 August 2024).

Motlagh, F.N., Hajizadeh, M., Majd, M., Najafi, P., Cheng, F. and Meinel, C. (2024) *Large Language Models in Cybersecurity: State-of-the-Art*, arXiv preprint arXiv:2402.00891.

Mukherjee, S., Mitra, A., Jawahar, G., Agarwal, S., Palangi, H. and Awadallah, A. (2023) *ORCA: Progressive Learning from Complex Explanation Traces of GPT-4*, arXiv preprint arXiv:2306.02707.

Noever, D. (2023) *Can Large Language Models Find and Fix Vulnerable Software?*, arXiv preprint arXiv:2308.10345.

Nous Hermes 2 Mixtral 8x7b DPO (2024) [online] https://huggingface.co/NousResearch/Nous-Hermes-2-Mixtral-8x7B-DPO (accessed 20 August 2024).

OpenAI (2024a) *GPT-4 is OpenAI's Most Advanced System, Producing Safer and More Useful Responses* (accessed 20 August 2024).

OpenAI (2024b) *GPT-4 Technical Report*.

OWASP (2016) *Top 10 Mobile Risks – Final List 2016* [online] https://owasp.org/www-project-mobile-top-10/2016-risks/ (accessed 28 May 2025).

OWASP (2024) *Mobile Top 10 2024: Final Release Updates* [online] https://owasp.org/www-project-mobile-top-10/ (accessed 28 May 2025).

Purba, M.D., Ghosh, A., Radford, B.J. and Chu, B. (2023) 'Software vulnerability detection using large language models', in *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp.112–119.

Qwen2.5-Coder (2025) [online] https://github.com/QwenLM/Qwen2.5-Coder (accessed 9 June 2025).

Sandoval, G., Pearce, H., Nys, T., Karri, R., Garg, S. and Dolan-Gavitt, B. (2023) 'Lost at C: a user study on the security implications of large language model code assistants', in *32nd USENIX Security Symposium (USENIX Security 23)*, USENIX Association, Anaheim, CA, August, pp.2205–2222.

Security Guidelines (2024) [online] https://developer.android.com/privacy-and-security/security-tips (accessed 20 August 2024).

Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Piras, L. and Petrovski, A. (2023) 'Labelled vulnerability dataset on Android source code (LVDAndro) to develop AI-based code vulnerability detection models', in *Proceedings of the 20th International Conference on Security and Cryptography – SECRYPT*, INSTICC, SciTePress, pp.659–666.

Solaiman, I., Brundage, M., Clark, J., Askell, A., Herbert-Voss, A., Wu, J., Radford, A., Krueger, G., Kim, J.W., Kreps, S., McCain, M., Newhouse, A., Blazakis, J., McGuffie, K. and Wang, J. (2019) *Release Strategies and the Social Impacts of Language Models*, Technical Report, OpenAI.

Sonatype (2023) *The Evolution of Software Supply Chain Attacks* [online] https://www.sonatype.com/resources/whitepapers/2023-evolution-of-ssc-attacks (accessed 10 July 2025).

Sun, Y., Wu, D., Xue, Y., Liu, H., Ma, W., Zhang, L., Shi, M. and Liu, Y. (2024) *LLM4VULN: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning*, arXiv preprint arXiv:2401.16185.

Thapa, C., Jang, S.I., Ahmed, M.E., Camtepe, S., Pieprzyk, J. and Nepal, S. (2022) 'Transformer-based language models for software vulnerability detection', in *Proceedings of the 38th Annual Computer Security Applications Conference, ACSAC '22*, Association for Computing Machinery, New York, NY, USA, pp.481–496.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C.C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P.S., Lachaux, M-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E.M., Subramanian, R., Tan, X.E., Tang, B., Taylor, R., Williams, A., Kuan, J.X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S. and Scialom, T. (2023) *Llama 2: Open Foundation and Fine-Tuned Chat Models*, Technical Report, Meta AI [online] https://arxiv.org (accessed 19 June 2026).

Tunstall, L., Beeching, E., Lambert, N., Rajani, N., Rasul, K., Belkada, Y., Huang, S., von Werra, L., Fourrier, C., Habib, N., Sarrazin, N., Sanseviero, O., Rush, A.M. and Wolf, T. (2023) *Zephyr: Direct Distillation of LM Alignment*, arXiv preprint arXiv:2310.16944.

Vulcorpus (2024) [online] https://github.com/billkoul/vulcorpus-2024 (accessed 20 August 2024).

Wan, Y., Zhao, W., Zhang, H., Sui, Y., Xu, G. and Jin, H. (2022) 'What do they capture? A structural analysis of pre-trained language models for source code', in *Proceedings of the 44th International Conference on Software Engineering, ICSE '22*, Association for Computing Machinery, New York, NY, USA, pp.2377–2388.

Wang, J., Huang, Z., Liu, H., Yang, N. and Xiao, Y. (2023) *DefectHunter: A Novel LLM-Driven Boosted-Conformer-based Code Vulnerability Detection Mechanism*, arXiv preprint arXiv:2309.15324.

Yang, X., Zhu, W., Pacheco, M., Zhou, J., Wang, S., Hu, X. and Liu, K. (2025) 'Code change intention, development artifact, and history vulnerability: putting them together for vulnerability fix detection by LLM', *Proc. ACM Softw. Eng.*, June, Vol. 2.

Yao, Y., Duan, J., Xu, K., Cai, Y., Sun, Z. and Zhang, Y. (2024) 'A survey on large language model (LLM) security and privacy: the good, the bad, and the ugly', *High-Confidence Computing*, Vol. 4, No. 2, p.100211.