SoK: Large Language models in Security Code Review and Testing

Reply to Reviewers' Comments

We would like to express our sincere gratitude to the editor and reviewers for their valuable comments and thorough insights. We have made changes in the paper based on the suggestions of the reviewers. We believe that the revised version addresses the concerns of the reviewers, and the quality and presentation of our paper have improved significantly.

To simplify the identification of the reviews, we've established the following colour-coding scheme:

- Reviewer 1 (qTtP) = cyan
- Reviewer 2 (PBsR) = teal
- Reviewer 3 (9e7b) = orange

We have also made improvements throughout the document. These mainly include grammatical changes and improvements to readability. As there were many changes in Section 7, we have marked bigger changes in red to ensure visibility.

We structured this response letter to firstly address the main comments from all reviewers. Afterwards, we go into the specific comments by restating the reviewers' comments and then present our responses for each of them.

Contents

1 All reviewers

1.1 Methodology

We restructured the entire Section 2 (Methodology) to provide a more detailed view of how we selected the papers. We added a complete list of the search strings, the used search databases and a visualisation of the paper selection process. Moreover, Table 1 was remodelled to give additional insights by making the chosen LLMs better comparable and by providing a summary of techniques. With these changes, we hope that we have addressed the feedback correctly and that our methodology is now clearly explained.

1.2 Alignment to a SoK

To address the feedback for a better alignment with a SoK, we remodelled Section 8 (Discussion). It now includes a more cohesive discussion about the different topics in connection with the research questions. Also, we made our opinions, conclusions and key takeaways clearer. The same applies to Section 9 (Conclusion), in which we now also describe our main points from the discussion.

The structure of our paper remains the same. We think that by providing a more neutral reporting in the previous sections and a more detailed discussion in Section 8, the difference between the research results and our drawn conclusions is more visible to the reader.

Overall, large parts of Sections 8 and 9 were modified to address this comment.

2 Reviewer 1 (qTtP)

2.1 Comment 1

Comment: Another concern with the presented paper is the methodology used to compare results. As an example, in relation to Figure 3 data from Purba et al. as well as Tamberg and Bahsi is compared. While the paper explains how the differing results may have been achieved, it makes no effort to explain weather this comparison makes sense, i.e. are they testing the same code? Are the models provided with the same context? An improvement could be made by already discussing these concerns in the paper.
Response: This point was not covered in our first submission, and we agree that it is important to talk about. We thus dedicated a paragraph in Section 2 to this comment.

> A practical challenge remains in comparing the selected papers as a single group. The foundation of the relevant research outcomes differs fundamentally. Often, different LLMs, testing datasets, techniques, and approaches are used, making a direct link between those selected papers difficult. However, we are interested in their outcomes and key findings. Having a broader view of the performance results of the different papers makes it possible to draw conclusions about the current state of the research and what challenges the research is facing. We thus cannot provide a direct comparison between the papers, but we can provide a high-level comparison by analysing the outcomes of the individual papers together.

2.2 Comment 2

Comment: Section 2 states the goal to allow readers to make comparisons over used LLMs. This does not appear again as an item in the discussion or conclusion, where only general advice is

given without references to specific LLMs. A short paragraph could be added explaining what the results of this comparison are, maybe tying in with the conclusions suggested above.
Response: We have now dedicated Section 8.2 (RQ2: What is the impact of the LLM on the performance?) for this feedback. However, like in the previous comment, a direct recommendation from our side for specific LLMs is difficult to make. Again, this is mainly because of the non-uniform benchmarking processes in the research papers. However, we could identify common patterns which can lead to better-performing LLMs and are discussing those points in the mentioned section.

2.3 Comment 3

Comment: A final concern is the long-term value of the provided paper. Given the fast-moving research in AI, how long until the results are outdated? The paper acknowledges this problem in the related work section as its own criteria to omit older papers, but what about the reviewed paper itself? Maybe a short paragraph explaining potential risks could be an improvement.
Response: We added a paragraph in Section 1 describing why we think that our paper has long-term value. While we also think that the measurements will be outdated as soon as a new version of LLMs is introduced, we think that key concepts will still be needed to improve the performance of LLMs.

> The research in this field is progressing rapidly. Each new version of an LLM can bring improvements in performance. This makes it particularly challenging to provide long-term value within this paper, since the current measurements are already outdated with the next iteration. However, we think that the currently used techniques and application areas will continue to exist in the same or a similar form. In other words, the identified research vectors will be valuable in the future, as well. Therefore, well-functioning concepts should continue to be used in the future to obtain better results from LLMs. Also, circumventing the technical limitations will still be valid for some time. In light of these points, we believe that this paper will provide long-term value for the research community as well as practitioners in the industry.

2.4 Comment 4

Comment: The introduction mentions some common security vulnerabilities like Log4j. These examples are never brought up again. Is there evidence that LLM-based tools could have helped in these cases?
Response: This introduction aimed to highlight the importance of software security by showing some examples from the past. Eventually, we are not able to answer this question and cannot state whether LLMs in their current form would have been able to identify vulnerabilities with this level of complexity. For the revised version, we decided to exclude those examples to reduce exaggeration.

2.5 Comment 5

Comment: In Tamberg and Bahsi the conclusion suggests that LLMs outperform traditional tools, yet your key takeaways state that traditional tools - had an overall better performance. Is there an explanation for this disparity?

Response: We think that our conclusion does not contradict the conclusion from Tamberg and Bahsi. However, the focus of our conclusion was more on the practical application of those tools. While LLMs achieve better scores in the usual metrics, a key disadvantage is the production of higher false positive rates, which means additional work for developers. Thus, we decided to stick with our conclusion but to include a paragraph in Section 4.1.3 giving more information on our view.

> Overall, one can say that traditional tools focus on keeping the false positives low by compromising on false negatives. However, this approach allows developers to focus on relevant findings without losing time on false positives. In contrast, LLMs are finding more true positives but with more false positives, handing over the task to developers to filter them out.

2.6 Comment 6

Comment: "Eventually, it will maybe find an exploit." The wording here is unusual.
Response: We changed the wording.

> With this approach, the function may be able to find a vulnerability [61].

2.7 Comment 7

Comment: Some of the spacing for the vertical headers in tables looks off. Some headlines are too close to the table borders. This appears in both Table 1 and Table 2, i.e. "Topic" in Table 1.
Response: The tables mentioned in the comment were changed or removed to comply with this and other feedback. The style of the tables should now be correct.

3 Reviewer 2 (PBsR)

3.1 Comment 1

Comment: The authors did do a literature search on Google Scholar and some indexes provided by ACM, IEEE, and Springer, as described on page three. I would argue that this can be used to identify candidate papers, but should be complemented with a search by recent PCs and augmented by citation data.
Response: We checked other sources and also included sources from pre-publishing venues like arXiv. Regarding the citation score, reputation, etc., we added an explanation in Section 2, explaining how we handled those metrics.

> We also looked at the reputation and citation score. However, since the topics all revolve around fast-moving research, we were willing to be accommodating in this aspect. This was particularly true when we could not find works with similar scope.

3.2 Comment 2

Comment: On page six, Section 5.1 uses a non-standard classification of fuzzing systems. The third item (greybox fuzzing) suggests the use of:

- blackbox
- whitebox

- graybox fuzzers. The second "category", however, is "mutation-based fuzzers", which is orthogonal to the blackbox/whitebox/graybox classification scheme. More of a systematic approach could be used, maybe even with an illustration where an LLM is used by related work (and, similarly, where LLMs are not *yet* used in fuzzing.)

Response: To be more aligned with the terms used, we adhered to the literature and revised the introductory section of Section 5.1 by adding a table discussing all classification types for a fuzzer. Additionally, we classified the research papers according to the table. However, we think that the remaining structure should be left since it represents the focus points of the researchers.

> A fuzzer can be viewed as a generator which creates random inputs [56]. Since the underlying implementation of the generator can vary a lot, different types of fuzzers exist in practice. In the work of Beaman et al. [7], a definition for the various types was created, depending on how advanced the fuzzer is. A fuzzer is mainly categorised by the knowledge it has about the function and the system under test, and also by how it generates the input and how it reaches the testing coverage. In Table 3, we give an overview of the different classification types and a short description based on the definitions from Beaman et al. [7]. Additionally, the table contains a categorisation of the fuzzing-related research we cover in this work.

3.3 Comment 3

Comment: Unidiomatic word choice:

- *multifarious* is literally the first time I have ever read this word.
- *systemised* is shown as a spelling error in all tools I checked, use *systematized*.
- *exploitables* is shown as a spelling error in all tools I checked, use *exploits*.
- *futuristic* approaches has the strong connotation of imagined things, i.e., derived from science-fiction. Such a meaning is clearly not intended, and should thus be called "recent", "modern", or "state-of-the-art".

Response: We changed the wording as follows:

- multifarious = various
- systemised = systematized
- exploitables = exploits
- futuristic = modern

3.4 Comment 4

Comment: Stylistic issues:

1. xz should be lower case and set using a teletype font.
2. Tables should use a booktabs layout.
3. The accepted phrasing of "related work" is not "related works", but "related work". (cf. Merriam-Webster's dictionary, or Oxford Learner's Dictionary).

4. Figures 2 and 6 should be made larger to simpler discern their contents. They also seem to be screenshots with ChatGPT prompts, so a representation w/o ChatGPT and using a listing style to separate the generic prompt template from the instantiated prompt would help readers.

5. Captions are proper grammatical units and thus should end in a "." period.

Response:

1. We changed the introduction part. Thus, we do not name xz anymore. We discuss this change in Section 2.4.

2. For Table 2, which is newly introduced in this revision, we used a booktabs layout. The table Section 8 is now integrated as text, and thus the table was removed. For Table 1, we decided to use a different style to keep a higher readability.

3. We changed the wording to "related work".

4. The figures should represent examples, to give a better understanding of how a prompt as well as an answer could look. We changed the styling to make it better readable, but also to suit our other graphics.

5. We added a period in the captions.

3.5 Comment 5

Comment: Page 3, left column, itemization item 3: Please simplify this sentence to make it easier for readers.
Response: We restructured Section 2 completly. Thus, the wording is now different.

4 Reviewer 3 (9e7b)

4.1 Comment 1

Comment: The introduction states that software security is a new research field for LLMs. I disagree with this claim, as security researchers have been exploring the use of LLMs for some time, much like in the field of software engineering. I suggest rephrasing this statement to avoid overstating the novelty.
Response: We changed the corresponding part to reduce the overstating.

> Since the upcoming of LLMs, interest has risen in using them to help developers achieve secure software.

4.2 Comment 2

Comment: The title of Section 4 refers to "Static" code vulnerability detection, but the content of the section does not specifically focus on static analysis for identifying vulnerabilities. Instead, it discusses automated techniques more broadly, some of which may fall under dynamic analysis.
Response: In the context of this paper, the word “static" should refer to the approach used for code vulnerability detection. Since all of the considered research is focusing on non-running code, we decided to specify this in the title. However, we never introduced this definition properly. Thus, we added a paragraph in Section 4 addressing this issue:

There are two different types of automated tools. The static approach (Static Application Security Testing, SAST in short) is applied to the source code, while the dynamic approach (Dynamic Application Security Testing, DAST in short) is applied to the compiled and running application [11]. In this section, we will focus on techniques for static code vulnerability detection.

4.3 Comment 3

Comment: The LLM acronym is introduced multiple times in the paper.
Response: We limited the introduction of the acronym to only two occurrences.

4.4 Comment 4

Comment: Can you elaborate on your paper selection and filtering strategy?
Response: The methodology part was completely restructured and added with additional content. We think that Section 2 now suits the feedback we received.

4.5 Comment 5

Comment: What key insights or takeaways should readers gain?
Response: The discussion section was extended by additional insights and conclusions. Thus, Section 8 should now comply with the feedback.

SoK: Large Language Models in Security Code Review and Testing

Anonymous authors
Paper under double-blind review

Abstract

In this paper, we present and discuss practical applications of Large Language Models (LLMs) in software security, concretely in code vulnerability detection, fuzz testing and exploit generation. Measurements of various research outcomes are analysed to answer questions about the performance of LLM in those fields, including a comparison with tools following traditional approaches. In addition, the drawbacks and a future overlook with a delineation of technical challenges are given. Challenges are found in the cost- and time-intensive training of LLM, the limited context-length understanding of program code, the high false positive rate because of hallucinations, and keeping the data up-to-date so that definitions of newly detected vulnerabilities are contained.

1 Introduction

Secure software development is an important topic, as vulnerabilities can impair applications and services, which have become the foundational pillars of our daily lives. Moreover, many critical services are based on software-intensive infrastructure where confidentiality, integrity, and availability, i.e. CIA triad, are key requirements. Therefore, software security is an omnipresent concern that can lead to incidents and breaches when done improperly.

Software development goes through different phases until the finished product is established. On top of that, every phase has different security aspects that need to be considered. Various methodologies provide a structured approach to secure software development. For instance, the SecDevOps lifecycle gives an overview of security topics that should be checked during the development and the operation of software [13]. Similarly, the Secure Software Development Lifecycle (SSDLC) also targets security but with a focus on software development [14]. Eventually, all these methodologies and frameworks entail some common traits: Efficient and diligent vulnerability detection and testing are crucial.

Since the emergence of LLMs, interest has risen in using them to help developers achieve secure software. The focus of interest for LLMs comes from its architecture, which was first introduced in 2017 by Vaswani et al. [49], in the current form. The architecture, consisting of an encoder and decoder, uses a so-called attention mechanism, allowing the LLM to focus on relevant input sequences. As for training, enormous datasets are used, covering many different topics. The aim is to create foundation models, which can be used in various ways [3]. For example, LLMs are already used on different topics, ranging from medicine to education, and also in software engineering [25], while software security represents a new research field.

In this paper, we provide a systematized description of LLM's role in software security, namely for security code review and testing. We present a high-level introduction to LLMs, including a technical description, a definition of the terminologies used, and LLM training methods (Section 3). The focus relies on secure implementation and security testing of software. Firstly, we elaborate on automated static code vulnerability detection in Section 4. It is an ideal entry point for a first security check before merging the code with the codebase in software development. Another technique for finding vulnerabilities is fuzz testing (Section 5). Here, we focus on the creation and mutation of fuzzer input while also including the generation of fuzz driver as a second subtopic. In Section 6, we also want to show how these detected vulnerabilities can be converted to exploits by using an LLM as an automated exploit generator. Lastly, we discuss our findings per our research questions, delineate the challenges of implementing LLM approaches in this domain, and give an outlook on what future developments could look like (Section 7).

Research questions - We build our systemisation of knowledge for our scope around the following research questions in this work:

1. What are practical use cases for LLMs in security code review and testing, and how do they perform against traditional tools?
2. What is the impact of the LLM on the performance?
3. What do current approaches look like?
4. What are the current challenges and what are the prospects for LLMs in security code review and testing?

2 Methodology

In this paper, we focus on technical works that utilise LLMs in software security. We defined a process to search for and select suitable research papers and outlined the criteria by using the proposal of van Wee and Banister [48] as an inspiration

Topic	Author	Description	LLM					Techniques			
			BERT	Davinci	Llama	OpenAI	Other	Tuning	Prompt	Tool	Dataset
C	Cheshkov et al. [10] in 2023	Cheshkov et al. created a performance comparison using GPT models from OpenAI in code vulnerability detection by categorising the vulnerability with a binary and multi-label classification approach.		x		x			x		x
C, E	Fu et al. [20] in 2023	Here, the whole lifecycle was considered. This includes tasks like vulnerability detection and its classification, a risk assessment and a proposal on how to mitigate the security risk.	x			x		x	x		x
C	Guo et al. [24] in 2024	Different LLMs with varying training backgrounds were chosen to conduct a comparison in a binary classification task. Six LLMs are especially trained for vulnerability detection, while the other six LLMs were only fine-tuned or taken as is without special training.	x		x	x	x	x	x		x
C	Purba et al. [41] in 2023	They compared different LLMs and traditional tools by using two different datasets to see whether the vulnerability is detected or not.		x		x	x	x	x		x
C	Tamberg and Bahsi [45] in 2025	Tamberg and Bahsi analysed the use of LLMs in code vulnerability detection by testing different prompt strategies and comparing the results with the performance of traditional tools.				x	x		x		x
C	Yin et al. [54] in 2024	Yin et al. not only considered the vulnerability task, but they also researched how capable LLMs are when it comes to detection, risk assessment, location and reporting of the vulnerability.	x		x		x	x	x		x
C	Yu et al. [55] in 2024	Yu et al. applied five different prompts and evaluated which of them led to the best performance, also in comparison to traditional tools.			x	x	x		x		x
F	Black et al. [8] in 2024	They analysed the effectiveness of LLM in seed generation in combination with the existing fuzzer Atheris, especially for the programming language Python.				x	x		x	x	
F	Tamminga [46] in 2023	Tamminga focused on an approach for using an LLM as a seed generator in combination with traditional fuzzers like AFL++ and libFuzzer. While focusing on the programming language Go, a priority was placed on interoperability between different programming languages.				x	x	x	x	x	x
F	Xia et al. [52] in 2024	Xia et al. show a practical implementation for a mutation-based fuzzer, which is called Fuzz4All.				x	x		x	x	
F	Zhang et al. [59] in 2024	They created a tool, called LLAMAFUZZ, which can be used to enhance greybox fuzzing.			x			x	x	x	x
F	Zhang et al. [58] in 2024	Zhang et al. are showing how an LLM can be used in fuzz driver creation.			x	x	x		x		x
E	Fang et al. [18] in 2024	The focus is on exploit generation for one-day vulnerabilities, using LLMs.			x	x	x		x		x
E	Zhang et al. [60] in 2023	The topic is about exploit generation by using an LLM. It focuses on the use case of dependency vulnerability alerts and the diminishing of false positives. The result is then compared with traditional tools.				x			x		x
E	Zhou et al. [61] in 2024	Zhou et al. present a tool called Magneto, which uses fuzzing techniques to exploit unpatched vulnerabilities from third-party dependencies.				x			x	x	x
O	Jiang et al. [29] in 2024	Here, the challenges as well as recommendations are considered. They focus on research done in LLM-based fuzzing.									
O	Kaddour et al. [30] in 2023	Kaddour et al. give a general overview of current challenges when applying LLMs in practical fields.									

Table 1: Overview of the related work.

for our methodology. Our focus topics are "code vulnerability detection (C)", "fuzz testing (F)" and "exploit generation (E)". To ensure readability, we will use the abbreviations defined in the brackets.

In the first step, we defined search keywords to use within the databases. A complete list of the used keywords can be found in Table 2, which depicts the topic and the corresponding search terms. The search terms are primarily structured based on the topic, including the keyword "LLM".

Topic	Search String
C	ChatGPT for Code Vulnerability Detection
C	LLM for Code Vulnerability Detection
C	LLM for Software Vulnerability Detection
C	LLM for Security Code Review
F	LLM in Fuzz Testing
F	LLM in Fuzz Driver Generation
F	LLM for Seed Generation in Fuzzing
E	Exploit Generation with LLM in Software Development

Table 2: Search terms.

Secondly, we applied those keywords to academic publishing venues and meta-search engines to find relevant papers. Concretely, we used the following databases:

- ACM [1]
- arXiv [5]
- Elsevier [17]
- Google Scholar [22]
- IEEE [28]
- Springer [43]
- Wiley [51]

Lastly, we applied our selection process, presented in Figure 1. We started by making a short metadata review. We also looked at the reputation and citation score. However, since the topics all revolve around fast-moving research, we were willing to be accommodating in this aspect. This was particularly true when we could not find works with similar scope. Afterwards, we checked important text passages like the abstract, discussion and conclusion for a first assessment of the topic coverage. In this step, we made sure that the content of the paper was aligned with our research questions and that the LLM was a main component. The next step was skimming over the paper. Since we were interested in practical examples, we excluded papers with only theoretical coverage. Additionally, we wanted to make sure that the papers contained valuable insights for us, especially by covering benchmarks and comparisons with traditional tools. In a detailed look, we paid attention to the LLM techniques used and also excluded duplicates or papers covering similar topics without additional insights. Lastly, we performed snowballing by analysing the references used by the papers to identify potential new candidate papers.

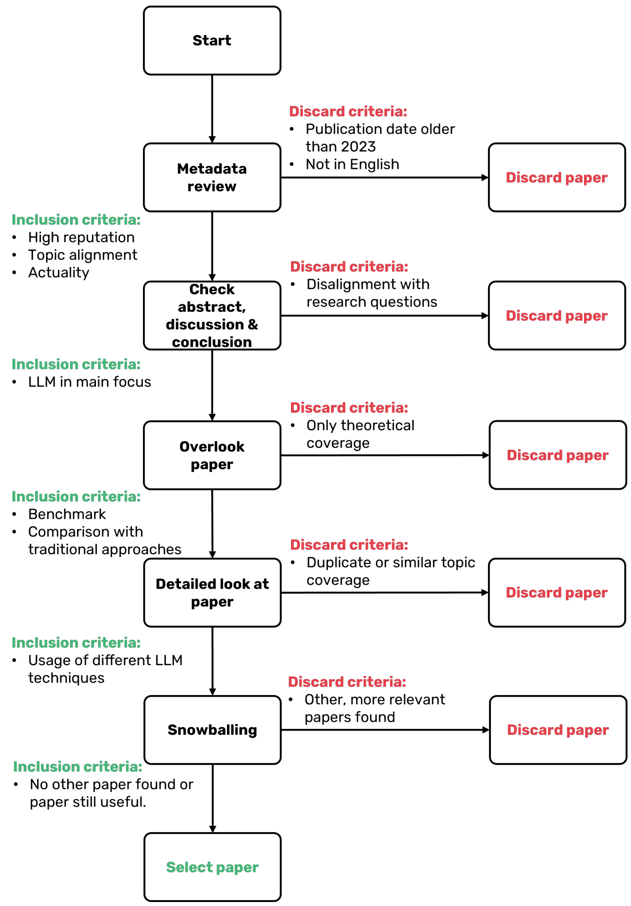

Figure 1: Paper selection process (adapted from [48]).

In addition, survey papers were examined to gain an overview of current works and the technical domain. The listed papers were also post-filtered manually to avoid any duplication, misselection, and quality issues. At the end, 17 papers were selected. Please note that our work is not an exhaustive literature survey paper but a Systemisation of Knowledge (SoK) paper presenting a concise and structured analysis of a focused scope. Ultimately, the scope includes LLMs for code vulnerability detection and testing, including fuzzing and exploit generation. A practical challenge remains in comparing the selected papers as a single group. The foundation of the relevant research outcomes differs fundamentally. Often, different LLMs, testing datasets, techniques, and approaches are used, making a direct link between those selected papers difficult. However, we are interested in their outcomes and key findings. Having a broader view of the performance results of the different papers makes it possible to draw conclusions about the current state of the research and what challenges the research is facing. We thus cannot provide a direct comparison between the papers, but we can provide a high-level comparison by analysing the outcomes of the individual papers together.

In Table 1, an outline of those surveyed papers is given. We categorised them into the three aforementioned categories and the additional topic “challenges and future outlook (O)". The final category was facilitated to provide a discussion on potential future technical research and development directions. The table consists of metadata information like author, year and a short description. Further, an overview is provided about the used LLMs in those works as well as the applied techniques, as listed below:

- Tuning: The paper includes fine-tuning mechanisms.
- Prompt: The paper applies prompt engineering techniques.
- Tool: The paper introduces a tool for a specific task with an advanced architecture, in which the LLM plays a significant role.
- Dataset: The paper creates or introduces a dataset that can be used for training or testing.

The research in this field is progressing rapidly. Each new version of an LLM can bring improvements in performance. This makes it particularly challenging to provide long-term value within this paper, since the current measurements are already outdated with the next iteration. However, we think that the currently used techniques and application areas will continue to exist in the same or a similar form. In other words, the identified research vectors will be valuable in the future, as well. Therefore, well-functioning concepts should continue to be used in the future to obtain better results from LLMs. Also, circumventing the technical limitations will still be valid for some time. In light of these points, we believe that this paper will provide long-term value for the research community as well as practitioners in the industry.

3 Large Language Models (LLMs)

The task of a language model is to predict and generate language. To do that, the likelihood of the next upcoming word needs to be calculated [2]. To illustrate, if we consider the sentence “I need an umbrella because it is ...” the next best-guessed word could be “raining”. There are different approaches and concepts for constructing a language model [2]. In the beginning, statistical language models were used, which are based on calculations made on text-containing datasets. One implementation is the n-gram language model, which predicts the next word based on the previous n-1 words [2, 3]. After the introduction and the rise in popularity of neural networks, the underlying technology in language models changed. With a neural network, one could improve its parameters to get optimised outputs by applying training methods using training datasets [2, 3].

A step forward was achieved with the transformer architecture, which was introduced in 2017 by Vaswani et al. [49]. This architecture, which is based on a deep neural network, enabled the creation of LLMs [2]. The name affix “large” comes from the count of parameters or the size of the used training dataset [23]. For example, Llama 2 has 70 billion parameters and used 10 TB of text for training, according to Karpathy [33]. The architecture builds on a so-called attention mechanism, which uses weights to distinguish the vital parts from the input [2]. It consists of an encoder and decoder, but some approaches use only one of the two parts [3]. The encoder processes the input and tries to understand it by depicting it in a suitable format. The decoder, on the other hand, is responsible for generating the result by taking the encoder's output as input [3].

In Figure 2, the LLM architecture, as well as the training steps before it can be used by a user, are visualised. The initial training of an LLM is called pre-training, and it is cost- and time-intensive [33]. The reason relies on the training process itself, which requires the gathering of a lot of information and the calculation of the parameters. For example, Llama 4 Maverick [37] has a total of 400 billion parameters, while DeepSeek-V3 [15] has a total of 671 billion parameters. For this reason, pre-trained, foundation LLMs are used as a base and, if needed, adjusted via fine-tuning [3].

The fine-tuning process starts with the gathering of labelled datasets. This training data usually contains examples similar to the data used for the classification task in production. In the next step, the labelled training data is used to fine-tune the model. As a result, an adjusted model is obtained. Fine-tuning is an iterative process. As soon as the productive model is rolled out, logs should be gathered to correct anomalies by applying the described process again [33].

Another adjustment technique is prompt engineering. It focuses on the input, which gets passed to the LLM. Various patterns can be used so that the LLM generates output within the boundaries given by the patterns. Sahoo et al. [42] created a survey, describing common prompt patterns. Creating prompts without further refinements is called zero-shot prompting. In few-shot prompting, examples are included, intending to give the LLM a clearer instruction. A different approach is the so-called chain-of-thought prompting. Here, the LLM is guided on how to calculate the result, such that it shows its calculation steps. Auxiliary, many other abbreviations exist using similar ideas [42].

4 Automated static code vulnerability detection

Code review is a technique used in software development, where the code gets reviewed by a second person before it is merged into the productive codebase [6]. There are different reasons to conduct code reviews, some of which are shown in

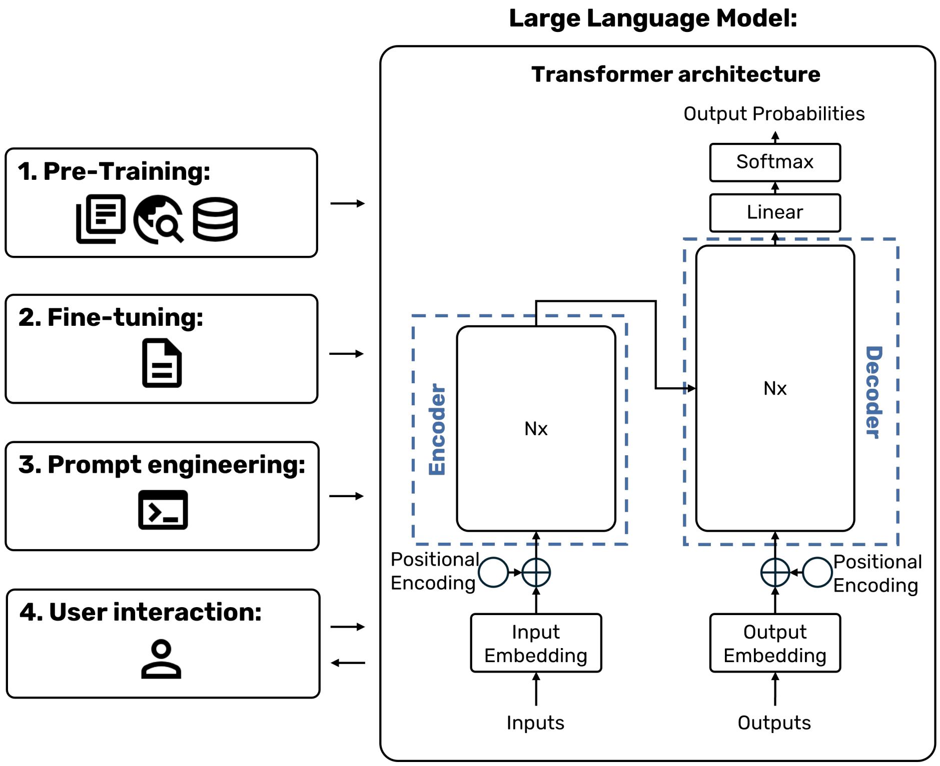

Figure 2: Architecture of LLMs and training methods (adapted from [3, 49]).

the study of Bacchelli and Bird [6]. In their survey, they show that the motivation for doing code reviews is to increase the overall code quality level of the codebase, find and eliminate bugs to reduce the error ratio and for know-how transfer. In this section, we will focus on security code review, which is a subcategory of error finding by aiming to detect and find security flaws in software [16].

Although there are benefits in conducting code reviews, they are not always carried out due to various factors. Codegrip [12] and Ghanbari et al. [21] have addressed the question of why code reviews are neglected. Both came up with similar reasons. One reason was the increased workload and time costs for carrying out a code review. A company might tend to leave out code reviews to reach certain aims and to increase the output. Another reason mentioned by both was motivation. The software development team could simply be disinterested in applying code reviews because of a lack of interest, not understanding the benefits, or having a false sense of risk [12, 21]. An additional reason given by Ghanbari et al. [21] was the technical complexity of the project environment, leading sometimes to negligence in applying code quality improvements. Bacchelli and Bird [6] supplement the list by adding the understanding of code changes to the challenges. In their interviews with software developers, they found out that the major challenge lies in understanding why the code change was made and what influence the change has on the functionality of the software.

Another aspect is the way code reviews are conducted. There are two types: manual and automatic code reviews [12, 16]. Although most companies favour manual code reviews [12], this is considered liable to errors, which is shown by the study of Edmundson et al. [16]. They measured the effectiveness of code vulnerability detection in manual security code reviews by interviewing software developers. On average, a software developer could find about a third of the known vulnerabilities. While this speaks in favour of using automated tools, just 27% of the surveyed companies in [12] are regularly using an automated code review tool. The reason lies in the missing know-how against such tools [12].

Because of those hindrances, new approaches are investigated to automate this process by using general-purpose LLMs. The benefit comes from the transformer architecture, which is trained on general data and therefore makes the LLM suitable for different tasks, with one of them being vulnerability detection in software [41]. There are two different types of automated tools. The static approach (Static Application Security Testing, SAST in short) is applied to the source code, while the dynamic approach (Dynamic Application Security Testing, DAST in short) is applied to the compiled and running application [11]. In this section, we will focus on techniques for static code vulnerability detection.

In Figure 3, an example is shown by using ChatGPT-4 [38]

User Prompt
Is there a software vulnerability? Answer only with yes or no, the corresponding code line and the vulnerability type.

```
#include <stdio.h>
int main(int argc, char **argv) {
  char buf[8];
  gets(buf);
  printf("%s\n", buf);
  return 0;
}
```

ChatGPT Answer
Yes, `gets(buf);` - Buffer Overflow

Figure 3: Interaction with ChatGPT giving the task to detect code vulnerabilities (adapted from [38, 39, 41]).

and a code example from OWASP [39]. The code example contains a buffer overflow vulnerability. In such a case, the application's memory is overwritten by exceeding the assigned memory, thereby causing unpredictable behaviour in the application [39]. The faulty line is the call to the method `gets()`, which is considered unsafe in C since it does not check the size of the buffer.

The task of the LLM is to notice the vulnerable code snippet. Including the code with a corresponding question to the prompt (similar to the research from Purba et al. [41]) leads to ChatGTP [38] detecting the vulnerability, pointing to the vulnerable code line and explaining why the code is viewed as unsafe.

4.1 Adapting LLMs for code vulnerability detection

In the research of Purba et al. [41], they compared different LLMs and applied vulnerable code to measure how effective LLMs are in noticing code vulnerabilities. Furthermore, they compared base models and fine-tuned models. The latter were trained with labelled data containing vulnerable and secure code examples. As for the testing dataset, they used code examples containing buffer overflow and SQL injection vulnerabilities [41].

Similar to Purba et al. [41], Guo et al. [24] tested the capability of LLMs in the binary classification task with a similar prompt. As a difference, they compared the performance of differently trained LLMs. They included general-purpose LLMs, self-fine-tuned LLMs and open-source LLMs that were already trained for code vulnerability detection tasks [24].

Cheshkov et al. [10] also evaluated how well GPT models perform in vulnerability detection. Like the previous two approaches [24, 41], they performed a binary classification but also added a performance measurement for a multi-label classifier. The multi-label classification was done by providing five different CWE vulnerability types and designing a prompt asking the GPT model if one of those five vulnerabilities is included in the provided code snippet [10].

Another technique that can influence the results of an LLM is prompt engineering. Thus, Yu et al. [55] designed five prompts and tested their effectiveness. They included an instruction and modified the prompt by adding or removing additional information, like project information or CWE descriptions and using techniques like chain of thought. Tamberg and Bahsi [45] also followed the approach of testing different prompt engineering approaches by applying 23 different prompts inspired by related work.

Yin et al. [54] not only discussed whether LLMs can detect vulnerabilities, but they also investigated whether LLMs are capable of finding the specific affected code location, disclosing why it is seen as a vulnerability and estimating the risk coming from the discovered vulnerabilities. They tested the performance of different base and fine-tuned LLMs by using public datasets. As for prompt engineering, a few-shot approach was chosen. The prompt contains a task description similar to the ones already seen, the code under test, and an indicator defining one of the four mentioned tasks [54].

Fu et al. [20] took this approach further and included the whole lifecycle in their research. They measured the capability of GPT models to detect vulnerabilities, but also to classify them. Moreover, the GPT models were tasked with evaluating the severity of the detected vulnerability and proposing a mitigation [20].

4.2 Results

In the results of Purba et al. [41], Davinci, with fine-tuning, achieved the best score across all models considered. Nevertheless, it had an F1 score of 73.2% with a recall of 94% and a precision score of 60%, indicating that there is a high false positive rate (FPR). Similarly, all of the compared LLM models suffered from a high FPR. In contrast, the false negative rate (FNR) of the Davinci model was low at 6%. In the work from Cheshkov et al. [10], the binary classifier also had a high FPR, while the multi-label classifier led to a lower F1 score and a lower precision and recall score. Thus, that work did not perform well for both classifiers.

Guo et al. [24] made two key findings: Firstly, LLMs perform well on known vulnerabilities seen through the training dataset, but are limited in the generalisation of their learned knowledge. Secondly, fine-tuning enables smaller LLMs to be better than larger LLMs in certain tasks. However, a problem encountered during training, which could also affect the results of other research, was the inaccuracy of the dataset [24].

As for the prompt, Yu et al. [55] observed that the prompt with an instruction and containing specific information about the CWEs performed the best. Tamberg and Bahsi [45], who also made a prompt-based approach, concluded that different models react differently to the prompt. For GPT-4 Turbo, the best result could be reached with a dataflow analysis prompt. This prompt includes a task description, which demands an analysis of the data flow within the provided source code and a template on how to answer. After receiving the answer, a second and a third prompt were added, in which the LLM is asked to review and improve its answer. For GPT-4 and Claude 3 Opus, the highest result was reached with a chain of thought prompt. Here, the process of how to approach the problem step by step was described, so that the LLM can follow this manual [45].

Yin et al. [54] concluded that there is potential for LLMs in the covered tasks, but they still need development. In the analysis from Fu et al. [20], they deduced that base models of ChatGPT are not suitable for use in all four observed tasks because of their poor performance.

4.3 Comparison with traditional tools

Contrary to LLM, Purba et al. [41] and Tamberg and Bahsi [45] included an overview of the performance of traditional tools executing static code analysis. The traditional tools work by using syntactic and semantic checks. For example, a rule set for a syntactic check could contain a list of different vulnerable functions, such as the mentioned `gets()` function in Figure 3 [35]. To detect intricate vulnerabilities, semantic checks are necessary. Here, the code base gets transformed to an enhanced control flow representation, allowing for a more sophisticated vulnerability detection approach [35].

In the research of Purba et al. [41], the tool Checkmarx[1] performed the best among the traditional tools with an F1 score of 47.3%. Compared to LLMs, this tool keeps a lower FPR at 43.1% but has a higher false negative rate (FNR) at 41.1% [41].

Tamberg and Bahsi [45] came to a similar conclusion regarding the FPR. However, their model reached a higher precision with a lower recall compared to the Davinci model from Purba et al. [41]. One explanation for this outcome could be the overall performance gain with newer models since [45] was published in 2025 using GPT-4 while [41] was published in 2023 using GPT-3.5-Turbo. However, the reason could also rely on the usage of different prompts, fine-tuning strategies or the dataset used for the benchmarking.

A performance comparison from Purba et al. [41] and Tamberg and Bahsi [45] can be seen in Figure 4. Overall, one can say that traditional tools focus on keeping the false positives low by compromising on false negatives. However, this approach allows developers to focus on relevant findings without losing time on false positives. In contrast, LLMs are finding more true positives but with more false positives, handing over the task to developers to filter them out.

[1] Checkmarx: https://checkmarx.com/

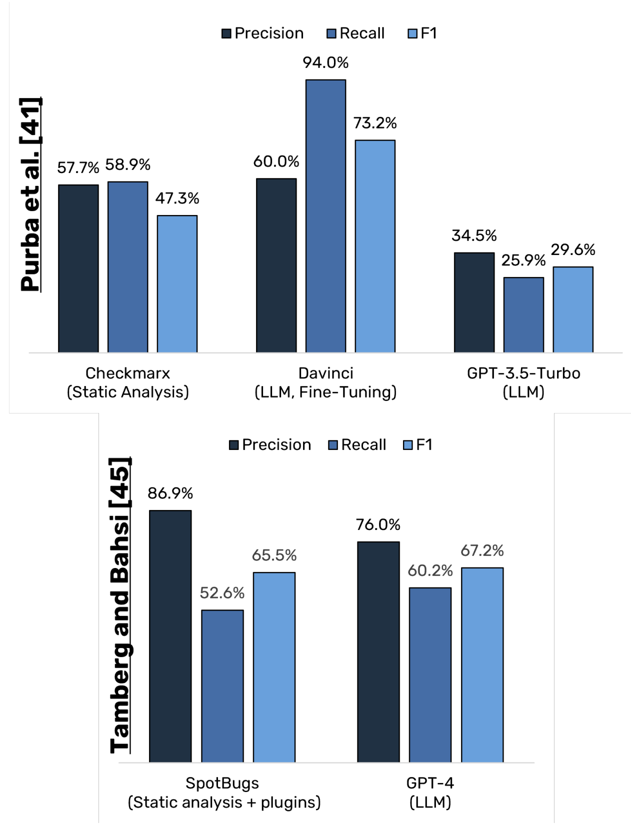

Figure 4: Performance comparison of different code vulnerability detection tools (adapted from [41, 45]).

5 Fuzz testing

Fuzz testing describes the method of using randomised input to test how a function reacts to it. The intention is to observe unusual behaviour and thereby to detect code flaws and potential vulnerabilities [56]. While it is considered effective in discovering software vulnerabilities, different hindrances prevent this technique's use in the industry.

Firstly, the complexity of the environment setup needs to be considered. Fuzzers have different requirements before they can be applied. Since an existing environment uses various technologies, including operating systems, programming languages and external libraries, it is difficult to adapt it to a fuzzer.

Secondly, fuzz driver implementation is challenging. A fuzz driver describes the link between the test function and the API. Thus, software developers need to know how the software works in technical and functional detail so that they can write a precise abstraction layer of the function for use in fuzzing [57].

For these reasons, research is done to automate the process. LLMs are also considered since, with their general-purpose implementation, they can adapt better to existing setups. This section focuses on using LLMs in fuzz testing, and it discusses the potential of LLMs in this field.

5.1 Input generation with LLM fuzzers

A fuzzer can be viewed as a generator which creates random inputs [56]. Since the underlying implementation of the generator can vary a lot, different types of fuzzers exist in practice. In the work of Beaman et al. [7], a definition for the various types was created, depending on how advanced the fuzzer is. A fuzzer is mainly categorised by the knowledge it has about the function and the system under test, and also by how it generates the input and how it reaches the testing coverage. In Table 3, we give an overview of the different classification types and a short description based on the definitions from Beaman et al. [7]. Additionally, the table contains a categorisation of the fuzzing-related research we cover in this work.

Type	Definition	Research
Input knowledge		
Dumb	It follows strictly its seed generation process.	-
Smart	Alters the seed generation process to better suit the function under test.	[8, 46, 52, 59]
System knowledge		
Black-box	It has no information about the underlying system.	-
White-box	It has all the information about the underlying system.	-
Grey-box	It is a mix between black-box and white-box, where it partially has information about the underlying system.	[8, 46, 52, 59]
Generation method		
Random	It creates seeds randomly.	-
Genration-based	It generates seeds based on certain parameters and routines.	[8, 46, 52]
Mutation-based	It mutates already generated seeds by adding, removing or changing parts from the seed.	[52, 59]
Testing coverage		
Directed	It tests a specific part of the code or function.	[8, 46, 52, 59]
Coverage-based	Code coverage describes the parts of the code which were executed by the calling function. The aim is to achieve the highest possible code coverage.	[52]

Table 3: Overview of the different classification types of a fuzzer (based on [7]).

5.1.1 Seed generation with LLM

Seed generation is fundamental for fuzzers, since it represents the basis of the inputs. It is challenging because it requires knowledge of the underlying functions, the technologies used, and the specifications of the software. In addition, the use of existing solutions may be impractical if the used tech stack is not compatible [8,34]. Because of the discussed obstacles, automated tools are preferred. Such automated tools are covered in the work of Tamminga [46] and Black et al. [8].

Tamminga [46] investigated if an LLM can be modified to use it as a seed generator for the existing fuzzer libFuzzer[2] and on the programming language Go, but with the aim to be independent of the used tech stack. As a basis, pre-trained LLMs were used and compared against each other. Furthermore, the LLMs were optimised for seed generation by either using prompting or by fine-tuning using a self-created dataset [46].

Black et al. [8] focused their seed generator for the Arthesis[3] fuzzer, which is a fuzzer for the programming language Python. As for the prompt, a task description, the function under test, and a description of the expected output were included. To test the effectiveness of seed generation with LLM, they created a testing pipeline that allows the generated seeds to be passed to the function under test [8].

To measure the performance, Tamminga [46] created an evaluation method based on the core idea of the benchmarking process Magma, which was developed by Hazimeh et al. [26]. The benchmark includes measurements about the count of detected bugs and the time within which they were discovered [46]. Ultimately, StarCoderPlus with prompt engineering could detect 39% of the crashes within 30 seconds, while 64% were triggered within 10 minutes. In comparison, libFuzzer without any seed generation only reached 23% in

[2] https://llvm.org/docs/LibFuzzer.html
[3] https://github.com/google/atheris

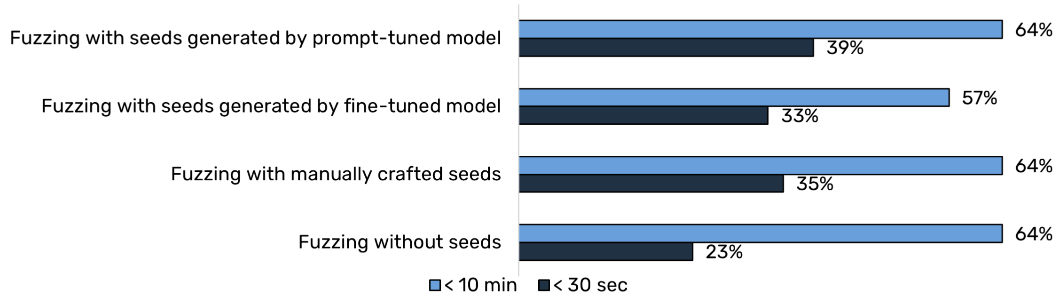

Figure 5: Comparison of libFuzzer with different seed generation approaches (adapted from [46]).

30 seconds, but also 64% in 10 minutes. A performance summary, based on the measurement from Tamminga [46] is in Figure 5 visualised.

Black et al. [8] used the reached coverage as a performance measurement. As for the tests, they had three different approaches. The first approach uses only the fuzzer. In the second approach, a combination of fuzzer and LLM is used, and in the last approach, only the LLM is used. While there was no clear winner, the combination of fuzzer and LLM performed the best in most of the test cases, while for the other test cases, fuzzing alone or LLM alone were better [8].

5.1.2 Mutation-based fuzzer with LLM

Fuzz4All, developed by Xia et al. [52], goes one step further and takes mutation generation into account. It implements a fuzzer, which is independent of the used tech stack and can be used for a wide range of programming languages. In their work, they present how Fuzz4All is designed and what steps it runs through. To get fuzzing inputs, the user has to provide information for Fuzz4All. This can include documentation, manuals or the application code. The input is then applied to the autoprompting step. Since this information is written in natural language, while the expected output should be code, Fuzz4All uses two LLM models; one is used for the distillation phase, while the other is used for the generation phase.

In the distillation phase, the LLM attempts to understand and bundle the provided user information and to represent it as candidate prompts containing only the relevant information. Those candidate prompts are then passed to the generation phase, in which another LLM tries to generate fuzzing inputs. Afterwards, the candidates are evaluated against the function under test by executing a fuzzing test. The evaluation calculates a score based on criteria like code coverage, triggered crashes or, like in the example of Xia et al. [52], the validity of the input. The candidate prompt with the highest score is then used as the initial prompt.

The next step in Fuzz4All is the fuzzing loop. It is an iterative process for generating fuzzing inputs. Some substeps are thereby similar to those of the autoprompting step. Firstly, the initial input prompt is applied to the generation LLM, which generates fuzzing inputs. Secondly, the fuzzing input is applied to the function under test to check the validity of the fuzzing input and, ultimately, to trigger bugs. With this process, code snippets are gained, which can be applied as fuzzing inputs. Since the aim is to gain as many different inputs as possible, a mutation step is applied. It mutates the code snippet based on a randomly chosen strategy to enable the LLM to generate different fuzzing inputs. There are three mutation strategies. Either the code snippets are mutated, semantically changed or completely newly generated. Lastly, the output is used again as input for the LLM generation. This process is applied iteratively until a stop criterion is met [52].

This structure has different advantages. Firstly, since two LLMs are in use, one can choose them according to their capabilities. For the distillation phase, an LLM with a good understanding of natural language should be chosen. The second LLM used in the generation phase should be trained for code generation. Secondly, time efficiency can be achieved since smaller LLM models can be used because the LLMs need only to focus on specific tasks.

Compared to traditional fuzzers, this approach generates fewer valid inputs, but it achieves a higher coverage in less time. Figure 6 shows the comparison of Fuzz4All with traditional tools based on the measurement from Xia et al. [52].

5.1.3 Greybox fuzzing with LLM

Zhang et al. [59] implemented a tool called LLAMAFUZZ, which is used for improving greybox fuzzing. Their approach consists of connecting the ALF++[4] fuzzer with Llama 2. While the LLM is used for input mutation, the fuzzer has the task to execute the generated input in the function under test, to monitor the execution and to pass the result back to the LLM [59].

LLAMAFUZZ could outperform traditional tools by reaching a higher code coverage and achieving a higher bug count.

[4] https://aflplus.plus/

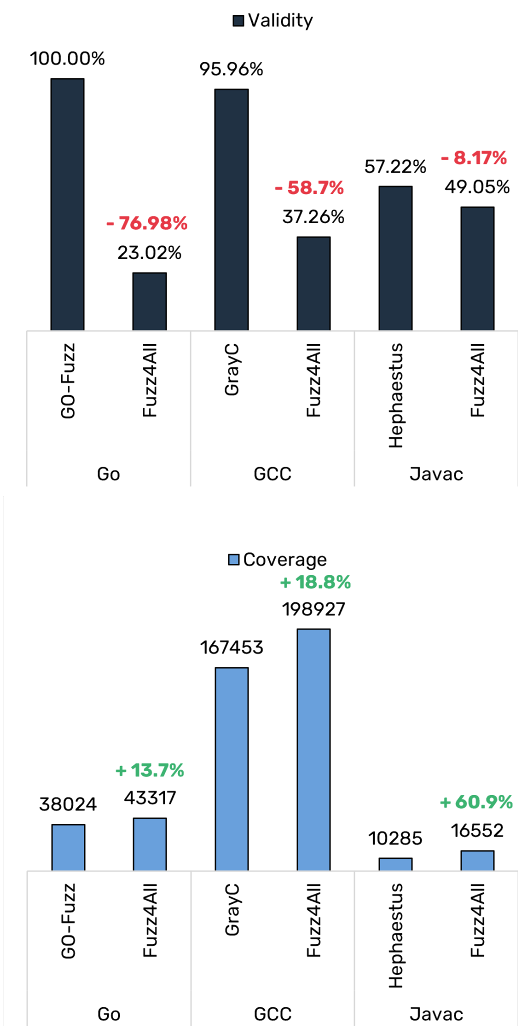

Figure 6: Validity and coverage comparison between Fuzz4All and traditional tools (adapted from [52]).

Overall, it increased the code coverage by 27.2% in comparison to AFL++ without LLM enhancement and detected 41 bugs more on average [59].

5.2 Fuzz driver generation

A fuzz driver describes the link between the system under test and the fuzzer by providing an API. This is a laborious task since they are typically written manually [57]. Additionally, the API needs to be as precise as possible since mistakes in the implementation could increase the falsification of the result for the system under test [58].

Zhang et al. [58] considered different approaches for prompt engineering an LLM to gain a fuzz driver generator. The best-performing strategy they found is iterative prompt processing. The strategy consists of a generation and several fix prompts. As seen in Figure 7, the prompt consists of a task description for the LLM, a code documentation and a code snippet of the API. ChatGPT gives, in this case, a proposal of how implementing the function “FuzzerTest” could look. The output can also be seen in Figure 7. Afterwards, one or more fix prompts are applied on cases where the produced LLM output has compilation errors [58].

User Prompt

```
// The following is a fuzz driver written in C language,
// complete the implementation. Output the continued code
// in reply only.
#include "example/array.h"

/*
* @brief: Gets the element at the given index.
* @param: index of the element
*/
extern int getArrayNumber(size_t index)

// the following function fuzzes getArrayNumber
int FuzzerTest(const uint8_t *Data, size_t Size)
```

ChatGPT Answer

```
{
  if (Size < sizeof(size_t)) {
    return 0; // Not enough data to extract an index
  }

  // Extract index from the input data
  size_t index;
  memcpy(&index, Data, sizeof(size_t));

  // Call the function being fuzzed
  int result = getArrayNumber(index);


  // Perform any sanity checks or validations on the result
  // For now, we'll simply return success
  return 0;
}
```

Figure 7: Generation prompt and output from ChatGPT (adapted from [38, 58]).

During the investigation of fuzz driver generation with LLMs, Zhang et al. [58] discovered that the performance depends on the complexity of the API. An LLM needs to be able to forecast how an API works based on the given questions and inputs from the user. Therefore, a conclusion can be drawn that when the complexity of the system under test rises, the performance of the LLM decreases. The best performance was reached by GPT-4, which could answer 78 out of 86 questions in the right configuration [58].

LLM-aided fuzzing is already used in practice. In 2023, Google [36] added LLM to their OSS-Fuzz project. It is used to automate vulnerability detection, mainly in open-source software. While doing so, they found that, in some cases, the code coverage was improved. In one case, they increased the code coverage by over 30%. To reach the same level for all OSS-Fuzz projects, they stated that "several years" of manual adaptation would have been necessary [36].

6 Exploit generation for software testing with LLM

As discussed in Section 4, LLM suffers from a high FPR. But traditional approaches also have false positives [55]. This causes software developers to question the results of automated vulnerability detectors. Therefore, showing if and how a vulnerability is exploitable is essential [60]. One example is a dependency vulnerability detector. When it reports a vulnerability in one of the used libraries, it does not necessarily mean it is exploitable in the application. A way of showing the exploitability of a vulnerability is to write an exploit [60].

6.1 Research approaches

In the research of Zhang et al. [60], ChatGPT-4 was used to generate security tests, which could then be used to test the exploitability of the vulnerable dependency. The following parameters were included in the prompt [60]: *Task description, function name, vulnerability ID, vulnerable API list, test function name incl. an exemplar test implementation, and client code implementation*. When the LLM output contained smaller errors, a manual process was applied to correct the output.

Fuzzing, which we discussed in Section 5, was used by Zhou et al. [61] in their tool called Magneto. It aims to exploit vulnerabilities in complex environments. They explain the mechanism of Magneto as follows: In a first step, information about the vulnerability itself is collected, including the affected version and the functions, as well as an exploit and its oracle. This information is then compared with the dependency tree of the software project, and only the dependencies matching the vulnerability description are kept. Afterwards, Magneto tries to understand the underlying architecture of the software by depicting the call chains of the software under test. Based on the gathered information, Magneto tries to exploit the vulnerability. For this, Magneto needs to find an input so that it can be passed to the function on top of the call chain while also remaining a capable input to trigger the vulnerability on the dependency under test. Its approach is done incrementally, by trying out different seeds, which are created by an LLM. It starts with the function call to the dependency and works its way up until it reaches an exposed function. With this approach, the function may be able to find a vulnerability [61].

Fang et al. [18] analysed the efficiency of LLM in exploit generation for one-day vulnerabilities. To do this, they used different LLMs and leveraged them for exploit generation. The LLMs were provided with different resources, such as a web browser, a terminal, and the ability to use a code interpreter and to create or modify files [18].

As already discussed in Section 4, Fu et al. [20] had a look at a variety of tasks in the software vulnerability spectrum. Here, we want to take a glance at the automated severity estimation and mitigation of vulnerabilities with LLM. For the severity estimation prompt, Fu et al. [20] proposed to include a description of what the LLM has to do and to include the vulnerable function. For the mitigation prompt, they included several generic examples of vulnerable functions and their repair methods with a task description, which is similar to a few-shot prompting approach [20].

6.2 Performance comparison

Zhang et al. [60] tested the GPT model on 55 apps containing vulnerabilities. As a result, in 24 cases, the vulnerability could be successfully exploited. In addition, a comparison was made with the traditional tools SIEGE [27] and TRANSFER [32]. While TRANSFER [32] achieved writing four exploits, SIEGE [27] could not generate one. This leads ChatGPT to surpass them [60]. Magneto, created by Zhou et al. [61], was at least 75.6% more successful in creating an exploit compared to traditional tools like SIEGE [27], TRANSFER [32] and VESTA [9].

Fang et al. [18] tested their prompt-engineered GPT-4 model against the traditional tools ZAP [5] and Metasploit [6]. In the end, they found out that the traditional tools, as well as the other considered models, were not able to generate exploits. Only their GPT-4 model could create exploits, but with a success rate of 87%. To perform that well, the inclusion of the CVE description in the prompt was mandatory [18].

For the severity estimation and mitigation task, Fu et al. [20] came to a sobering result since the model was not able to sufficiently estimate the severity or propose mitigations.

7 LLM challenges and future outlook

In this section, we delineate our key takeaways through the lens of the gathered information from the listed papers in

[5]https://www.zaproxy.org/

[6]https://www.metasploit.com/

Table 1 and discuss the challenges and possible future approaches. The identification was made by mapping the mentioned challenges from the discussed papers while also considering the general difficulties of LLMs.

7.1 High False-Positive Rate (FPR)

A problem encountered in the discussed topics was the high FPR and the low accuracy, which was discovered by various researchers [10, 20, 29, 41, 45].

The recommendation from Cheshkov et al. [10] is to invest further research in prompt engineering. Having a more enhanced prompt, like the proposed chain-of-thought technique, could lead to better performance in the LLM for code vulnerability detection [10]. To correct the errors produced by the LLM, Jiang et al. [29] suggest using the LLM again for output correction. This enables the LLM to improve its output by gaining insights into bugs introduced in its previous output.

An explanation can also be found in hallucination. Hallucination describes the effect of an LLM that writes factually incorrect outputs [31]. According to Kamath et al. [31], a reason for this behaviour could be a lack of knowledge because of missing, biased, or untrue training data in the pre-training process. To mitigate some of the named reasons, a common technique is retrieval augmented generation (RAG), which aims to include external knowledge sources [4]. We describe this technique in more detail in Section 7.2.

Decoding strategies can also be a culprit for hallucination. They are often used to introduce randomness in favour of producing more natural-sounding language. Therefore, new decoding strategies are designed to diminish hallucinations [30]. Two examples are uncertainty-aware beam search [53] and confident decoding [47].

7.2 Outdated data

Another challenge encountered in training is keeping the information up to date. Some research from the field of code vulnerability detection [10, 20, 54, 55] and exploit generation [18, 61], handled with CWE and CVE definitions. While they used mostly already known CWEs and CVEs covering general vulnerability topics, it is important that the information basis of an LLM is up-to-date to also understand new vulnerability definitions.

There are different new approaches used in updating the information of a model. The simplest is to enhance the prompt by including missing information directly into the prompt. For example, this is shown by Fang et al. [18], where they had to include the CVE description to gain a well-performing model.

A new approach to facing this challenge is RAG. It enables an LLM to extend its knowledge base with additional resources like websites, databases or other forms of data storage. Those reference points are then considered when a corresponding prompt is placed, asking for specific information [4].

Model editing can also be used for this challenge. It tries to identify the incorrect information base of an LLM and to modify it correspondingly [30].

7.3 LLM training

Pre-training an LLM is cost- and time-intensive [30]. This limits the capability of researchers to create LLMs specifically designed for the purposes we discussed in this paper.

To reduce computational power requirements, efforts are made to understand so-called scaling laws. Concretely, in the context of LLM, the interplay between the model size, data size and computational power is analysed. An alternation in one of those three resources could lead to a corresponding change in a different resource [50].

A common technique used today to bypass pre-training is to take a pre-trained model and fine-tune it. As found out by Guo et al. [24], this allows for better performance in some tasks compared to larger LLMs. However, it comes with its challenges, like finding high-quality datasets. This challenge was mentioned by Guo et al. [24], in which they made the finding that there was some incorrect labelling in the datasets, leading to wrong training of the LLM.

Another technique is to apply different prompt strategies as discussed in various works throughout this paper. The benefit comes from the relatively easy application since no modification to the model itself has to be made. However, while there exist some guidelines and strategies, like those from Sahoo et al. [42], it is difficult to determine which prompt leads to better results [30], making it a trial-and-error process.

Also, the requirements for the setup have to be considered, since they are similar to the ones for pre-training. To fine-tune a model, it must be downloaded, installed and executed [30]. Parameter-efficient fine-tuning (PEFT) is a recent technique to train the LLM for a specific task. The fundamental concept is to add a final layer with trainable parameters. During training, only the parameters of the additional layer are modified while the other layers remain unchanged [44].

7.4 Limited context length understanding

The applications of the discussed areas in this paper require a deeper contextual understanding to perform well. As a data basis, software code is provided, which includes different information like the architecture, program logic and documentation. An LLM has the requirement to understand the system under test and to perform the described tasks on it. However, LLMs have a limitation in understanding contexts, leading to missing essential parts [30]. A further effect can be seen in fuzzing, since the limited context length understanding leads to an increase in the creation of invalid seeds [29].

There are different ways to tackle this type of challenge, which may be applied in the future. According to Kaddour et al. [30], current research focuses on improving the capability of attention mechanisms to understand a wider context. Other research is focusing on length generalisation to keep the advantage of training LLMs on short input, while also being able to understand longer inputs. Kaddour et al. [30] also discussed using alternatives to transformers. Architectures like state space models (SSMs) [19] or receptance weighted key value (RWKV) [40] are designed for understanding longer context.

8 Discussion

In this section, we summarise the key takeaways based on the analysis of the related work and make the connection to the research questions. We also provide additional insight by offering tips and recommendations.

8.1 RQ1: What are practical use cases for LLMs in security code review and testing, and how do they perform against traditional tools?

LLMs have their right to exist in the context of security code review and testing. While in some topics they reach practical relevance, others are still in the research stage. In the following subchapter, we go into detail about each topic's prospects.

8.1.1 Code vulnerability detection

LLMs in code vulnerability detection can overcome various challenges from the use of traditional tools, since their setup, application and leveraging to the code base is simpler than that of conventional tools. However, the LLMs in the research suffered from a high FPR, limiting the capability of LLMs in this use case. In comparison, traditional tools focus more on true positives and thus reduce the exertions from developers. Because of those findings, our recommendation is to put more effort into research and, if the setup allows, to rely on traditional tools for production at the moment.

8.1.2 Fuzz testing

In fuzzing, LLMs have advantages. On the one hand, it is less time-consuming and can find coverage-increasing seeds faster than its traditional counterparts. On the other hand, it can be used independently of the technology environment, while traditional tools have mostly binding requirements.

In all considered work, the LLM could outperform traditional tools in input generation. This is also true for fuzz driver generation, which is already used in productive use cases. This makes LLMs in the context of fuzz testing valuable, especially for testing a wide range of projects. Our recommendation is therefore to convert research findings into practice and develop productive and marketable tools that enable the easy use of LLMs in fuzz testing. However, some challenges, especially in the limited understanding of complex implementations, remain.

8.1.3 Exploit generation

The included papers show that exploit generation has potential, especially since they outperform conventional tools. However, prerequisites have to be met, like providing specific prompt information or manual adjustments, to function properly. Additionally, only certain LLMs achieve acceptable results, making it dependent on the chosen LLM.

This leads to our conclusion that, at the moment, LLMs can only be used to a limited extent for exploit generation. However, since traditional tools also do not perform well, manual effort is still needed in practice. Nevertheless, further research investigations should be made.

8.2 RQ2: What is the impact of the LLM on the performance?

The most important component in the considered tools is the LLM itself. There exist a lot of different LLMs with varying parameter count, training basis, etc. In the related work, various models were used, including open-source and closed-source. While the performance between those works is not directly comparable, some general conclusions can be drawn:

1. Prompt engineering and fine-tuning can improve the performance in most cases compared to their base counterparts.
2. Newer LLMs are usually better than their previous versions.
3. While improvements can be achieved with techniques already mentioned, an indicator for better performance can also be the parameter count, while LLMs with more parameters should perform better.

This also reflects in the works, where using newer models leads to better results or even makes an approach possible. With the introduction of new models (i.e. GPT-5), the tests should be repeated so that the performance improvement can be measured. The parameter count of an LLM should also not be neglected. However, some works show that by applying prompt engineering or fine-tuning, the performance can be improved. Thus, this should be taken into consideration, especially if computational power is limited.

Choosing an LLM is a difficult undertaking and not always obvious. Indicators for good LLMs are their introduction dates, the newer the better, as well as their parameter count and general benchmarking results. Additionally, one can take a look at whether a security-trained version exists. Nevertheless, comparing the different LLMs in a benchmarking process is essential to identify the best-performing one.

Also, an interesting concept is to use a combination of LLMs. This allows for focusing LLMs on one specific task, making their tuning simpler.

8.3 RQ3: What do current approaches look like?

Different techniques can be used to tweak LLMs to be more tailored to specific areas of application. Since we included a range of papers, different approaches were used for the same aim.

8.3.1 Prompt engineering

Prompt engineering is a common technique which involves designing and structuring a prompt. The aim is to instruct and teach the LLM on how to complete a task in a way that is understandable from the perspective of an LLM.

In the considered works, various prompts were tested, each containing different levels of information. One common finding was that prompts including more information were more successful. However, the information has to be articulated and designed such that known challenges like limited context length understanding are not triggered.

Since prompt engineering is not an exact science and varies from use case to use case, we propose to try different combinations of information sources and to benchmark them accordingly. Also, having a look at recent findings of prompt engineering techniques is worthwhile. However, again, there is no predetermined solution.

8.3.2 Fine-tuning

Fine-tuning is more efficient than the training of an LLM and allows for supplementing the LLM with additional training datasets. Though in practice, finding qualitative and good training datasets is difficult. Having incomplete or false datasets could even worsen the performance of an LLM. Thus, the key takeaway is to apply fine-tuning, but the creation of clean datasets should be a focus point too.

8.3.3 Tool creation

In the different considered topics, but especially for fuzz testing, the researchers created tools in which an LLM is only one part of the system. The advantage of the combination of LLMs and traditional approaches is the possibility of having deterministic behaviour for precise tasks. For example, for the testing of seeds, a unit test is created that directly applies the generated seed to the function, reducing the false positive count. Whenever possible, deterministic tasks should be outsourced to dedicated processes, so that for those tasks, hallucination can be reduced.

8.4 RQ4: What are the current challenges and what are the prospects for LLMs in security code review and testing

In general, the creation and training of LLMs is a common challenge. Because of the high cost and time consumption involved when creating an LLM, we recommend using existing general-purpose LLMs and leveraging them by using techniques like prompt engineering and fine-tuning. Since fine-tuning itself is also resource-intensive, new methods like PEFT should be explored, especially when resources are missing for full tuning. Also, finding the right datasets for training is difficult. The dataset has to be cleaned, and wrong data has to be removed.

Furthermore, pitfalls can be found in general LLM challenges like high FPR, outdated data and limited context length understanding. While all topics suffer from those challenges, they are not affected equally.

8.4.1 Code vulnerability detection

For the code vulnerability detection task, the high FPR is the main challenge. A high FPR in this topic makes an LLM unusable, since software developers would need to check more possible threats. Thus, the aim for future research should be the reduction of the FPR, even if this would mean an increase in FNR. This would allow a developer to focus on true positives.

For this, already mentioned techniques like prompt engineering and fine-tuning should be used. Also, the creation of clean datasets should be a focus point, since those are currently missing. Moreover, a combination with the topic exploit generation is conceivable, since this would allow for implementing a check.

8.4.2 Fuzz testing

In Fuzzing, FPRs can be mitigated with enhanced controls, such as applying software tests. However, here, the limited context length makes it difficult for the LLM to understand complex and extensive software.

Thus, experimenting with the right prompt configuration is essential. Also, providing the right amount of information is necessary.

8.4.3 Exploit generation

Exploit generation lives from data actuality, since it needs to know the information about new vulnerabilities. However, since training is cost-intensive, the knowledge base of LLMs suffers from outdated data.

To overcome this challenge, our recommendation is to use RAG and to connect the LLM to corresponding vulnerability databases.

9 Conclusion

Practical implementation areas for an LLM in software security can be found in code vulnerability detection, fuzz testing and exploit generation. The LLMs can be applied to those tasks by using prompt engineering, fine-tuning and creating dedicated tools.

However, the performance varies between those different disciplines. While LLMs achieve excellent results in fuzz testing, they need further research in code vulnerability detection and exploit generation. This is mainly attributable to challenges like hallucination, high training costs, data actuality and the limited context length understanding. Those challenges lead to a high FPR in code vulnerability detection, poor performance in complex fuzz testing cases and insufficient knowledge for exploit generation.

To overcome those challenges, we propose in future research to explore prompt engineering and fine-tuning further. What is more, modern technologies like RAG can help in improving data actuality, while PEFT can help reduce training costs.

References

[1] ACM. ACM Digital Library. Accessed: 04.08.2025. URL: https://dl.acm.org/.

[2] Altexsoft. Language models, explained: How GPT and other models work, January 2023. Accessed: 2024-10-31. URL: https://www.altexsoft.com/blog/language-models-gpt/.

[3] Thimira Amaratunga. *Understanding Large Language Models*. Apress Berkeley, CA, 2023. URL: https://doi.org/10.1007/979-8-8688-0017-7.

[4] Amazon. Was ist Retrieval-Augmented Generation (RAG), 2024. Accessed: 2024-11-29. URL: https://aws.amazon.com/de/what-is/retrieval-augmented-generation/.

[5] arXiv. arXiv. Accessed: 04.08.2025. URL: https://arxiv.org/.

[6] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721, 2013. https://doi.org/10.1109/ICSE.2013.6606617.

[7] Craig Beaman, Michael Redbourne, J. Darren Mummery, and Saqib Hakak. Fuzzing vulnerability discovery techniques: Survey, challenges and future directions. *Computers & Security*, 120:102813, September 2022. URL: https://www.sciencedirect.com/science/article/pii/S0167404822002073, https://doi.org/https://doi.org/10.1016/j.cose.2022.102813.

[8] Gavin Black, Varghese Mathew Vaidyan, and Gurcan Comert. Evaluating large language models for enhanced fuzzing: An analysis framework for LLM-driven seed generation. *IEEE Access*, 12:156065–156081, 2024. https://doi.org/10.1109/ACCESS.2024.3484947.

[9] Zirui Chen, Xing Hu, Xin Xia, Yi Gao, Tongtong Xu, David Lo, and Xiaohu Yang. Exploiting library vulnerability via migration based automating test generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. URL: https://doi.org/10.1145/3597503.3639583.

[10] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. Evaluation of ChatGPT model for vulnerability detection. *arXiv e-prints*, page arXiv:2304.07232, April 2023. arXiv:2304.07232, https://doi.org/10.48550/arXiv.2304.07232.

[11] Edward Chopskie. SAST vs. DAST: 5 key differences and why to use them together, March 2025. Accessed: 18.07.2025. URL: https://brightsec.com/blog/sast-vs-dast/.

[12] Codegrip. Code review trends in 2022. 2022. Accessed: 2024-10-14. URL: https://media.trustradius.com/product-downloadables/DD/D7/XID8MVZTH0JF.pdf.

[13] CodeSigning. What is secure DevOps? SecDevOps explained. Accessed: 2025-04-05. URL: https://codesigningstore.com/what-is-secure-devops.

[14] Malik Imran Daud. Secure software development model: A guide for secure software life cycle. In *Proceedings of the international MultiConference of Engineers and Computer Scientists*, volume 1, pages 17–19, 2010. URL: https://www.iaeng.org/publication/IMECS2010/IMECS2010_pp724-728.pdf.

[15] DeepSeek. Introducing deepseek-v3, December 2024. Accessed: 11.04.2025. URL: https://api-docs.deepseek.com/news/news1226.

[16] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner.

An empirical study on the effectiveness of security code review. In Jan Jürjens, Benjamin Livshits, and Riccardo Scandariato, editors, *Engineering Secure Software and Systems*, pages 197–212, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. URL: https://doi.org/10.1007/978-3-642-36563-8_14.

[17] Elsevier. ScienceDirect. Accessed: 13.09.2025. URL: https://www.sciencedirect.com/.

[18] Richard Fang, Rohan Bindu, Akul Gupta, and Daniel Kang. LLM agents can autonomously exploit one-day vulnerabilities, 2024. URL: https://arxiv.org/abs/2404.08144, arXiv:2404.08144.

[19] Daniel Y. Fu, Tri Dao, Khaled K. Saab, Armin W. Thomas, Atri Rudra, and Christopher Ré. Hungry Hungry Hippos: Towards language modeling with State Space Models, 2023. URL: https://arxiv.org/abs/2212.14052, arXiv:2212.14052.

[20] Michael Fu, Chakkrit Tantithamthavorn, Van Nguyen, and Trung Le. ChatGPT for vulnerability detection, classification, and repair: How far are we?, 2023. URL: https://arxiv.org/abs/2310.09810, arXiv:2310.09810.

[21] Hadi Ghanbari, Tero Vartiainen, and Mikko Siponen. Omission of quality software development practices: A systematic literature review. *ACM Comput. Surv.*, 51(2), February 2018. https://doi.org/10.1145/3177746.

[22] Google. Google Scholar. Accessed: 04.08.2025. URL: https://scholar.google.com/.

[23] Google. Introduction to large language models, September 2024. Accessed: 2024-10-29. URL: https://developers.google.com/machine-learning/resources/intro-llms.

[24] Yuejun Guo, Constantinos Patsakis, Qiang Hu, Qiang Tang, and Fran Casino. Outside the comfort zone: Analysing LLM capabilities in software vulnerability detection, 2024. URL: https://arxiv.org/abs/2408.16400, arXiv:2408.16400.

[25] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. A survey on large language models: Applications, challenges, limitations, and practical usage. *Authorea Preprints*, 2023. https://doi.org/10.36227/techrxiv.23589741.v1.

[26] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), November 2020. https://doi.org/10.1145/3428334.

[27] Emanuele Iannone, Dario Di Nucci, Antonino Sabetta, and Andrea De Lucia. Toward automated exploit generation for known vulnerabilities in open-source libraries. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 396–400, 2021. https://doi.org/10.1109/ICPC52881.2021.00046.

[28] IEEE. IEEE Xplore. Accessed: 04.08.2025. URL: https://ieeexplore.ieee.org/Xplore/home.jsp.

[29] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, and Quan Zhang. When fuzzing meets LLMs: Challenges and opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 492–496, New York, NY, USA, 2024. Association for Computing Machinery. https://doi.org/10.1145/3663529.3663784.

[30] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. Challenges and Applications of Large Language Models. *arXiv e-prints*, page arXiv:2307.10169, July 2023. arXiv:2307.10169, https://doi.org/10.48550/arXiv.2307.10169.

[31] Uday Kamath, Kevin Keenan, Garrett Somers, and Sarah Sorenson. *LLM Challenges and Solutions*, pages 219–274. Springer Nature Switzerland, Cham, 2024. https://doi.org/10.1007/978-3-031-65647-7_6.

[32] Hong Jin Kang, Truong Giang Nguyen, Bach Le, Corina S. Păsăreanu, and David Lo. Test mimicry to assess the exploitability of library vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 276–288, New York, NY, USA, 2022. Association for Computing Machinery. URL: https://doi.org/10.1145/3533767.3534398.

[33] Andrej Karpathy. Intro to LLMs, November 2023. Accessed: 2024-11-22. URL: https://drive.google.com/file/d/1pxx_ZI7O-Nwl7ZLNk5hI3WzAsTLwvNU7/view.

[34] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 562–566, 2018. https://doi.org/10.1109/SANER.2018.8330260.

[35] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In

Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022, page 544–555, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3533767.3534380.

[36] Dongge Liu, Jonathan Metzman, Oliver Chang, and Google Open Source Security Team. AI-powered fuzzing: Breaking the bug hunting barrier, August 2023. URL: https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html.

[37] Meta. The Llama 4 herd: The beginning of a new era of natively multimodal ai innovation, April 2025. Accessed: 11.04.2025. URL: https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

[38] OpenAI, 2024. Accessed: 2025-01-12. URL: https://chatgpt.com/.

[39] OWASP. Buffer overflow attack. Accessed: 2024-10-13. URL: https://owasp.org/www-community/attacks/Buffer_overflow_attack.

[40] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Jiaju Lin, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartlomiej Koptyra, Hayden Lau, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Bolun Wang, Johan S. Wind, Stanislaw Wozniak, Ruichong Zhang, Zhenyuan Zhang, Qihang Zhao, Peng Zhou, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. RWKV: Reinventing RNNs for the transformer era, 2023. URL: https://arxiv.org/abs/2305.13048, arXiv:2305.13048.

[41] Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, and Bill Chu. Software vulnerability detection using large language models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 112–119, 2023. https://doi.org/10.1109/ISSREW60843.2023.00058.

[42] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. *arXiv e-prints*, page arXiv:2402.07927, February 2024. arXiv:2402.07927, https://doi.org/10.48550/arXiv.2402.07927.

[43] Springer. Springer Nature Link. Accessed: 04.08.2025. URL: https://link.springer.com/.

[44] Cole Stryker and Ivan Belcic. What is parameter-efficient fine-tuning (PEFT)?, August 2024. Accessed: 2025-01-12. URL: https://www.ibm.com/think/topics/parameter-efficient-fine-tuning.

[45] Karl Tamberg and Hayretdin Bahsi. Harnessing large language models for software vulnerability detection: A comprehensive benchmarking study. *IEEE Access*, 13:29698–29717, 2025. https://doi.org/10.1109/ACCESS.2025.3541146.

[46] Elwin Tamminga. Utilizing Large Language Models for Fuzzing: A Novel Deep Learning Approach to Seed Generation. Master's thesis, Radboud University, Faculty of Science, November 2023. URL: https://www.cs.ru.nl/masters-theses/2023/E_Tamminga___Utilizing_large_language_models_for_fuzzing.pdf.

[47] Ran Tian, Shashi Narayan, Thibault Sellam, and Ankur P. Parikh. Sticking to the facts: Confident decoding for faithful data-to-text generation, 2020. URL: https://arxiv.org/abs/1910.08684, arXiv:1910.08684.

[48] Bert van Wee and David Banister. Literature review papers: the search and selection process. *Journal of Decision Systems*, 33(4):559–565, April 2023. URL: "https://doi.org/10.1080/12460125.2023.2197703", arXiv:https://doi.org/10.1080/12460125.2023.2197703, https://doi.org/10.1080/12460125.2023.2197703.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL: https://arxiv.org/abs/1706.03762, arXiv:1706.03762.

[50] Dagang Wei. Demystifying scaling laws in large language models, May 2024. Accessed: 2025-01-12. URL: https://medium.com/@weidagang/demystifying-scaling-laws-in-large-language-models-14caf8ac6f80.

[51] Wiley. Wiley Online Library. Accessed: 13.09.2025. URL: https://onlinelibrary.wiley.com/.

[52] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4All: Universal fuzzing with large language models, 2024. Accessed: 2024-11-01. URL: https://arxiv.org/abs/2308.04748, arXiv:2308.04748.

[53] Yijun Xiao and William Yang Wang. On hallucination and predictive uncertainty in conditional language generation, 2021. URL: https://arxiv.org/abs/2103.15025, arXiv:2103.15025.

[54] Xin Yin, Chao Ni, and Shaohua Wang. Multitask-based evaluation of open-source LLM on software vulnerability. *IEEE Transactions on Software Engineering*, 50(11):3071–3087, 2024. https://doi.org/10.1109/TSE.2024.3470333.

[55] Jiaxin Yu, Peng Liang, Yujia Fu, Amjed Tahir, Mojtaba Shahin, Chong Wang, and Yangxiao Cai. An insight into security code review with LLMs: Capabilities, obstacles and influential factors, 2024. URL: https://arxiv.org/abs/2401.16310, arXiv:2401.16310.

[56] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. Accessed: 2025-01-16. URL: https://www.fuzzingbook.org/.

[57] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. APICraft: Fuzz driver generation for closed-source SDK libraries. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2811–2828. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-cen.

[58] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. How effective are they? exploring large language model based fuzz driver generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '24, page 1223–1235. ACM, September 2024. https://doi.org/10.1145/3650212.3680355.

[59] Hongxiang Zhang, Yuyang Rong, Yifeng He, and Hao Chen. Llamafuzz: Large language model enhanced greybox fuzzing, 2024. URL: https://arxiv.org/abs/2406.07714, arXiv:2406.07714.

[60] Ying Zhang, Wenjia Song, Zhengjie Ji, Danfeng, Yao, and Na Meng. How well does LLM generate security tests? *arXiv e-prints*, page arXiv:2310.00710, October 2023. arXiv:2310.00710, https://doi.org/10.48550/arXiv.2310.00710.

[61] Zhuotong Zhou, Yongzhuo Yang, Susheng Wu, Yiheng Huang, Bihuan Chen, and Xin Peng. Magneto: A step-wise approach to exploit vulnerabilities in dependent libraries via LLM-empowered directed fuzzing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1633–1644, New York, NY, USA, 2024. Association for Computing Machinery. URL: https://doi.org/10.1145/3691620.3695531.