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 life. 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. In that regard, many security incidents and breaches have been occurring, leading to damage caused by vulnerabilities in software. One example is the Java framework Apache Log4j, which was discovered to have a vulnerability, allowing an attacker to execute a remote code execution attack [21]. Some other examples are related to insecure software development practices, like the discovered vulnerabilities in Zoom [1] or malicious actors trying to introduce backdoors in popular dependencies, with one recent incident being the XZ backdoor [39].

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 [10]. Similarly, the Secure Software Development Lifecycle (SSDLC) also targets security but with a focus on software development [11]. Eventually, all these methodologies and frameworks entail some common traits: Efficient and diligent vulnerability detection and testing are crucial.

New methods and tools are researched and implemented to help developers achieve secure software. One new approach is using Large Language Models (LLMs). The focus of interest for LLMs comes from its architecture, which was first introduced in 2017 by Vaswani et al. [40]. 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 multifarious ways [3]. For example, LLMs are already used in various ways, ranging from medicine to education, and also in software engineering [19], while software security represents a new research field.

In this paper, we provide a systemised 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 exploitables 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 do current approaches look like?

3. What are the current challenges and what are the prospects for LLMs in security code review and testing?

Table 1: Overview of the related work used in this paper

Topic	Year	Work	Key Points
C	2023	Cheshkov et al. [8]	Cheshkov et al. created a performance comparison using GPT models in code vulnerability detection by categorising the vulnerability with a binary and multi-label classification approach. *LLMs: GPT-3.5 Turbo, Davinci*
C, E	2023	Fu et al. [15]	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. *LLMs: GPT-4, GPT-3.5 Turbo, CodeBERT, GraphCodeBERT, AIBugHunter*
C	2024	Guo et al. [18]	Different LLM models with different 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. *LLMs: GPT-4, CodeBERT, VulBert, CodeLlama, Mistral, Mixtral*
C	2023	Purba et al. [34]	They compared different LLMs and traditional tools by using two different datasets to see whether the vulnerability is detected or not. The vulnerabilities were categorised by a binary classification approach. *LLMs: GPT-3.5 Turbo, Davinci, Codegen*
C	2025	Tamberg and Bahsi [37]	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. *LLMs: GPT-4 Turbo, GPT-4, Claude 3 Opus*
C	2024	Yin et al. [43]	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. *LLMs: CodeBERT, CodeLlama, DeepSeek-Coder, StarCoder, CodeT5+, etc.*
C	2024	Yu et al. [44]	Yu et al. applied five different prompts and evaluated which of them led to the best performance compared to traditional tools. *LLMs: GPT-4 Turbo, GPT-4, GPT-3.5, Gemini Pro, Llama 2*
F	2024	Black et al. [6]	They analysed the effectiveness of LLM in seed generation in combination with the existing fuzzer Atheris, especially for the programming language Python. *LLMs: GPT-4, GPT-3.5, Gemini-1.0, Claude*
F	2023	Tamminga [38]	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 laid on interoperability between different programming languages. *LLMs: GPT4, GPT-3.5 Turbo, CodeGen, StarCoderPlus, CodeT5+, etc.*
F	2024	Xia et al. [42]	Xia et al. show a practical implementation for a mutation-based fuzzer, which is called Fuzz4All. *LLMs: GPT-4, StarCoder*
F	2024	Zhang et al. [48]	They created a tool, called LLAMAFUZZ, which can be used to enhance greybox fuzzing. *LLMs: LLama 2*
F	2024	Zhang et al. [47]	Zhang et al. are showing how an LLM can be used in fuzz driver creation. *LLMs: GPT-4, GPT-3.5 Turbo, CodeLlama, Wizardcoder, text-bison*
E	2024	Fang et al. [14]	The focus is on exploit generation for one-day vulnerabilities, using LLMs. *LLMs: GPT-4, GPT-3.5, Llama 2, Mistral, Mixtral, etc.*
E	2023	Zhang et al. [49]	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. *LLMs: GPT-4*
E	2024	Zhou et al. [50]	Zhou et al. present a tool called Magneto, which uses fuzzing techniques to exploit unpatched vulnerabilities from third-party dependencies. *LLMs: GPT-4 Turbo, GPT-3 Turbo*
O	2024	Jiang et al. [23]	Here, the challenges as well as recommendations are considered. They focused on fuzzing by conducting different research in this field. *LLMs: -*
O	2023	Kaddour et al. [24]	Kaddour et al. give a general overview of current challenges when applying LLMs in practical fields. *LLMs: -*

2 Related work

To identify the related works as a manageable set, we followed the following filtering steps to the wider set of papers collected after a comprehensive search based on keywords like "LLM in Code Vulnerability Detection", "LLM Fuzzing" or "Exploit Generation with LLM in Software Development" in academic publishing venues such as ACM, IEEE, and Springer and meta-search engines like Google Scholar:

- Identify which LLM is used. Prefer more recent LLMs since they typically also demonstrate increased efficiency. Hence, we only looked at papers from 2023 or newer.
- Prefer implementations of prototypes or tools in the respective field since the application and limitations of the tools are more understandable.
- Check if performance comparisons by the creation of a test harness, including a way to collate traditional tools with LLMs, are employed.

In addition, survey papers were examined to gain an overview of current works. The listed papers were also post-filtered manually to avoid any duplication, misselection, and quality issues. At the end, 17 papers were selected.

In Table 1, an outline of those surveyed papers is given. We categorised them into four aforementioned categories, namely "code vulnerability detection (C)", "fuzz testing (F)", "exploit generation (E)" and "challenges and future outlook (O)". The final category was facilitated to provide a discussion on potential future technical research and development directions. We discuss the current approaches and summarise them to give a broad view of different approaches. We also look at the results and compare them with other papers. Please note that our work is not an exhaustive literature survey paper but a SoK paper presenting a concise and structured analysis of a focused scope (LLMs for code vulnerability detection and testing, including fuzzing and exploit generation). Furthermore, we included the LLMs that the works used while reducing the list to the more important ones and excluding version-specific details. This approach allows the reader to make a comparison over used LLMs in current research. Besides the core papers in Table 1, additional auxiliary papers were included to support the ideas or to give background information, e.g., LLM fundamentals, in our paper. Overall, we use the papers listed in Table 1 to accomplish our goal of answering the research questions in Section 1.

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. [40]. This architecture, which is based on a deep neural network, enabled the creation of large language models (LLMs) [2]. The name affix "large" comes from the count of parameters or the size of the used training dataset [17]. For example, Llama 2 has 70 billion parameters and used 10 TB of text for training, according to Karpathy [27]. 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 1, 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 [27]. 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 [31] has a total of 400 billion parameters while DeepSeek-V3 [12] has a total of 671 billion parameters. For this reason, pre-trained, foundation LLMs are used as a base and, if needed, only 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, 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 [27].

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. [35] created a survey, describing common prompt patterns. Creating prompts without further refinements is called zero-shot prompting. In few-shot prompting, we additionally include some examples, intending to give the LLM a clearer instruction. A different approach is the so-called chain-of-thought

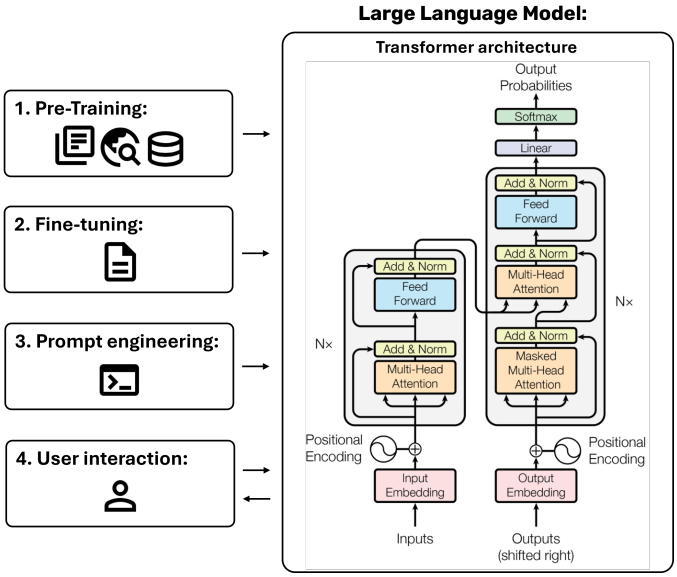

Figure 1: Architecture of LLMs and training methods (adapted from [40])

prompting. Here we guide the LLM on how to calculate the result, such that it shows its calculation steps. Auxiliary, many other abbreviations exist using similar ideas [35].

4 Automated static code vulnerability detection

Code review is a technique used in software development, where the code gets inspected by a coworker before it is merged into the productive codebase [5]. There are different reasons to conduct code reviews, some of which are shown in the study of Bacchelli and Bird [5]. They concluded 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 aims to detect and find security flaws in software [13].

Although there are benefits in conducting code reviews, they are not always carried out due to various factors. Ghanbari et al. [16] and Codegrip [9] 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 [9, 16]. An additional reason given by Ghanbari et al. [16] was the technical complexity of the project environment, leading sometimes to negligence in applying code quality improvements. Bacchelli and Bird [5] 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 the influence of the change on the functionality of the software is.

An additional aspect is the application of code reviews. There are two types: manual and automatic code reviews [13]. Although most companies favour manual code reviews [9], this is considered liable to errors, which is shown by the study of Edmundson et al. [13]. 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 [9] are regularly using an automated code review tool. The reason lies in the missing know-how for integrating such tools into their existing workflow [9].

4.1 Automated security code review with LLM

As seen in the previous part, different hindrances prevent the effective use of code reviews for code vulnerability mitigation. Hence, 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 [34].

Following, an example is shown by using ChatGPT-4 [32] and a code example from OWASP [33]. The code example contains a buffer overflow vulnerability. In such a case, the ap-

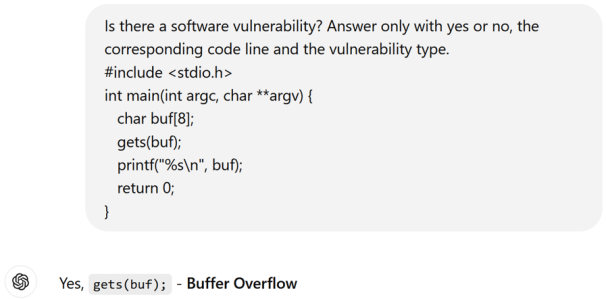

Figure 2: Interaction with ChatGPT giving the task to detect code vulnerabilities (adapted from [32–34])

plication's memory is overwritten by exceeding the assigned memory, thereby causing unpredictable behaviour in the application [33]. 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. [34]) leads to ChatGTP [32] detecting the vulnerability, pointing to the vulnerable code line and explaining why the code is viewed as unsafe. The input and output can be seen in Figure 2.

4.1.1 Adapting LLMs for code vulnerability detection

In the research of Purba et al. [34], 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 validation dataset, they used code examples containing buffer overflow and SQL injection vulnerabilities [34].

Similar to Purba et al. [34], Guo et al. [18] 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 [18].

Cheshkov et al. [8] also evaluated how well GPT models perform in vulnerability detection. Like the previous two approaches [18, 34], 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 which of those five vulnerabilities are included in the provided code snippet [8].

Another technique that can influence the results of an LLM is prompt engineering. Thus, Yu et al. [44] designed five different 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 [37] also followed the approach of testing different prompt engineering approaches by applying 23 different prompts found in related works.

Yin et al. [43] 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 descriptions already seen, the code under test, and an indicator defining one of the four mentioned tasks [43].

Fu et al. [15] 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 [15].

4.1.2 Results

In the results of Purba et al. [34], 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. [8], the binary classifier also reached a high FPR, while the multi-label classifier led to a lower F1 score and a lower precision and recall score.

Guo et al. [18] made the finding that the performance depends on the training dataset, with only a limited capability to generalise it. Fine-tuning, on the other hand, can be used to adapt smaller LLMs to certain tasks, leading to better performance than larger LLMs in those tasks. However, a problem encountered during training, which could also affect the results of other research, was the inaccuracy of the dataset [18].

As for the prompt, Yu et al. [44] observed that the prompt with an instruction and containing specific information about the CWEs performed the best. Tamberg and Bahsi [37], who also made a prompt-based approach, concluded that different models react differently to the prompt. For GPT-4 Turbo, the best performance 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 best performance was reached with

a chain of thought prompt. Here, the process on how to approach the problem step by step was described, so that the LLM can follow this manual [37].

Yin et al. [43] concluded that there is potential for LLMs in the covered tasks, but they need further improvements. In the analysis from Fu et al. [15], they concluded that base models of ChatGPT are not suitable for use in all four observed tasks because of their poor performance.

4.1.3 Comparison with traditional tools

Contrary to LLM, Purba et al. [34] and Tamberg and Bahsi [37] included an overview of the performance of traditional tools executing static code analysis. The traditional tools work by using syntactic and semantic checks [29]. For example, a rule set for a syntactic check could contain a set of different vulnerable functions, such as the mentioned `gets()` function in Figure 2 [29]. 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 [29].

In the research of Purba et al. [34], the tool Checkmarx[1] performed the best among the traditional tools with an F1 score of 47.3%. In comparison to LLMs, Checkmarx keeps a lower FPR rate at 43.1%. However, the drawback of such tools is the high false negative rate (FNR), which in the case of Checkmarx was at 41.1% [34].

Tamberg and Bahsi [37] came to a similar conclusion regarding the false positive rate. However, their model reached a higher precision with a lower recall compared to the Davinci model from Purba et al. [34]. One explanation for this outcome could be the overall performance gain with newer models since [37] was published in 2025 using GPT-4 while [34] 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.

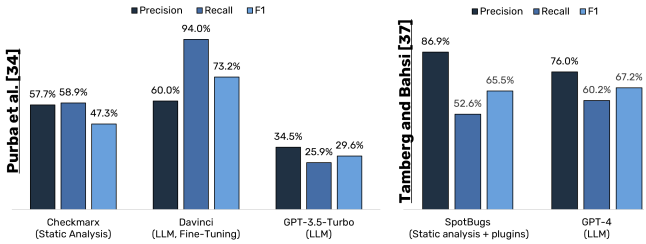

Figure 3: Performance comparison of different code vulnerability detection tools (adapted from [34, 37])

5 Fuzz Testing

Fuzz testing describes the method of using randomised input to test how a function reacts, intending to detect code flaws and potential vulnerabilities [45]. While it is considered effective in discovering software vulnerabilities, different hindrances prevent this technique's use in the industry [28].

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 [28].

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 test cases ensuring the security of the provided code [46].

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 generates random inputs. This input is later used to feed the function under test. The aim is to check whether the input is causing the function to behave unexpectedly. With an early discovery of such behaviour, bugs and security vulnerabilities can be fixed. Different degrees of fuzzers exist [45]:

- *Standard fuzzers:* A standard fuzzer creates random input without prior knowledge of how the function works. A drawback is that most of the randomly generated input will be invalid and discarded by the function under test.
- *Mutation-based fuzzers:* An enhanced method is to use a (partial) valid input, called seed, and to mutate it by adding, removing or changing input parts. This is done to achieve a more valid input, which passes initial input checks in the function under test.
- *Greybox fuzzers:* Code coverage describes the parts of the code which were executed by the calling function. A fuzzer, which has access to this information, can steer the input to trigger new processes within the function, intending to enhance the code coverage. This ensures that most of the function is executed and hence tested. Fuzzers which have underlying information of the system under test are called greybox fuzzers.

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,

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

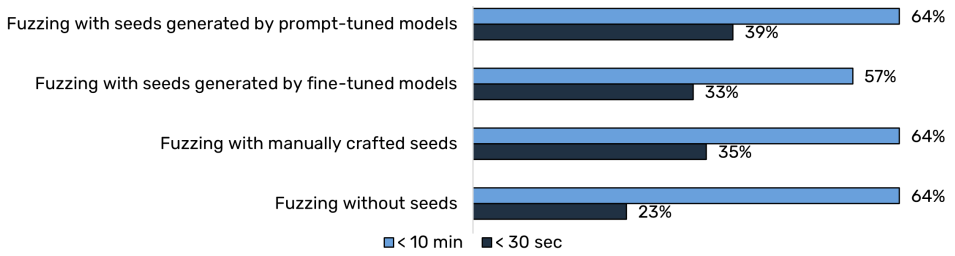

Figure 4: Comparison of libFuzzer with different seed generation approaches (adapted from [38])

and the specifications of the software [6]. In addition, the use of existing solutions may be impractical if the used tech stack is not compatible [28]. Because of the discussed obstacles, automated tools are preferred. Such automated tools are covered in the work of Tamminga [38] and Black et al. [6].

Tamminga [38] investigated if an LLM can be modified to use it as a seed generator for the existing fuzzer libFuzzer 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 fine- or prompt-tuned with a focus on seed generation. This process was done with a self-created dataset [38].

Black et al. [6] focused their seed generator for the Atheris[2] 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 [6].

To measure the performance, Tamminga [38] created an evaluation method based on the core idea of the benchmarking process Magma, which was developed by Hazimeh et al. [20]. The benchmark includes measurements about the count of detected bugs and the time within which they were discovered [20, 38]. Ultimately, StarCoderPlus with prompt engineering could detect 39% of the crashes within 30 seconds, while 64% were triggered within 10 minutes [38]. In comparison, libFuzzer without any seed generation only reached 23% in 30 seconds, but also 64% in 10 minutes [38]. A performance summary, based on the measurement from Tamminga [38] is in Figure 4 visualised.

Black et al. [6] 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 [6].

[2]https://github.com/google/atheris

5.1.2 Mutation-based fuzzer with LLM

Fuzz4All, developed by Xia et al. [42], 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. 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. [42], 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. Now, we have code snippets that are applicable as fuzzing inputs. Since we want 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 [42].

This structure has different advantages. Firstly, since two LLMs are in use, we 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 [42].

Compared to traditional fuzzers, this approach generates fewer valid inputs, but it achieves a higher coverage in less time. Figure 5 shows the comparison of Fuzz4All with traditional tools based on the measurement from Xia et al. [42].

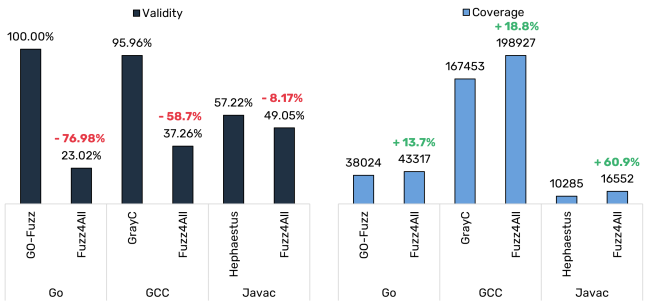

Figure 5: Validity and coverage comparison between Fuzz4All and traditional tools (adapted from [42])

5.1.3 Greybox fuzzing with LLM

Zhang et al. [48] implemented a tool called LLAMAFUZZ, which is used to enhance greybox fuzzing. As the name indicates, the used LLM is Llama 2. In greybox fuzzing, the tool gets some more information about the system. This can, for example, be the coverage reached while fuzzing. With this information, a fuzzer can then improve the generated input by trying to get a higher coverage [48].

The LLM is connected with the fuzzer, which in this case is AFL++ [3]. While the LLM is used for input generation and 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 [48].

LLAMAFUZZ could outperform traditional tools by reaching a higher code coverage and achieving a higher bug count. Overall, it increased the code coverage by 27.2% in comparison to AFL++ without LLM enhancement and detected 41 bugs more on average [48].

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 by software developers [46]. 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 [47].

[3] https://aflplus.plus/

Zhang et al. [47] 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 6, the prompt consists of a task description for the LLM model, 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 6. Afterwards, one or more fix prompts are applied on cases where the produced LLM output has compilation errors [47].

```
// 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)
```

c — Code kopieren

```
{
    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 6: Generation prompt and output from ChatGPT (adapted from [32, 47])

During the investigation of fuzz driver generation with LLMs, Zhang et al. [47] 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 [47].

LLM-aided fuzzing is already used in practice. In 2023, Google [30] 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 [30].

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 [44]. This causes software developers to question the results of automated vulnerability detectors. Therefore, showing if and how a vulnerability is exploitable is essential [49]. 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 [49].

6.1 Research approaches

In the research of Zhang et al. [49], 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 [49]: *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 compilation errors, a manual process was applied to correct the output.

Fuzzing, which we discussed in Section 5, was used by Zhou et al. [50] in their tool called Magneto. It aims to exploit vulnerabilities in complex environments. 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 incremental, by trying out different seeds, which are created by an LLM. Eventually, it will maybe find an exploit [50].

Fang et al. [14] 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 [14].

As already discussed in Section 4, Fu et al. [15] 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. [15] 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 [15].

6.2 Performance comparison

Zhang et al. [49] 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 [22] and TRANSFER [26]. While TRANSFER [26] achieved writing four exploits, SIEGE [22] could not generate one. This leads to ChatGPT outperforming both tools [49]. Magneto, created by Zhou et al. [50], was at least 75.6% more successful in creating an exploit compared to traditional tools like SIEGE [22], TRANSFER [26] and VESTA [7].

Fang et al. [14] tested their prompt-engineered GPT-4 model against the traditional tools ZAP [4] and Metasploit [5]. 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 [14].

For the severity estimation and mitigation task, Fu et al. [15] 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 Section 2 and discuss the challenges and possible future approaches. The challenges were identified 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 [8, 23, 34, 37] and the low accuracy [15], which was discovered by various researchers.

The recommendation from Cheshkov et al. [8] is to invest further research in prompt engineering. Having a more enhanced prompt, like chain-of-thought, could lead to better performance in the LLM for code vulnerability detection [8]. Jiang et al. [23] propose an iterative process in which the output of the LLM is checked for errors.

An explanation can also be found in hallucination. Hallucination describes the effect of an LLM that writes factually incorrect outputs. A reason for this behaviour could be a lack of knowledge because of missing, biased, or untrue training

[4] https://www.zaproxy.org/
[5] https://www.metasploit.com/

data in the pre-training process [25]. 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. Two examples are uncertainty-aware beam search and confident decoding [24].

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 [8, 15, 43, 44] and exploit generation [14, 50], 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. This is shown by Fang et al. [14], where they had to include the CVE description to gain a well-performing model.

A new approach facing this challenge is RAG. It enables the LLM to access external knowledge which is not included in the training dataset. In this case, an LLM has reference points to gather additional information, which can, for example, be websites, but also databases or other 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 [24].

7.3 LLM Training

Pre-training an LLM is cost- and time-intensive [24]. 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. A modification in one of the three resources could lead to a proportional alteration in a different resource [41].

A common technique used today to bypass pre-training is to take a pre-trained model and to fine-tune it. This allows for better performance in specific tasks compared to larger models [18]. However, it comes with its challenges. One of them is finding correctly labelled datasets. Guo et al. [18] found out that the dataset used for fine-tuning can affect the performance of the LLM. However, they found that the labelling of the dataset is not always correct, 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. [35], it is difficult to determine what prompt leads to better results [24], making it a trial-and-error process.

Another one is the requirements for the setup, which are similar to the ones for pre-training. To fine-tune a model, it must be downloaded, installed and executed [24]. Parameter-efficient fine-tuning (PEFT) is a new way to train the LLM for a specific task. The fundamental concept is to solidify the neural network layers of the LLM and to add a layer at the final layer. During training, only the parameters of the additional layer are modified [36].

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 [24]. A further effect can be seen in fuzzing, since the limited context length understanding also affects the variation of the created input, leading to the generation of similar input [23].

There are different ways to tackle this type of challenge, which may be applied in the future. According to Kaddour et al. [24], current research focuses on implementing efficient attention mechanisms capable of understanding the wider context. Other research is focusing on length generalisation with the aim that an LLM can be trained on short inputs but that it can generalise to understand longer inputs. Kaddour et al. [24] also discussed using alternatives to transformers. Architectures like state space models (SSMs) or receptance weighted key value (RWKV) are designed for understanding longer context.

8 Discussion

In this section, we summarise the key takeaways based on the analysis of the related works and make the connection to the research questions.

RQ1: What are practical use cases for LLMs in security code review and testing, and how do they perform against traditional tools? In Table 2, an overview of the key take-

Table 2: Key takeaways per topic

Topic	Key takeaway
Vulnerability detection	LLM suffers from a high false positive rate (FPR). This limits the capability of LLM in use cases like code vulnerability detection, since a low FPR is preferred to reduce the overhead for the software developer. On the other side, traditional tools are also fighting against FPR and false negative rate (FNR). However, they focus on the true positives, so that software developers can focus on them. While traditional tools had an overall better performance, the idea of using LLM remains interesting since its advantage is the general-purpose design. In comparison, traditional solutions have specific requirements on the technical environment.
Fuzz testing	Using LLM as a fuzzer has 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 addition, LLMs can also be used in mutation-based and greybox fuzzing when using corresponding architectures. Nevertheless, there are different challenges that need to be considered, mainly in the area of the limited context length understanding of LLMs.
Fuzz driver generation	LLMs also have their applicability as fuzz drivers. Especially in autonomous applications like automatically generating fuzz drivers for a wide range of open-source projects. There are also already existing implementations found, which are used in practice. However, same as in the fuzz testing, the LLM reaches its limit as soon as the complexity of the API rises because of the limited context length understanding of LLMs.
Exploit generation	According to the found papers in this area, exploit generation has potential. It is also an important research topic, since it can be used in other areas to diminish false positives, but further investigations have to be made. In comparison to traditional tools, it can outperform them because of its general-purpose design.

aways per topic is provided. The general outcome of these takeaways is that the performance of LLMs still needs to be evaluated and scrutinised in a comprehensive set of circumstances and use cases for their utility to be properly quantified. There are various challenges, as noted in the previous part.

RQ2: What do current approaches look like? We included a range of papers having different approaches for the same aim. To sum up, prompt engineering and fine-tuning are the most important techniques, since they allow guiding the LLM with reasonable effort. Other researchers propose enhanced architectures ensembling different LLMs for different tasks, as well as including numerous prompts in an iterative process.

RQ3: What are the current challenges and what are the prospects for LLMs in security code review and testing? Pitfalls can be found in general LLM challenges like high FPR, training, outdated data, and limited context length understanding. While all topics suffer from those challenges, they are not affected equally.

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. In Fuzzing, FPRs can be mitigated with enhanced controls, like applying software tests. However, here, the limited context length makes it difficult for the LLM to understand complex and extensive software. Exploit generation on the other hand, lives from data actuality, since it needs to know the information about new vulnerabilities.

Research is done to counteract those challenges. Such new approaches are retrieval augmented generation (RAG), dataset sanitisation, model editing, parameter-efficient fine-tuning (PEFT) or / and architectural changes.

9 Conclusion

Practical implementation areas for an LLM in software security can be found in code vulnerability detection using a static code analysis approach, fuzz testing, and exploit generation. The performance varies between different disciplines. In code vulnerability detection, the LLM suffers from a high false positive rate, making it impractical to use. In the other two fields, LLM could outperform traditional tools based on the considered measurements.

Nevertheless, there are different challenges faced when implementing LLMs in this field. Some of them are hallucination, training costs, data actuality, and the limited context length understanding. Prospective research could include futuristic approaches like retrieval-augmented generation, parameter-efficient fine-tuning, model editing, or architectural changes. Further research can be made in understanding the impact of prompt- and fine-tuning on those tasks.

References

[1] Mazin Ahmed. Hacking Zoom: Uncovering tales of security vulnerabilities in Zoom, August 2020. Accessed: 2025-04-04. URL: https://mazinahmed.net/blog/hacking-zoom/.

[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] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] Raphael Hiesgen, Marcin Nawrocki, Thomas C. Schmidt, and Matthias Wählisch. The race to the vulnerable: Measuring the Log4j shell incident. *arXiv e-prints*, page arXiv:2205.02544, May 2022. arXiv:2205.02544, https://doi.org/10.48550/arXiv.2205.02544.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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/.

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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] Bill Toulas. New XZ backdoor scanner detects implant in any linux binary, 2024. Accessed: 2025-04-04. URL: https://www.bleepingcomputer.com/news/security/new-xz-backdoor-scanner-detects-implant-in-any-linux-binary/.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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/.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.