future internet

MDPI

Article

A New Approach to Web Application Security: Utilizing GPT Language Models for Source Code Inspection

Zoltán Szabó * and Vilmos Bilicki

Department of Software Engineering, University of Szeged, Dugonics Square 13., 6720 Szeged, Hungary; bilickiv@inf.u-szeged.hu

* Correspondence: szaboz@inf.u-szeged.hu

Abstract: Due to the proliferation of large language models (LLMs) and their widespread use in applications such as ChatGPT, there has been a significant increase in interest in AI over the past year. Multiple researchers have raised the question: how will AI be applied and in what areas? Programming, including the generation, interpretation, analysis, and documentation of static program code based on promptsis one of the most promising fields. With the GPT API, we have explored a new aspect of this: static analysis of the source code of front-end applications at the endpoints of the data path. Our focus was the detection of the CWE-653 vulnerability—inadequately isolated sensitive code segments that could lead to unauthorized access or data leakage. This type of vulnerability detection consists of the detection of code segments dealing with sensitive data and the categorization of the isolation and protection levels of those segments that were previously not feasible without human intervention. However, we believed that the interpretive capabilities of GPT models could be explored to create a set of prompts to detect these cases on a file-by-file basis for the applications under study, and the efficiency of the method could pave the way for additional analysis tasks that were previously unavailable for automation. In the introduction to our paper, we characterize in detail the problem space of vulnerability and weakness detection, the challenges of the domain, and the advances that have been achieved in similarly complex areas using GPT or other LLMs. Then, we present our methodology, which includes our classification of sensitive data and protection levels. This is followed by the process of preprocessing, analyzing, and evaluating static code. This was achieved through a series of GPT prompts containing parts of static source code, utilizing few-shot examples and chain-of-thought techniques that detected sensitive code segments and mapped the complex code base into manageable JSON structures.Finally, we present our findings and evaluation of the open source project analysis, comparing the results of the GPT-based pipelines with manual evaluations, highlighting that the field yields a high research value. The results show a vulnerability detection rate for this particular type of model of 88.76%, among others.

Keywords: large language models; GPT; sensitive data; vulnerability detection; CWE-653; Angular; static code analysis

Citation: Szabó, Z.; Bilicki, V. A New Approach to Web Application Security: Utilizing GPT Language Models for Source Code Inspection. *Future Internet* **2023**, *15*, 326. https://doi.org/10.3390/fi15100326

Academic Editor: Hui Tian

Received: 14 August 2023
Revised: 25 September 2023
Accepted: 27 September 2023
Published: 28 September 2023

1. Introduction

In the ever-changing landscape of contemporary software development, the intricate web of code that drives our digital experiences frequently conceals hazards. As web applications become more complex, so do the vulnerabilities that threaten their security. Front-end frameworks such as Angular [1] have ushered in new development paradigms, but with them come difficulties in ensuring data security and implementing appropriate access management, especially when automation is required, reducing human interaction. Large language models (LLMs) such as the various GPT models [2] and ChatGPT [3] have risen to prominence against this backdrop, promising to revolutionize duties such as code generation, documentation, and debugging. However, their true potential for comprehending, evaluating, and improving code is still the subject of intense debate and

investigation. Especially pressing is the question of whether or not these models can discern the nuances of code to identify potential vulnerabilities and weaknesses. This is particularly true in the vast landscape of complex modern web frameworks, where improper solutions might result in the leakage of confidential data.

1.1. AI in Software Development

The first focus of our inquiry is directed towards the involvement of artificial intelligence, particularly large language models, within this domain, drawing upon our fundamental understanding of the evolving landscape of software development. In recent years, the development of large language models (LLMs) has accelerated substantially, and in the past year, both their popularity and adoption have attained new heights due to popular services such as ChatGPT [3] and its competitors. Numerous research groups are currently investigating the precise impact of the use of these models in various fields and the extent to which they can be used for process optimization and decision support improvements, with a particular focus on the IT sector and programming, in addition to education, healthcare, administration, and business. While more research is focused on the quality and optimality of the code generated by generative pre-trained transformers (GPTs),the exact limitations of the models in terms of comprehension and interpretation remain unclear. There is a similar challenge with regard to interpretability, or the depth to which a large language model can comprehend the function and purpose of the code in order to test, debug, or even enhance it with minimal developer intervention.

This kind of AI-assisted debugging and code interpretation is particularly intriguing. With the right semantic and logical knowledge, such systems are likely able to detect potential sources of errors in the code that might otherwise only be discovered through extensive testing. Another branch of our research team has already obtained promising results with GPT-assisted static code analysis of Angular applications [4], achieving better results on the problem set under investigation than with BERT, using only prompting and no additional training of models. Trends in recent years suggest that testing itself is becoming an area where AI-enabled tools are becoming increasingly popular [5]. The significance of aggregating and making the code base more transparent and manageable for large, complex code bases in order to effectively manage configurations such as vulnerability analysis and security configuration enforcement is not a new concept. Heydon et al. [6] introduced Miró, a set of visual description languages, in 1990 to make security configurations of software systems more straightforward and transparent. A similar motivation led Giordano and Polese [7] to introduce the Vicoms framework, which allowed developers to administer fine-grained, role-based access control settings for their Java applications in a manner that was distinct from the source code but still readily comprehended. However, the implementation and utilization of such a complex tool, as well as its testing, are invariably subject to the quality of the aforementioned code interpretation.

1.2. The Challenges of Contemporary Web Frameworks

After discussing the capabilities and potential of LLMs in software development, it is necessary to explore the complexities of contemporary web frameworks. Angular, for example, presents a paradigm that, while potent, presents its own set of difficulties. Understanding these complexities is crucial because it lays the groundwork for how LLMs can be utilized effectively to resolve them. The architecture of modern web frameworks, like Angular [1], introduces unique challenges. For instance, building a DOM structure that aligns with the call graph and dependency graph of single-page applications becomes difficult due to modular nesting and partial navigation of components. Even the very first version of Angular was challenging from an analytical standpoint. Misu et al. [8], for instance, created FANTASIA, an AST-based utility for analyzing AngularJS MVCs, to aid developers in detecting inconsistencies in static source code. The utilization of high modularity, typology, and object-oriented paradigms in TypeScript has proven to be advantageous for debugging and design purposes. However, the enhanced modularity,

intricate call and dependency graphs, and complex navigation have posed challenges for both static and manual analysis, as is the case in other contemporary front-end frameworks.

In the past, our research team has participated in the development of several healthcare applications and, while working on these solutions, investigated the issues of access management and data security [9], with an emphasis on optimizing the access control implementation. The main objective was to effectively and expeditiously obtain the required information from the databases, while adhering to ethical guidelines and ensuring that access to the information was limited to authorized organizations.However, there were limitations at the end of the data path for the front-end applications, as quantifying the security of the applications or web applications located there would be required to evaluate their security, while this information is necessary to evaluate the security of the entire data path. While we focus on Angular, the methodology presented in this paper should be adaptable to other frameworks for similar problems. The foundational concepts of our approach—mapping the code base, executing the steps of vulnerability detection, and aggregating partial results—are programming-language-agnostic. They can be applied to various back-end and front-end frameworks.

The common weakness enumeration (CWE) [10] is a standardized catalog of potential software and hardware design vulnerabilities compiled by the MITRE Corporation. This comprehensive framework provides the technological universe with a consistent method for identifying and addressing these vulnerabilities. Using CWE's guidance, automated tools are adept at quickly identifying a multitude of vulnerabilities. Nonetheless, the nuanced nature of some vulnerabilities prevents automated detection, particularly when they are profoundly intertwined with the code's semantics. In addition, comprehending the vulnerabilities in certain applications requires more than just code analysis. Potential weak points can be affected by the character of the data being processed, as well as its origin, flow, and final destination. An application that handles sensitive medical records, for instance, may have vulnerabilities that are only apparent when the regulatory and ethical implications of data handling are considered [11]. Similarly, applications operating in a particular cultural or geopolitical context may be vulnerable to unique threat vectors. This highlights the utmost significance of human expertise, as specialists not only interpret code but also comprehend the larger ecosystem within which an application operates. As cyber threats continue to adapt and evolve, the relationship between an application's context, the data it processes, and its code becomes increasingly complex. Combining automated tools with the profound contextual insights of human experts remains the most comprehensive method for protecting systems, representing the pinnacle of vulnerability detection and remediation.

The "improper isolation of compartmentalization" vulnerability, identified as CWE-653 , is prevalent in front-end applications. This vulnerability for web applications means that certain critical processes or operations are implemented in the code with improper isolation. Consequently, if the same piece of code is equally responsible for handling critical, sensitive data and less relevant data, there is a risk of significant data leakage or unauthorized use due to developer inattention or malicious user activity, as defined in CWE members of the CWE-200 ("exposure of sensitive information") category, with members that are easy to encounter in front-end applications such as CWE-203 ("observable discrepancy"), CWE-359 ("exposure of private personal information to an unauthorized actor"), and CWE-497 ("exposure of sensitive system information to an unauthorized control sphere"). The foundation of our approach was the identification of sensitive data, which should be accorded a high level of importance in the source code with appropriate isolation, protection, and access control; failure to do so can indeed result in significant misuse. In the case of progressive web applications, we have assumed that such data originates from a server or database connection, through which it is passed to the web application; therefore, the key to protecting it is the proper protection of the entities that manage these operations, in the case of the Angular framework, the service classes, which in many cases implement the classic data object access (DAO) pattern.

Such services operate as singletons within the Angular framework and are accessed and invoked via injection from component classes responsible for each element of the user interface or from each other. This paper's research questions evolved from the question of whether the component classes that invoke these services have appropriate access control, which in the instance of the Angular framework is controlled via the AuthGuards, classes implementing the CanActivate interface, which, when attached to the navigation rules within the application, can prevent or redirect a user from accessing user interfaces they are not authorized to access.

1.3. Sensitivity and Potential Vulnerabilities

After examining contemporary web frameworks, we move on to a crucial topic: data sensitivity and potential vulnerabilities. In the domain of web applications, especially those developed with frameworks such as Angular, the management of sensitive data is of the utmost importance. The distinction between varying degrees of sensitivity and the potential impact of data leakage is essential for the development of secure and robust software. We encountered a notable obstacle: Is it possible to classify sensitive data and measure its level of sensitivity? It is indisputable that the technological advancements of the past decade, not to mention the various scandalous violations of personal data by large technology companies against their users, have raised an entirely new set of concerns regarding how we comprehend, handle, and expect our personal data to be handled. In the past few years, numerous studies have examined issues such as how we understand the concept of sensitive data in different domains [12], how the concept of personal sensitive data has evolved in the first place, and its various degrees [13].

Data collection parsimony, which sets a metric for machine learning algorithms on how little data they require for efficient learning and operation [14], has become a dominant trend, as have categorizing and quantifying the sensitivity of data [15,16] and the automated recognition of sensitive data [17]. In our proposed categorization, detailed further in the Methodology section, we divided sensitive data into three categories based on the potential damage from disclosure. While at the bottom of the scale we can only speak of potential damage through the accumulation of a large volume of leaked data, the data at the top of the scale is vital information of its own. Although the categorization of such a method set can be arbitrary for a given development team, we wanted to create a scale for large language models that is easily defined and can be described using the chain-of-thought principle.

The primary findings of this paper are as follows:

- Definition of a proprietary classification to quantify the sensitivity and immunity of the data and evaluating it on a test set of 200 variable names using the GPT API. This classification provides a robust framework for comprehending and classifying data according to its sensitivity. By evaluating it using the GPT API, we ensure that the model recognizes and processes diverse data categories appropriately, thereby making our approach more applicable to real-world scenarios.
- Demonstration of prompts generated using prompt engineering for the GPT API for static analysis of the code of a complex web application, taking the larger context and deeper context into account. We maximize the potential of the GPT models through prompt engineering, ensuring that the static analysis is not only accurate but also context-aware. This method bridges the divide between generic AI analysis and specific software vulnerabilities, yielding both broad and profound insights.
- Evaluation of the efficacy of the GPT-3.5 and GPT-4 models in detecting sensitive data in an application via static code analysis. By comparing the performance of various variants of the GPT models, we seek to discern the evolution and enhancements in these models over time. This evaluation sheds light on how advancements in LLMs can further strengthen software application security.
- Evaluation of the ability of the GPT-3.5 and GPT-4 models to determine the protection levels of front-end application elements. In addition to detecting sensitive data, it is essential to ensure that it is adequately protected. By assessing the GPT models' ability

to estimate protection levels, we can better comprehend their potential role in security assessments and potential interventions.

- Evaluation of the effectiveness of the GPT-3.5 and GPT-4 models in detecting when sensitive data handling in a web application is not adequately isolated and protected, thus producing a possible vulnerability. This evaluation is essential to comprehending the practical applicability of LLMs in cybersecurity. If these models can reliably identify data management vulnerabilities, they could become indispensable for application developers seeking to fortify their applications against intrusions.

We begin with a literature review in Section 2. Then, in Section 3, we present our methodology, in which we attempted to identify parts of the code handling sensitive data, categorize their sensitivity, determine their protection levels, and detect when the sensitivity level and protection level are inconsistent by executing an analysis pipe supported by GPT API calls on a set of open-source Angular applications randomly selected from GitHub. In Section 4, following the presentation of our results, we discuss the interpretation of the errors and anomalies found in the analysis as well as the improvements obtained by optimizing our method. In Section 5, we discuss potential issues that threaten the validity of our results. Lastly, in Section 6, we discuss the future research potential of our developed method.

2. Related Works

When we started to develop our own methodology, our first step was to assess the state of the art of exactly what others have achieved so far in bringing artificial intelligence into the domain of programming. We divided the papers we found into three groups, the first group being those that have achieved significant results in improving the quality of program code and in helping people write code using AI. In the second, we included papers that specifically evaluated the code-understanding capabilities of LLMs. Finally, the third group includes the papers most relevant to our own research, which, like ours, attempted to detect various vulnerabilities using LLM models.

2.1. The Impact of Artificial Intelligence on Software Development

The rapid growth in the use of artificial intelligence in programming did not follow the rise in prominence of GPT, but its potential was recognized by a large number of people long before that.

In their 2019 paper, Jiang et al. [18] investigated how, for instance, they could enhance the readability of program code using machine learning to suggest method names based on their root, using a state-of-the-art approach titled code2vec. They found that code2vec's performance drops in more realistic settings, with many successful recommendations being trivial, and a proposed simpler heuristics-based approach outperformed code2vec.

Momeni et al. [19] used machine learning to detect vulnerabilities and bugs in smart contracts on the Ethereum chain. They achieved a 95% accuracy rate in their experiments, identifying 16 different types of bugs. Their analysis demonstrated that their method was faster, more efficient, and simpler compared to other static parsers.

Mhawish and Gupta [20] concentrated on detecting code smells, and with the various machine algorithms evaluated and validated, they obtained over 90% accuracy in detection rate, which has tremendous potential for enhancing code quality and AI-based programming.

Using artificial intelligence, Cui et al. [21] addressed the vulnerability issues of IoT systems based on Android by conducting an empirical analysis of security risk prediction with six machine learning techniques. Their results indicate that machine learning algorithm prediction performance varies, with the random forest (RF) algorithm emerging as the most effective across all risk levels.

In a similar manner, Park and Choi [22] addressed the expanding cybersecurity concerns in vehicles with open connectivity. Their study proposes a machine-learning-based method for detecting anomalous behaviors caused by malware in large-scale network traffic in real time. Their experimental results indicate that static code metrics have the

potential to predict Android risk vulnerabilities for IoT applications, despite the current limitations of small and imbalanced datasets.

Another research project, [23], CURE, strengthens the category of automatic program repair (APR), which aims to automatically detect, patch, and debug errors based on static program code. It has significantly exceeded the capabilities of similar existing solutions.

In the comprehensive review by Sharma et al. [24], in addition to categories such as program interpretation and classification of code quality, the identification of vulnerabilities, which corresponds to our research area, was highlighted. The paper also emphasizes perceived challenges in the field, such as dataset standardization and reproducibility, and suggests opportunities for integrating machine learning techniques into software engineering applications more effectively.

2.2. The Code Interpretation Capabilities of GPT

The emergence and dissemination of GPT and other transformer-based neural networks and large language models, which have demonstrated exceptional results in code generation, code interpretation, documentation, and automated debugging, marked a turning point in this research domain.

A comprehensive evaluation by Sarkar et al. [25] examined the efficacy of various LLMs for programming. They concluded that AI-assisted programming is an entirely new form of programming with its own challenges and complexities, and as part of their research, they considered, among other things, how proficiently a neophyte developer could program with the assistance of these tools.

Wei et al. [26] focused their research on the evaluation of the capabilities of LLMs, which surprised the research community as their emergence did not result from the scaling of the capabilities of lesser language models. According to them, one of the primary directions of NLP research is to determine how these emergent abilities will continue to change and evolve as the models continue to scale.

Liu et al. [27] aimed to provide a comprehensive overview of researchers' use of ChatGPT and GPT-4 in their work. They based their analysis on a total of 194 papers found in the arXiv database. In their summary, they highlighted the potential of using the models and reinforced learning from human feedback (RLHF), as well as the numerous concerns regarding the ethical use of the models, their potential for producing detrimental content, and their potential privacy violations.

The research of Surameery and Shakor [28] has already centered on evaluating ChatGPT's ability to effectively repair flaws in response to prompts. While they believe that the effectiveness and limitations of ChatGPT will largely depend on the quality of the training data and the types of bugs to be fixed, ChatGPT will undoubtedly be a very important and relevant tool in this regard, and its strength will be bolstered by the use of other debugging and analysis tools as support.

Using pre-defined queries, Borji and Mohammadian [29] benchmarked the largest LLMs, such as GPT-4 and Bard, in their work. Based on their findings, the GPT-4 model appeared to be the most reliable for tasks centered on software development, code generation, and code comprehension, with great potential for usage in more complex scenarios.

2.3. Vulnerability Detection with Large Language Models

Naturally, we are not the first to attempt to use the interpretive capacity of large language models to examine program code vulnerabilities.

In their 2021 literature review [30], Wu presented research on software vulnerability detection based on BERT and GPT. Common among the presented efforts was the segmentation of source code, followed by the extraction of features, and the fact that the introduction of specific vulnerabilities was one of the final fine-tuning steps.

Thapa et al. [31] investigated the efficacy of large transformer-based language models, such as GPT-2, in detecting software vulnerabilities in C/C++ source code and demonstrated that these models outperform conventional models such as BiLSTM and BiGRU on

a variety of performance metrics. It is important to note that implementing these language models presents challenges, such as GPU memory constraints and the complexities of managing model parallelism and dependencies when refining and running them locally.

The study by Omar [32] presents VulDetect, a transformer-based vulnerability detection framework. It fine-tunes a pre-trained language model (GPT) to identify vulnerable code, achieving up to 92.65% accuracy. This outperforms current state-of-the-art techniques such as SyseVR and VulDeBERT. However, the practicality of this approach might be questioned. Traditional deep learning models like CNNs and LSTMs, which their research aims to surpass, are known for their high computational needs. This makes them less ideal for real-time applications.

Sun et al. [33] observed that many studies, like the ones described above, do not leverage the full capabilities of GPT-type models. These models can interpret and evaluate domain-specific information. Instead, the focus has mainly been on identifying vulnerabilities through control and data flow analysis. They presented GPTScan, which searched the source code of smart contracts for logical inconsistencies with 90% accuracy. However, while GPTScan achieves high precision for token contracts, it only achieves "acceptable" precision (57.14%) for larger projects like Web3Bugs.

Cheshkov et al. [34] attempted to detect the most prevalent CWE vulnerabilities in open-source Java applications in a manner similar to ours by prompting GPT-3.5 and ChatGPT. However, their work was based on rather naive assumptions about the existing knowledge set of GPT; they only mentioned prompt engineering techniques in their conclusions, and their results were not particularly convincing, performing essentially on par with their benchmarking dummy classifier.

Feng and Chen [35] attempted a similar experiment with ChatGPT, although they have improved their approach by using chain-of-thought and few-shot example prompt engineering techniques, showing promising results in creating an Android bug replay automation tool supported by ChatGPT. Still, the current automated approaches to bug replication are limited by the limitations of manually-crafted patterns and pre-defined vocabulary lists, which determine their success based on the characteristics and quality of defect reports.

A comparison of the mentioned works is given in Table 1.

Table 1. List of related works with the most relatable solutions to our own research.

Author	Models	Subject	Weaknesses
Wu [30]	BERT and GPT	Software vulnerability detection	Relies on segmentation of source code and feature extraction
Thapa et al. [31]	GPT-2	Software Vulnerabilities in C/C++ source code	GPU memory constraints, managing model parallelism and dependencies
Omar [32]	GPT-2	Vulnerabilities in C/C++ codebases	High computational burden of traditional deep learning models like CNNs and LSTMs

Table 1. *Cont.*

Author	Models	Subject	Weaknesses
Sun et al. [33]	GPT-type models	Smart contract source code	Only "acceptable" precision for larger projects like Web3Bugs
Cheshkov et al. [34]	GPT-3.5 and ChatGPT	CWE vulnerabilities in Java applications	Naive assumptions about GPT's knowledge, results on par with dummy classifier
Feng and Chen [35]	ChatGPT	Android bug replay	Limited by manually crafted patterns and pre-defined vocabulary lists

This literature review indicates that GPT models have more promise. Nevertheless, rather than using ChatGPT, we made the decision to use API calls directly. The use of this methodology enabled us to establish the desired temperature, thereby mitigating non-deterministic tendencies. Furthermore, we used quick engineering methodologies in order to improve the precision of the generated results. However, the implementation of these tactics in isolation did not provide a definitive assurance of the effectiveness of the approach. Web applications provide a more sophisticated domain as a result of their intricate behaviors. The accurate interpretation of the behavior and context of the program code played a crucial role in identifying the vulnerabilities that were the focus of our investigation.

3. Methodology

3.1. Categorization of Sensitivity and Security

Our methodology is predicated on the notion that in Angular web applications, sensitive data are centralized in services, which are accessed and utilized as singletons by Angular framework classes via their methods. In order to discover sensitivity levels and vulnerabilities in static code, our first step was the detection of elements inside the component classes that exhibit indications of sensitivity. This was accomplished by using the capabilities of contextual understanding of the GPT models. Subsequently, we proceeded to determine which of these elements are managed by services.

To classify the sensitivity level of the data, we considered the attempts to categorize the data [15,16] mentioned in the Introduction, but our primary metric was the potential harm to users if unauthorized individuals accessed the data. Based on this, we divided the sensitive data into three levels. Motivated by the need to provide a categorization whose levels build upon one another, the classification of data is based on logical reasoning. This approach allows us to debug the thought process of the large language model. We can then improve or validate it following the chain-of-thought principle.

The categories are defined as follows:

- **Level 1 (Low Sensitivity)**: The accumulation and compilation of large quantities of data at this level is required to infer confidential information and create abuse opportunities. Sensitive information includes, among other things, a user's behavior history, a list of websites visited, products and topics of interest on a website, and search history.
- **Level 2 (Medium Sensitivity)**: By obtaining data at this level, it is possible to retrieve and compile potentially exploitable information. Solutions such as two-factor authentication, regular email and SMS notifications, and exhaustive logging in to affected applications can mitigate the effects of a potential breach. This is the largest and most

extensive group. It includes data such as usernames, passwords, private records, political views, sexual orientation, IP address, physical location, and hectic schedules.

- **Level 3 (High Sensitivity)**: The data at this level is sensitive in and of itself, and its acquisition or disclosure can have severe legal repercussions and cause extensive harm. Examples include health information, medical history, social security number, driver's license, and credit card information.

As stated in the Introduction, this categorization, which is based on a rationale for the level of data sensitivity for large language models, is anticipated to function well in arbitrary applications but may not be ideal for all development teams. In certain instances, it may require some refinement, perhaps by providing more specific examples for each level in the prompts. However, we hope that such fine-tuning or even the use of a more specific or different sensitivity scale will not be a problem for GPT's ability if our method works and can find and classify sensitive data efficiently and with good reasoning.

To validate the categories developed and the analysis capabilities of the GPT-3.5 and GPT-4 APIs, which at the time of writing were the most efficient GPT models available and promptable via API, we compiled a dictionary of 200 variable names, equally divided among the non-sensitive, low-sensitivity, medium-sensitivity, and high-sensitivity categories. In preparing the dictionary, we introduced challenges for the models. Some variant names had deliberate misspellings, some were in foreign languages like Hungarian, German, or Italian, and some combined sensitive words with less sensitive endings.

The GPT-3.5 API incorrectly classified 45.5% of the dictionary, with the majority of cases being trivial. In contrast, GPT-4 only made errors in 23.5% of the cases, and a deeper examination of the errors revealed that it only failed on the explicitly difficult terms, and in more than half of the cases, it correctly identified the errors as sensitive data but incorrectly categorized the exact category. Although both models were used throughout our investigations, preliminary results suggested that GPT-4 might be able to produce more accurate results by analyzing codes, whereas GPT-3.5 appeared more prone to failure in all but the most straightforward cases.

The aim of this preliminary assessment was not to successfully evaluate a domain-specific set of variables with the models. We merely wanted to see how effectively the models could understand the rules outlined and to assess in advance whether there were any significant differences in the interpretive abilities of the two models. A more extensive or domain-specific dictionary in terms of methodology and experiments would not have changed the assumptions reached here.

Additionally, we have sought to align the application's protection scale with the three-level sensitivity scale. Although this alignment assumes some level of security measures in the data path to the front-end application, such as two-factor authentication, front-end applications might offer supplementary measures that can help mitigate the risk of unauthorized access, especially when server-side and data path protections have vulnerabilities. These include isolation of functionality combined with strict role handling and access control.

The scale presented is as follows:

- **Protection Level 0**: The component has no protection at all.
- **Protection Level 1**: Bare-bones authentication: the application checks whether the user trying to access the resource is logged in to the application.
- **Protection Level 2**—RBAC [36]: In addition to being logged in, the user has different roles that define their scope of privileges within the application.
- **Protection Level 3**—ABAC [37]: In addition to the login and possible role scopes, other attributes such as time, physical location, or type/ID of device used are checked before granting access.

In our experiments, we attempted to identify instances in which the AI-based analysis of the source code produced pairings in which these two values did not match, i.e., in accordance with the definition of CWE-653 failure, a service dealing with sensitive data could be

accessed and called by a component with a protection level lower than the sensitivity level of the data due to a lack of proper isolation.

3.2. Evaluation Dataset

Our research team collected thousands of public Angular projects from GitHub using a crawler algorithm in order to conduct various source code analyses. We minified the TypeScript source files of the projects by labeling, removing whitespace characters, and removing line breaks, and then randomly sampled 12 of the largest, most complex, and largest projects. The imposition of this constraint stems from the findings of our prompt testing and development process. It was observed that evaluating numerous projects across the entire range of prompts would nearly deplete the monthly hard limit of the GPT API [38]. Additionally, our schedule for the current research phase imposed further restrictions on the number of possible projects.

In the Threats to Validity section, we will discuss how this may endanger the validity of the conclusions, as there may be individual sources of problems and anomalies that necessitate additional investigation. However, if the methodology proves effective on such large and complex codebases, we anticipate that fine-tuning the prompts may be sufficient to handle potential anomalies, while the preparation of the codebase, the validity of our methodology, the outlined steps, and the ability of the GPT models to interpret static code will remain unchanged while being able to handle specific cases. The random selection was modified with the constraint that the 12 selected projects must contain non-English terms, variable names, and class names to support the hypothesis, confirmed during the dictionary evaluations, that GPT can effectively detect sensitive information regardless of its language, given the appropriate context.

OpenAI's privacy policy [39] indicates that they prioritize user privacy. According to them, OpenAI does not store for long-term use the queries sent to the GPT API. However, when transmitting sensitive data, users should always be cautious and consult OpenAI's most recent terms of service and privacy policies to ensure they are abreast of the most recent practices and protocols. This is why we decided to use open-source Angular applications instead of our own healthcare applications for the experiments. If other researchers wish to conduct similar investigations, this may be a crucial factor for them to consider. As ChatGPT and other GPT models continue to gain popularity, it is hoped that OpenAI will soon adopt significantly more stringent and robust privacy principles that permit the analysis of private code bases.

3.3. GPT API and Evaluation

OpenAI's generative pre-trained transformer (GPT) series represents cutting-edge neural language models [40] that leverage deep learning for a variety of natural language processing tasks. Based on a transformer architecture, these models are capable of identifying intricate patterns in enormous quantities of text data. GPT-3.5 and GPT-4, the more advanced iterations, were trained using unsupervised learning on vast datasets, making them adept at generating coherent and contextually pertinent content across diverse domains. These models' capacity to analyze and interpret program code is a significant strength. Due to their extensive training data, which includes immense programming contexts, they are ideally suited for tasks requiring a comprehensive comprehension of code semantics and syntax.

Utilizing the official GPT APIs provides numerous benefits over local deployment and training. The execution of models within the scope of the GPT requires substantial computational resources, which can be prohibitively expensive, resource demanding, and technically difficult to manage, as explained in detail by Shoeybi et al. [41]. Training or refining models such as these from inception necessitates access to massive datasets, which may present difficulties in terms of data acquisition and processing. Utilizing the API simplifies these complexities, allowing users to leverage the model's capabilities without the burden of infrastructure and data management. In addition, the use of APIs permits the

setting of the temperature level, whereby the generated responses can be forced to behave in a factual, meaningful, non-random manner, precisely adhering to the rules passed in the prompt and excluding the so-called "hallucinations" [42], as opposed to ChatGPT, which has a significantly higher non-deterministic factor.

3.4. GPT-Driven Detection Pipeline

The pipeline's prompts were developed using the principles of prompt engineering [43], a relatively new discipline, through multiple cycles of experimentation; the used prompts are included in the Appendices of this paper. Criteria included plain wording, the inclusion of rules prohibiting the reappearance of anomalies discovered during initial testing, and the inclusion of examples pertinent to the expected response format. In order to achieve this objective, we used bigger and more complete prompts, enabling us to circumvent issues concerning the output formats. This was further improved upon by the incorporation of prevalent prompt engineering approachessuch as "few-shot learning" [44] and the chain-of-thought principle [45]. Few-shot learning refers to a prompting technique whereby a few (usually one to five) samples are passed to the LLM model as part of the prompt for the correct output. This may cover the correct output format or possibly an appropriate classification. Its purpose is to trigger further teaching on the basis that the model itself already has the teaching needed to work correctly, so it merely needs to be tuned to use it as required. Chain-of-thought, on the other hand, is a technique whereby structured sample prompts are passed to the LLM so that it can make more logical and accurate decisions. It is typically used in conjunction with the few-shot example. Although this significantly increased the size of the prompts and we were initially concerned about exceeding the 8096 token limit of GPT-4 (in the case of GPT-3.5, we were not at risk of exceeding it due to the 16,384 token limit of gpt-3.5-turbo-16k), a survey conducted prior to the actual evaluations to measure the maximum size of the prompts and input files dispelled such concerns. The results are shown in Table 2, and the prompts and precise methodology are described in the sections that follow.

Table 2. The token counts of the various prompt parts used in the GPT API requests.

Prompt Type	Token Count
Sensitivity detection prompt (Appendix A.1)	905 tokens
Feature extraction prompt (Appendices A.2 and A.3)	1564 tokens
AuthGuard and protection level evaluation prompt (Appendix A.4)	2354 tokens
Average token count of the largest files from the largest projects	3560 tokens
Average token count of the largest mapped JSON	1145 tokens

In addition, it is important to note that, at the time of writing, GPT-4 has already begun to implement its extended model with 32,000 tokens, which will permit the delivery of significantly larger code segments or even more complex prompts than those under consideration, thereby mitigating this risk [46].

The scripts that make up the analysis pipeline are included as Supplementary Material S0–S5. Of these, S0 contains the dictionary analysis code, and S5 contains the readme file describing the exact dependencies and the proposed running order of the scripts.

The complete analysis process consists of the following steps:

1. **Minifying Code Base**. **Input:** The raw project folder and files; **Output:** A single .txt file per project, containing the relevant source files without line breaks and whites-

pace characters; **Prompt Appendix:** -; **Supplementary Material:** S1; **Description:** A transformation step to prepare the project files to be included in the prompts.

2. **Sensitive Element Detection**. **Input:** The minified .txt file of the project; **Output:** A CSV file with the sensitive elements from the projects and their sensitivity levels; **Prompt Appendix:** Appendix A.1; **Supplementary Material:** S2; **Description:** The analysis first identifies the sensitive data, its sensitivity level, explains how it appears in the application, how it is used, and then an aggregation script uses this information to select the services that were involved in read or write operations on sensitive data in the project.
3. **JSON Mapping of Project Files**. **Input:** The minified .txt file of the project; **Output:** An array of JSON objects following the template of Appendix A.2; **Prompt Appendix:** Appendix A.3; **Supplementary Material:** S3 contains the script, S3_1 contains the prompts; **Description:** A JSON file is created for each project, in which the components of the application are reduced to JSON objects, containing only the most necessary information.
4. **Protection Level Discovery**. **Input:** The array of JSON objects mapped from the project; **Output:** The extended JSON list, containing the protection levels; **Prompt Appendix:** Appendix A.4; **Supplementary Material:** S3 contains the script, S3_1 contains the prompts; **Description:** The JSONs and the relevant routing and guard configurations are passed one by one to the GPT API, which adds the protection level corresponding to our scale.
5. **Vulnerability Detection**. **Input:** The CSV file from Step 2. and the expanded JSON file from Step 4.; **Output:** An Excel file containing the sensitivity levels, protection levels, and detected vulnerabilities for each project; **Prompt Appendix:** -; **Supplementary Material:** S4_1–S_2 contain the scripts for the various Excel file generations, S4_4 is the vulnerability detector, but it needs to have the results of the previous S4_X scripts; **Description:** By aggregating the results, the vulnerabilities from the tested projects are detected.

In every instance, the results of the experiments were evaluated manually. Members of our research team compared the output generated by each phase to the original, unadulterated program code from which it was generated. During the evaluation, we distinguished between two critical error categories from a methodological point of view: missing information, where, for example, a program component was not detected or recognized; and misidentification, where detection was successful but the information was incorrect or incomplete. More detailed explanations and outputs of the steps are presented in the following subsections.

3.4.1. JSON Mapping of Project Files and Protection Level Discovery

This step aims to distill the project's source code to its most crucial elements. This reduces the token count for future operations and improves the evaluation quality by focusing only on the most relevant information for subsequent prompts. The step is similar to the code reduction or feature extraction phase frequently mentioned in the literature [31–33]. The output format is detailed in Table 3.

During the JSON mapping of project files phase, the guarded, guards, and guardlevel fields continue to be populated with empty values. During the protection level discovery step, the already mapped JSON files are passed along with the AuthGuards and routing configurations to the GPT API in order to determine the protection levels based on the interpretation of the AuthGuards' functionality and mappings.

Table 3. The output structure of the JSON_mapping operation.

Name	Content
Type	The type of the analyzed file, component, service, etc.
Selector	If the type is component, this field contains selector string specified in the decorator method of the class, which indicates the rendering locations in the html template codes which are handled by the component.
Name	The name of the analyzed class.
File	The name of the analyzed file.
Path	If the parsed file type is component, the string of the application's routing configuration to access it; empty string for embedded child components only.
Guarded	Boolean value indicating the presence of an AuthGuard or AuthGuards protecting the class if it is a component.
Guardlevel	A numeric value in the 0–3 interval according to the security levels defined in the previous section.
Guards	The names of the AuthGuard classes protecting this class if it is a component.
Navigations	A list containing the names of the components to which we can navigate forward from this class.
Injections	A list of injected services in the class, also containing the exact interactions with them which might be asynchrons, attributes, or functions.
Parents	The list of components, which contain the selector of this class (if it is a component) in their html template.
Children	If the analyzed file is a component, then a list of the components that are contained in its html template by their selectors.

Our goal was to design a format that extends beyond security risk analysis. If our research shifts to other aspects of static analysis in the future, this methodology can still be applied. It could lead to alternative schematic representations of the codebase, such as visual diagrams showcasing dependencies, navigations, and security configurations. This concept mirrors visual description languages like [6,7] mentioned in the Introduction.

In Appendix A.3, we have included an example output of the JSON mapping prompt.

3.4.2. Sensitive Element Detection

As input, the step also receives the component's minified source code, which is embedded in the corresponding prompt. The intent was for the GPT API to mark in the received source code the variables and objects that it identified as sensitive, classify them according to the sensitivity levels, and provide explanations for tracing back the findings in the event of a problem. The output format, as depicted in Table 4, employed the CSV format for the purpose of facilitating manual validation. This format enabled the utilization of the type field, which allowed GPT to generate a concise string that precisely identified the nature of the sensitive data, such as confidential user data, company information, or financial infor-

mation. Additionally, the function field contains a more elaborate description, elucidating the rationale behind designating the information as sensitive. For instance, it could include explanations such as "This pertains to the device data collected for payment processing" or "It consists of an array of Materia objects containing educational records".

Table 4. The output structure of the sensitivity detection operation.

Name	Content
Element	The name of the sensitive object or variable.
Type	The type of the sensitive element, following a conviction similar to the list in prompt Appendix A.1.
Level	The sensitivity level of the detected element.
Origin	Userinput, service, or localstorage, the three most common sources of sensitive data in a web application.
ServiceName	If the origin is service, then the name of the service which is the source of the sensitive element.
Function	The role of the sensitive element in the component where it was detected.
Goal	The fate of the sensitive element after its usage: stored, read, or service.
GoalName	If the value of the goal field was service, then the name of the service to which the sensitive element is passed to.
Classname	The name of the class in which the sensitive element was detected.

3.4.3. Vulnerability Detection

This step consists of a simple aggregating Python script that reads the JSON files containing the protection levels and the CSV file containing the list of sensitive services,then identifies the vulnerabilities based on these files. The output lists all application components, their protection levels, detected sensitive services, and the highest sensitivity level among them. If a component's protection level is lower than its sensitivity level, it ismarked as potentially vulnerable. The results must bemanually inspected.

4. Results and Discussion

Through the evaluations we sought answers to the following research questions:

- **RQ1**: How effectively can we identify sensitive data and services based on context?
- **RQ2**: How effectively can we detect component protection levels?
- **RQ3**: How accurately can we identify vulnerabilities belonging to the CWE-653 category?

The analysis of runtimes and resources was not conducted for the obtained findings. The performance of runtimes is significantly impacted by the load on the GPT API and the network throughput observed during the experiment.In terms of resource requirements, the running tools only needed to read the minified text files, CSV files, and JSON files, outputting the responses in the appropriate formats. The runtime of each step (which includes the sum of the response times of all the prompts for that step) is shown in Table 5. These results are included to demonstrate that the analyses are not particularly lengthy, running in less than 10 min even on the largest projects evaluated, and that there was no significant difference in processing times depending on whether the scripts were run on a more powerful desktop PC or an older, less powerful laptop, since the models were not run on them but accessed via the network as APIs.

Table 5. Average response times for the various prompts from the GPT-4 and GPT-3.5 APIs.

Step	Average Runtime with GPT-4	Average Runtime with GPT-3.5
Sensitivity Element Detection	216.846 s	194.266 s
JSON Mapping of Project Files	217.334 s	163.882 s
Protection Level Discovery	247.904 s	140.559 s

4.1. RQ1

The first evaluation addressed the effectiveness of detecting sensitive services. The results of GPT-4 and GPT-3.5 are shown in Table 6 for the sensitive element detection step.

Table 6. Results of sensitive data detection by the two GPT models.

	GPT-4	GPT-3.5
Number of sensitive data elements	363	375
Incorrect detection	7	128
Duplicate detection	25	23
Insufficient detection	0	29

From the results, it is clear that although GPT-3.5 marked significantly more data as sensitive in the source codes tested, it did so incorrectly in many more cases compared to GPT-4. Incorrect detections count cases where the detection was incorrect. For instance, a function or the service itself might be mistakenly marked as sensitive data instead of a variable or object. Duplicate detection counts instances where the same sensitive data were marked multiple times within the same file. Insufficient detection counts situations where the sensitive data were correctly identified, but the justification for its sensitivity was either incomplete or incorrect. Next, the services that served as the source or target for the sensitive data and objects were identified. The results can be seen in Table 7, and although GPT-4 was clearly more efficient here, it made a significant number of errors in consistently detecting sensitive services. In the table, undetected sensitive services is the number of services that, although using sensitive data, were not detected; incorrect sensitivity levels is the number of services that were not assigned a correct sensitivity level even once; and missing information is the number of services that were detected for which some key information, such as the exact name of the service, was not detected but its presence was; while the insufficient detection count is the number of services that were successfully detected but not as many times as they occurred in the components of the application.

Table 7. Results of sensitive service identification by the two GPT models.

	GPT-4	GPT-3.5
Detected Sensitive Services	37	28
Undetected Sensitive Services	17	26
Incorrect Sensitivity Levels	2	17
Insufficient Detection Count	23	24
Missing Information	3	5

Here, detected services and undetected services refer to those services interacting with sensitive data that have been correctly or incorrectly detected by the model. Incorrect sensitivity levels denote cases where a service has a data member with an incorrect sensitivity level (this does not include cases where a service may have multiple data members with different sensitivity levels mixed together, as this level of lack of isolation was not anticipated in our original naive assumption). The insufficient detection count is similar; it counts cases where a service has been previously marked as having sensitive data, but for some reason,

when a new occurrence occurs, it no longer does. Finally, insufficient information indicates detections where some critical information, such as the data member or service name, has not been detected or has been detected incorrectly.

A prevalent error observed during our analysis was the violation of the SOLID principle [47], as defined by Robert C. Martin. This principle, interpreted in programming languages based on an object-oriented paradigm, emphasizes that instead of managing multiple similar operations, services should be responsible for handling a single, well-defined resource. For example, it was a common phenomenon in selected projects that all database operations, regardless of the tables or collections manipulated, were performed by a single service, which could then make a single ill-parameterized or insufficiently tested function call to attempt to retrieve data that a significant proportion of users would not have had access to on a relatively unprotected interface. Another part of our research team has demonstrated, through the development of a library [48], how a more precise, resource-based allocation can have additional productivity benefits. Such a lack of resource isolation is in fact a direct practical implementation of the CWE-653 vulnerability, as illustrated by the fact that the insufficient detection count for GPT-4 and GPT-3.5 is almost identical.

The three most common development practices that led to errors during the evaluation were the following.

- **Injection Circumvention**: Although Angular formally documents the exact role and intended use of services, a recurring problem in debugging has been that developers have circumvented the development pattern of injecting and then invoking methods. There were cases where services saved sensitive data in localStorage and the components of the application later read it from there, and one case where services used EventEmitters to pass sensitive values. This resulted in missing information cases in which the presence of services was detected but not their precise identities because they were not used in a conventional manner. Detecting and handling such cases is not easy. The prompt of the step to detect sensitive data and services will be extended in the later evaluations with an additional element to tag sensitive data from the EventEmitter, but because of the similarities (subscribe operation), they can easily be confused with observables, and thus, when detecting such cases, it is necessary to analyze the service code to see if the transmitted event is from inside or outside the application and if sensitive data are transmitted as a payload.
- **Monolith Service**: The most common issue was the lack of isolation mentioned above, a textbook example of the CWE-653 vulnerability, where services are not organized around a resource but around a type of operation. For example, there is an entity called DbService, which is responsible for the database operations of registration, login, shopping, commenting, and private messaging at the same time. This flaw led to very high insufficient detection count values, where the service was identified either as sensitive or not, depending on which of the many resources it was currently managing. In the future, this can be resolved so that at the end of the detection, each service is assigned the highest sensitivity value that was encountered during the detection.
- **Delegated Responsibility**: This issue refers to instances in which a service was initially used correctly, but then the data it handles was stored and transmitted in the application differently, such as by passing parameters between parent and child components or by creating a separate component or other object that acted as a session database. We expect that this will only be fully resolved if we can further develop our methodology as we plan and track accurate navigations and calls between application components. In the meantime, since the sensitivity detection prompt also tries to interpret what happens to sensitive data after it is first retrieved from the service, in the next round of analyses we will tag those that were derived from a sensitive service and, at the end of the detection, we will attempt to recover sensitivity levels based on these tags for data retrieved from local storages as well.

These error possibilities, while to some extent mitigatable, have also led to more severe issues later in the evaluation process.

4.2. RQ2

Regarding the detection of the components' protection level, the role of the GPT models at this stepwas primarily aggregation rather than deep interpretation. The results of the JSON mapping and protection level discovery steps for the total of 292 components examined are shown in Table 8.

Table 8. Results of sensitive service identification by the two GPT models.

	GPT-4	**GPT-3.5**
Components with Incorrect Service Detection	0	23
Components with Incorrect AuthGuard Detection	14	96
Components with Incorrect Protection Level	4	112

In this table, components with incorrect service detection are components for which the GPT model either did not detect a service, mistakenly detected a service, misidentified a service for a data member, or made a similar error. Similarly, components with incorrect AuthGuard detection is the number of components for which an error was made in accurately detecting the number and identity of AuthGuards. The third metric, components with incorrect protectivity level, is a direct consequence of this: the number of components for which manual validation revealed that the detected protectivity level was inaccurate.

In addition to the results presented in the table, the majority of incorrect AuthGuard detection issues for GPT-4 were caused by the nesting of parent and offspring modules within an application, which made it difficult to detect AuthGuards accurately. During the investigations, the maximum level of AuthGuard was detected in the majority of cases, resulting in only four cases of inadequate protection level out of 292 investigated cases. When evaluating the results of GPT-3.5, we had to confirm the findings of Cheshkov et al. [34] because, despite employing prompt engineering techniques and setting the temperature of the models to 0, we encountered difficult-to-explain problems. In the case of incorrect service detection, errors occurred where interfaces and elements not acting as services, such as elements of the Angular material library or the observable interface, were marked as services; during AuthGuard detection, despite a temperature of 0, hallucinations occurred and generated AuthGuards that did not exist in the application and either left their associated protection level at 0 or changed it to 1 or 2, causing 38% of the components to have incorrect protection levels.

Thus, although we used the GPT-3.5 models for completeness in the rest of the evaluation, it was guaranteed that the aggregate results of vulnerability detection would not be adequate for errors of this magnitude, and the errors it made would be difficult to correct even with prompt engineering.

4.3. RQ3

The vulnerability detection step was performed based on the aggregation of the two sub-results, and the outcomesof this step are shown in Table 9.

Table 9. Results of the vulnerability detection.

	GPT-4	GPT-3.5
Detected Vulnerability	40	17
False Vulnerability	0	8
Undetected Vulnerability	49	80
Undetected High-Sensitivity Vulnerability	10	16
Undetected Medium-Sensitivity Vulnerability	31	56
Undetected Low-Sensitivity Vulnerability	8	8

Detected vulnerability refers to correctly identified instances that align with the CWE-653 vulnerability. These are cases where a service handling sensitive data can be accessed from a component without adequate protection. This scenario poses a potential threat of data leakage or unauthorized use, either due to developer oversight or user malfeasance. In contrast, a false vulnerability is the number of cases that, although detected as vulnerabilities by the pipeline, are in fact not vulnerabilities because either the sensitivity level has been incorrectly scored high or the protection level has been incorrectly scored as low. The undetected vulnerability is the more severe problem: cases where the CWE-653 vulnerability was detected during manual validation but was not detected by the pipeline due to incorrect protection or sensitivity level.

As evident, our assumptions about GPT-3.5 were validated. The aggregated poor results from previous steps led to only a 1% successful detection rate for vulnerabilities. However, our initial naive assumption about the identification of sensitive services had very serious consequences for GPT-4 in the first round. Of the total number of errors that should have been identified, only less than 45% were successfully detected, and 20% of the errors (i.e., cases where a vulnerability should have been detected but was not) were in the high-sensitivity category, meaning that attempts to retrieve or manipulate highly-sensitive health, financial, or other personal data could theoretically be accessed fromthe front-end using an interface that was not supposedto handle such data. A significant part of the problem was identified in the RQ1 investigation as the monolith service, where services did not have one well-defined task and role but rather developers crammed several similar procedures into them without any consideration that all levels from non-sensitive to high-sensitivity data categories could appear within a single service. This problem caused the sensitivity level of the services to vary enormously in many cases, where a service was identified as having either medium or high sensitivity in a single component, while in another it was declared irrelevant to sensitive data.

To address these issues and improve the detection results, we significantly enhanced the vulnerability detection criteria for GPT-4. If a service was categorized as handling sensitive data in any component of the application, it was deemed sensitive across the entire application. Consequently, it was assigned the highest sensitivity level observed in any affected component. A spectacular improvement in the results can be seen in Table 10.

Table 10. Comparison of the original and more strict vulnerability detection rules.

	GPT-4 Original	GPT-4 Strict
Detected Vulnerability	40	79
False Vulnerability	0	0
Undetected Vulnerability	49	10
Undetected High-Sensitivity Vulnerability	10	0
Undetected Medium-Sensitivity Vulnerability	31	3
Undetected Low-Sensitivity Vulnerability	8	7

The detection success rate has jumped from 45% to 88.76%, and the high-sensitivity category misses have completely disappeared. In other words, with this tightening, which could have been performed in the sensitive element detection step as a post-processing step, we achieved an improvement of 44%. It is anticipated that the persisting problems will be resolved by further input data refinement and adjustments to service sensitivity prompts, taking into account the findings presented. Moreover, the expanded token limit will enable GPT-4 to analyze larger chunks of data within even more extensive and specific prompts.In the case of GPT-3.5, although some improvement may occur with similar tightening and further refinement of the prompts,anomalies such as hallucinated AuthGuards and high misinterpretation in the detection of sensitive data lead us to strongly question the usefulness of the model for such purposes and are in line with the observations of Cheshkov et al. [34], who obtained better results even with a dummy categorizer than with GPT-3.5. On the other hand,the results fromGPT-4 support the conclusion of Borji and Mohammadian [29] that GPT-4 is possibly the most promising LLM currently available for software development and source code interpretation tasks.

4.4. Future Work

Based on these results, we plan to continue our experiments in several different directions. In the future, we plan to either further validate our work, either with the C/C++ datasets [31,32] discussed in the survey literature or other frameworks—after all, the underlying concept of our methodology does not depend on the specific programming language and framework—or fine-tune it to explore further areas of research, such as navigation, as a result of which we might at some point even enable a new kind of dynamic, artificial-intelligence-based visual description language for code bases where this has so far proved impossible.

Another interesting area could be the testing of alternative sensitivity and protection scales. As the current scales have been highly subjective and focused on the usability of the methodology through logical reasoning and construction, we plan to test more domain-specific definitions in the future, for example, tailored to our own healthcare applications.

Since it has been confirmed that GPT is capable of detecting weaknesses and vulnerabilities that previously required human intervention due to contextual interpretation, it is now worthwhile to attempt to automatically detect many other vulnerabilities in the CWE database using large language models along similar lines.

As GPT APIs become more lenient with token limits, concerns about potential overruns diminish. It should be remembered that high token counts also increase costs. Also, if a research group might want to test our methodology on smaller LLMs that can be run locally and trained further, a significantly lower token limit should be expected. Thus, although the use of large prompts has been shown to be necessary to achieve the results presented, further research may include minimizing the size of the prompts.

5. Threats to Validity

Conclusion Validity: We attempted to select 12 of the largest and most complex available projects. It is likely thatthere may be others with structures and unique solutions that pose new challenges. However, our results demonstrate that large language models, especially GPT-4, can interpret code sufficiently by following the examples and thought processes defined in the prompts. Based on this, we hypothesize that these individual challenges can be addressed using the same methodology and by refining the rules in the prompts.

Internal Validity: We have not identified any threat to internal validity.

Construct Validity: As pointed out during evaluations, if the developers of the evaluated project significantly deviated from the Angular framework's development conventionsthe likelihood of GPT-4 misinterpreting or having trouble detecting contextual relationships may also increase. The 44% improvement achieved by refining the criteria during the vulnerability detection phase shows that such issues can be mitigated with augmentation and data cleansing after thefeature extraction. On the other hand, the problems identified in the RQ1 evaluation would most likely be considered code smells or vulnerabilities of a type that cannot be detected without a deeper understanding of the code base. Their mere presence has a negative impact on code quality, code interpretability, and code security, so their detection alone could be useful to development teams interested in static analysis and evaluation of their source code.

External Validity: In order to establish a comprehensive classification for data sensitivity and protection, we aimed to develop a broad taxonomy that focused on the potential harm resulting from data leakage. Our intention was to create a framework that could be easily understood and implemented by artificial intelligence. Through our investigation of vulnerabilities in the source code, we were able to uncover more intricate issues. However, it is important to acknowledge that there may be situations where the identification of a particular type of sensitive data necessitates the use of a distinct scale.Nevertheless, as we suggested in the conclusion validity threat assessment above, since our methodology with the GPT-4 API performed well overall in the evaluations, according to our hypothesis based on the results so far, these assumptions require mere modifications to the content of the prompts.

6. Conclusions

Detecting and resolving vulnerabilities is essential for preserving the availability, confidentiality, and integrity of digital systems. In an era where data intrusions can result in substantial financial losses, tarnished reputations, and legal repercussions, ensuring system security becomes crucial. Malicious actors can exploit unpatched vulnerabilities, resulting in unauthorized access, data theft, or even system shutdowns. Moreover, our growing dependence on digital infrastructure in industries like healthcare, finance, and transportation amplifies the real-world consequences of these vulnerabilities. A compromised system can endanger lives, destabilize economies, and diminish public confidence. Beyond tangible effects, addressing vulnerabilities reinforces an organization's commitment to its users, enhancing trust in its digital offerings. Detecting and resolving vulnerabilities is a fundamental aspect of responsible digital stewardship, not just a technical necessity.

In this paper, we have provided an overview of the primary directions in which artificial intelligence is increasingly being used in programming, formulated a categorization for determining both the nature of sensitive data and the application's vulnerability, and developed a process based on the GPT API to detect the CWE-653 vulnerability. Although GPT-3.5 underperformed during the evaluations, GPT-4 obtained an accuracy of 88.76% , confirming our hypotheses regarding the static code analysis and interpretation capabilities of large language models. We hope our research offers a deeper understanding of AI's potential in software engineering. Our presented methodology aims to facilitate the design and implementation of new code analysis techniques. These techniques can greatly aid in analyzing more complex source code, improving its quality and enhancing

vulnerability detection for development teams. Furthermore, we aspire to promote similar research directions for fellow research groups.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/fi15100326/s1.

Author Contributions: Conceptualization, Z.S. and V.B.; methodology, Z.S.; validation, Z.S.; writing—original draft preparation, Z.S.; writing—review and editing, Z.S. and V.B.; visualization, Z.S.; supervision, V.B. All authors have read and agreed to the published version of the manuscript.

Funding: The research was supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (RRF-2.3.1-21-2022-00004). Project No. TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

Data Availability Statement: No further data beyond what was published in the main text is publicly available.

Acknowledgments: We would like to thank Zoltán R. Jánki for developing the GitHub crawler and helping us with the selection and minification of the sample projects.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

Application programming interface	API
Attribute-based access control	ABAC
Automatic program repair	APR
Common weakness enumeration	CWE
Data access object	DAO
Generative pre-trained transformer	GPT
Large language model	LLM
Reinforcement learning from human feedback	RLHF
Role-based access control	RBAC

Appendix A. Prompts Used in the Analysis Pipeline

Appendix A.1. The Prompt for Detecting Sensitive Elements in the Component Code

Inspect the following Angular code, list all variables or attributes that could contain sensitive information for the user (usernames, passwords, email addresses, physical addresses, phone numbers, birthdates, social security numbers, credit card numbers, bank account numbers, routing numbers, driver's license numbers, passport numbers, health insurance numbers, medical records, IP addresses, biometric data, purchase history, tax IDs, company confidential information, employee IDs, salary information, private messages or chats, security questions and answers, location data, photographs, educational records, cookies, browser history, user behavior data, web server logs, API keys, encryption keys, cloud storage keys, OAuth tokens, cryptocurrency wallets, session IDs, device IDs, personal identification numbers (PINs), legal documents, employment history, marital status, family information, religious affiliation, sexual orientation, political views, racial or ethnic origin, trade union membership), their origin (meaning, how it is retrieved—user input, database, service or storage—in the latter case you must include the exact name of the service or storage), and function in a CSV format. In the function, try to include what happens with the attribute in the code, how it is stored, where it is forwarded Be concise! Avoid redundancy, for example, if an object contains sensitive attributes, you do not need to include them individually, just include the object and in its function mention that it contains email address, password, etc. The code is as follows, separated by three backticks: ing"'{text}'"

In your answer, use this CSV format:
Element; Type; Level; Origin; ServiceName; Function; Goal; GoalName; Classname
Where the Element is the name of the sensitive element, the Type is its type based on the list above i.e., username, password, email address, etc., the reason why this element is identified as sensitive. The Level is a numeric value from 1–3, signifying how sensitive this information is. A value of 1 means low severity, such as browser history, user behavior, cookies, or the identifiers of such information; 2 means medium severity such as username, email, password, birthdate, IP address, location data, political/religious views, sexual orientation, etc.; 3 means high severity, information that might lead to identity theft, financial loss, or significant privacy invasion such as social security number, health insurance, drivers license, legal documents, passport, medical information, credit card data, encryption keys, tax IDs. The Origin is one of the following values: ('UserInput', 'LocalStorage', 'Service') The ServiceName is the name of the service from where the element came if the value of the Origin is 'service', else its an empty string. The Function is a description of what the given element is used for. The Goal is where the element is forwarded, stored, what happens to it, it must be one of the following values: ('read', 'stored', 'service'). The GoalName is the name of the service the element is forwarded to if the value of the Goal field was 'service'. The Classname is the name of the Angular class where the element was detected. For example:
‘‘‘
customersArrayData; CustomerInformation;2; Service; DbServiceService;Array of Customer objects containing sensitive information about the customers; Read; ; CustomersComponent
id; User Identifier; 1; UserInput; Id to query a Customer object from the database;Used to call the GetCustomer function of the DbServiceService; Service, DbServiceService; CustomersComponent
’’’
Your answer must include only the CSV format and nothing else! If there are no sensitive elements in the file, your answer must be exactly 'There are no sensitive elements in the file!' The element and variable names in the files might not be English! Even if parts of the source code is in Spanish, German, Italian, Hungarian, or other languages, etc., you must correctly identify the sensitive elements!

In this prompt, an example value for the [text] field in the code might be the following minified code (from the app.component.ts of a provided Angular project):
import{Component}from'@angular/core';@Component({selector:'app-root', templateUrl: './app.component.html',styleUrls: ['./app.component.css']})exportclassAppComponent{}

Appendix A.2. The Definition of the Mapping JSON, to Be Used by Multiple Prompts

"type": type of the analyzed file, for example, component, service, guard, etc.
"selector": if the type of the analyzed file is a component, this field should contain its selector string from its class decorator. If the file is not a component, the value should be an empty string.
"name": name of the class in the analyzed file, for example AppComponent, LoginService, etc.
"file": the name of the analyzed file. If you received multiple files for a component—the .ts and the .html—this should be the name of the .ts file.
"path": the path of the component—if there is any—from the navigation modules. In this step, the value of this should remain an empty list.
"guarded": if the class is a component, and it is protected by an AuthGuard, the value should be 1, the default value is 0.
"guardlevel": the value of this should be 0.
"guards": the names of the guard classes that protect the class in the analyzed file. The default value is an empty list []. If the analyzed class is not an Angular component, it must

remain an empty list.
"navigations": A list of possible navigations from the file by class names. If you cannot find out the class name of the component targeted by the navigation, you should include its path. An optimal value of this field would be: ['LoginComponent', 'DashboardComponent']; an acceptable value would be ['/login', '/dashboard']. If there are no possible navigations from the analyzed class, the list should be empty.
You also need to take into account navigations that can possibly happen via a service, such as this.superService.navigate('home') or this.tereloService.menj('/profile')
If the analyzed file was an Angular module, the value should be an empty list ([])!
"injections": A dict, where the keys are the names of the services and other injectables that were injected into the analyzed class via the constructor. The values of the keys are a dict, containing and organizing the calls when said injectable was used in the file. The keys of this inner dict are asynchrons, functions, and variables. Asynchron refers to promises and observables used by the analyzed file, functions to the names of functions called by the analyzed file, variables to the various variables of the injected accessed directly by the analyzed file. The values of this inner dict are lists which are either empty, if that type of interaction with the injectable did not happen, or the names of said interaction. An example value for this field would be: {'LoginService': 'Functions': ['login', 'forgotPassword'], 'Variables': [], 'Asynchrons: ['currentUser']', 'LoggerService: 'Functions': ['logInfo', 'logError'], 'Variables': ['log'], 'Asynchrons': []'} If there are no injectables in the file, the value should be an empty dict!
"parents": If the analyzed file is a component and it has parent components, this should be a list that contains the names of those components, like ['ProfileComponent'] or ['LoginComponent', 'RegisterComponent']. The default value is an empty list.
"children": If the type of the analyzed file is a component, this list should contain the names—or if the names cannot be determined from the context, then the selector strings of every child component that are contained by the .html template of the component. For example: ['SpinnerComponent', 'alert-component']. If the type of the file is not component or it does not have any children, then the value of this field should be an empty list. The router outlet should also be considered to be a child, so if the .html template contains it, it should be included here!

Appendix A.3. The Mapping Prompt That Converts the Minified TypeScript Files into JSONs

You will receive a minified version of a TypeScript file from an Angular web application, separated by three backticks, describing either a component or a service of an Angular project. Your goal is to create a JSON from said file, which contains the services that were injected into it via the constructor, the names of the functions called from said services, the possible navigation operations, and in case of components, the names of the children components in a list. If any of the mentioned elements are missing, its place in the JSON should contain the key, but only an empty list as its value. The JSON should follow this format:
{firstResultFormat}
A short example of a correct output:
{firstResultFormatExample}
The file to be analyzed is the following:
‘‘‘
{section}
’’’
Your answer should only contain the JSON and nothing else, as your answer will be instantly parsed as a JSON by the receiver! Use double quotation marks (") and no single quotation mark (') under any circumstance!

In this prompt, the value of firstResultFormat is the content of Appendix A.2, the value of firstResultFormatExample was

{ “type”: “Component”,
“selector”: “profile-comp”,
“name”: “ProfileComponent”,
“file”: “profile.component.ts”,
“path”: [“/profile”],
“guarded”: 0,
“guards”: [],
“guardlevel”: 0,
“navigations”: [‘/dashboard’],
“injections”: {‘OAuthService’: {‘Functions’: [‘checkAuth’], ‘Variables’: [],
’Asynchrons’: [‘token’]}, ‘LoggerService’: {‘Function’: [‘logInfo’, ‘logError’],
’Variable’: [‘log’], ‘Asynchron’: []}},
“parents”: [],
“children”: [] }
during the evaluations.
The value of section is a minified .ts source file such as:
import{Component,OnInit}from‘@angular/core’;@Component({selector:’app-
about’,templateUrl:’./about.component.html’,styleUrls:[‘./about.component.scss’]})
exportclassAboutComponentimplementsOnInit{constructor(){}ngOnInit(){}}

Appendix A.4. The Access Control Evaluation Prompt

You will receive a JSON object, which is describing a component from an Angular project and their relations to each other. The JSON object will follow this format: {firstResultFormat} With this JSON, you will also receive minified versions of the Angular files, that will contain information concerning the routing and the AuthGuard configurations of the web application. Based on these files you will need to extend the JSON representing the component with the following steps.
1. Modify the path value if their ‘path’ is an empty list. If there is no path defined in any of the route[] arrays in the received files that contain a path value for that particular component, then the value of its path should remain an empty list in the JSON. There is a good chance for example, that components that are children to another component should not have any paths, because they are most likely to be accessible through their parent component. For example: If there is {{ path: ‘register’, component: RegisterComponent }} in one of the routing modules Then, in the JSON, where type: ‘component’ and name: ‘RegisterComponent’, ‘path’ should be [‘/register’] Make sure that even if there is already a value in the path field, it has a unified format, such as ‘/register’, ‘/home’, ‘/profile/:id’ The reason why the path is a list and not a string is because you also need to include the default/home path to the appropriate JSON (‘/’) and the wildcard path (‘**’) to the appropriate JSON if they exist in the route definitions. The paths might be also stored in multiple files if there is a module–submodule relationship. For example, if one routing module contains the following:
{{path: ‘users/:id’,loadChildren: () => import(“./users-module/users.module”).then(m => m.UsersModule)}}, then the UsersModule imports another routing module, for example, a UsersRoutingModule, where there is a line such as:
{{path: ‘addition’, component: AddUserComponent}},
then the path for the AddUserComponent is ‘/users/:id/adddition’ if there is
{{path: ‘’, redirectTo: ‘profile’}}
in that module, then the component with the ‘/users/:id/profile’ path should also get the ‘/users/:id/’ route in their paths array. Do not under any circumstance mix the paths and the possible navigations from that given component! - only components have paths! - most components only have one path. If the component is used as the child component of another, it is very likely that it does not have a path in the routing information, therefore, its path list should remain empty. - the only exceptions are usually the default routes: ‘’, ‘/’ and the wildcard route: ‘**’

2. List every ‘guard’ by their class names in the ‘guards’ list of the JSON that are protecting the component based on the canActivate parts of the routing information. For example: if there is a configuration like this in one of the module files {{path: ‘profile’, component: ProfileComponent, canActivate: [AuthguardGuard]}} then in the JSON, where ‘type’: ‘component’ and ‘name’: ‘ProfileComponent’, ‘guarded’ should be set to 1, and the ‘guards’ should be: [‘AuthguardGuard’] Again, take into account the module level imports: if there is a {{path: ‘users/:id’, canActivate: [RoleGuard], loadChildren: () => import(“./users-module/users.module”).then(m => m.UsersModule)}} then every route/component that is imported by the UsersModule already has the RoleGuard, in addition to any other they might have individually! For example, a {{path: ‘addition’, component: AddUserComponent}} in the UsersModule’s imported routes would give the AddUserComponent’s JSON the [‘RoleGuard’] as its guard, while a {{path: ‘deletion’, component: DeletionUserComponent, canActivate: [AdminGuard]}} would give the DeletionUserComponent’s JSON the [‘RoleGuard’, ‘AdminGuard’] list as its ‘guards’.
3. As a final step, you need to calculate the guardlevel if the JSON has ‘component’ as its type, based on the following rules:
- if the ‘guarded’ value is 0 and their ‘guards’ list is still empty after the previous steps, the ‘guardlevel’ is 0 - if it has at least one guard in their ‘guards’ list, but based on the implementation of it, that guard only checks whether the user is logged in in or not, the ‘guardlevel’ should be 1. - if there is implementation of guard or guards in the ‘guards’ list, they not only check whether they are logged in, but they also check for access level—admin or normal user; patient, relative, nurse or practitioner; buyer or seller—the ‘guardlevel’ should be 2. - if the implementation of the guards contains everything necessary for guardlevel 2, but go beyond them and check for extra information, for example, location, IP address, timezone, device/browser type, the guardlevel should be 3.
Your answer should contain the full version of the modified JSON, it will have to include every object which was in the input in the same format and contain every modification you have made to it.
This is the input JSON file, separated by three backticks:
‘‘‘
{firstResult}
’’’
And these are the minified files containing the routing and guard information, also separated by three backticks:
‘‘‘
{filteredGuards}
’’’

Here, the value of the firstResult is a mapped JSON, following the format described in Appendix A.2, such as the value of firstResultFormatExample from Appendix A.3. The value of filteredGuards is the minified, concatenated code of the AuthGuards and routing modules from a project such as:
import{NgModule}from‘@angular/core’;import{PreloadAllModules,RouterModule, Routes}from‘@angular/router’;import{AuthGuard}from‘./auth-module/auth.guard’ ;import{MainMenuComponent}from‘./components/main-menu/main-menu.component’; import{NotFoundComponent}from‘./components/not-found/not-found.component’; import{ReclutamientoComponent}from‘./components/reclutamiento/reclutamiento .component’;import{RoleGuard}from‘./guards/role.guard.ts’;const routes:Routes= {path:’’,redirectTo:‘inicio’,pathMatch:‘full’},{path:‘inicio’,component:MainMenuCompo nent},{path:‘reclutamiento’,component:ReclutamientoComponent,canActivate:roleGuard} ,{path:‘404’,component:NotFoundComponent};@NgModule({imports:RouterModule.for Root(routes,{preloadingStrategy:PreloadAllModules,scrollPositionRestoration:‘enabled’}) exports:RouterModule})export class AppRoutingModule{}import{Injectable}from‘@angular /core’;import{CanActivate,ActivatedRouteSnapshot,RouterStateSnapshot,UrlTree,Router}

from‘@angular/router’;import{Observable}from‘rxjs’;@Injectable({providedIn:‘root’})export class PlanningGuard implements CanActivate{constructor(private router:Router){}canActivate(next:ActivatedRouteSnapshot,state:RouterStateSnapshot):Observable<boolean UrlTree> Promise<boolean UrlTree>,boolean UrlTree{if(this.router.getCurrentNavigation().extras.state)return true;this.router.navigate(‘planning’);return false;}}

References

1. Introduction to the Angular Docs. Available online: https://angular.io/docs (accessed on 20 July 2023).
2. Sanderson, K. GPT-4 is here: What scientists think. *Nature* **2023**, *615*, 773–773. [CrossRef]
3. Deng, J.; Lin, Y. The Benefits and Challenges of ChatGPT: An Overview. *Front. Comput. Intell. Syst.* **2023**, *2*, 81–83. [CrossRef]
4. Jánki, Z.R.; Bilicki, V. Rule-Based Architectural Design Pattern Recognition with GPT Models. *Electronics* **2023**, *12*, 3364. [CrossRef]
5. Hourani, H.; Hammad, A.; Lafi, M. The Impact of Artificial Intelligence on Software Testing. In Proceedings of the 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT), Amman, Jordan, 9–11 April 2019. [CrossRef]
6. Heydon, A.; Maimone, M.; Tygar, J.; Wing, J.; Zaremski, A. Miro: Visual specification of security. *IEEE Trans. Softw. Eng.* **1990**, *16*, 1185–1197. [CrossRef]
7. Giordano, M.; Polese, G. Visual Computer-Managed Security: A Framework for Developing Access Control in Enterprise Applications. *IEEE Softw.* **2013**, *30*, 62–69. [CrossRef]
8. Hossain Misu, M.R.; Sakib, K. FANTASIA: A Tool for Automatically Identifying Inconsistency in AngularJS MVC Applications. In Proceedings of the Twelfth International Conference on Software Engineering Advances, Athens, Greece, 8–12 October 2017.
9. Szabó, Z.; Bilicki, V. Access Control of EHR Records in a Heterogeneous Cloud Infrastructure. *Acta Cybern.* **2021**, *25*, 485–516. [CrossRef]
10. Martin, B.; Brown, M.; Paller, A.; Kirby, D.; Christey, S. CWE. *SANS Top* **2011**, *25*
11. Rainey, S.; McGillivray, K.; Akintoye, S.; Fothergill, T.; Bublitz, C.; Stahl, B. Is the European Data Protection Regulation sufficient to deal with emerging data concerns relating to neurotechnology? *J. Law Biosci.* **2020**, *7*, lsaa051. [CrossRef]
12. Cheng, S.; Zhang, J.; Dong, Y. How to Understand Data Sensitivity? A Systematic Review by Comparing Four Domains. In Proceedings of the 2022 4th International Conference on Big Data Engineering, Beijing, China, 26–28 May 2022. [CrossRef]
13. Belen Saglam, R.; Nurse, J.R.; Hodges, D. Personal information: Perceptions, types and evolution. *J. Inf. Secur. Appl.* **2022**, *66*, 103163. [CrossRef]
14. Lang, C.; Woo, C.; Sinclair, J. Quantifying data sensitivity. In Proceedings of the Tenth International Conference on Learning Analytics & Knowledge, Frankfurt, Germany, 23–27 March 2020. [CrossRef]
15. Chua, H.N.; Ooi, J.S.; Herbland, A. The effects of different personal data categories on information privacy concern and disclosure. *Comput. Secur.* **2021**, *110*, 102453. [CrossRef]
16. Rumbold, J.M.; Pierscionek, B.K. What Are Data? A Categorization of the Data Sensitivity Spectrum. *Big Data Res.* **2018**, *12*, 49–59. [CrossRef]
17. Botti-Cebriá, V.; del Val, E.; García-Fornes, A. Automatic Detection of Sensitive Information in Educative Social Networks. In Proceedings of the 13th International Conference on Computational Intelligence in Security for Information Systems (CISIS 2020), Burgos, Spain, 14 May 2020; pp. 184–194. [CrossRef]
18. Jiang, L.; Liu, H.; Jiang, H. Machine Learning Based Recommendation of Method Names: How Far are We. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019. [CrossRef]
19. Momeni, P.; Wang, Y.; Samavi, R. Machine Learning Model for Smart Contracts Security Analysis. In Proceedings of the 2019 17th International Conference on Privacy, Security and Trust (PST), Fredericton, NB, Canada, 26–28 August 2019. [CrossRef]
20. Mhawish, M.Y.; Gupta, M. Predicting Code Smells and Analysis of Predictions: Using Machine Learning Techniques and Software Metrics. *J. Comput. Sci. Technol.* **2020**, *35*, 1428–1445. [CrossRef]
21. Cui, J.; Wang, L.; Zhao, X.; Zhang, H. Towards predictive analysis of android vulnerability using statistical codes and machine learning for IoT applications. *Comput. Commun.* **2020**, *155*, 125–131. [CrossRef]
22. Park, S.; Choi, J.Y. Malware Detection in Self-Driving Vehicles Using Machine Learning Algorithms. *J. Adv. Transp.* **2020**, *2020*, 3035741 . [CrossRef]
23. Jiang, N.; Lutellier, T.; Tan, L. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spain, 22–30 May 2021; IEEE: Piscataway, NJ, USA, 2021. [CrossRef]
24. Sharma, T.; Kechagia, M.; Georgiou, S.; Tiwari, R.; Vats, I.; Moazen, H.; Sarro, F. A Survey on Machine Learning Techniques for Source Code Analysis. *arXiv* **2022**, arXiv:2110.09610.
25. Sarkar, A.; Gordon, A.D.; Negreanu, C.; Poelitz, C.; Ragavan, S.S.; Zorn, B. What is it like to program with artificial intelligence? *arXiv* **2022**, arXiv:2208.06213.
26. Wei, J.; Tay, Y.; Bommasani, R.; Raffel, C.; Zoph, B.; Borgeaud, S.; Yogatama, D.; Bosma, M.; Zhou, D.; Metzler, D.; et al. Emergent Abilities of Large Language Models. *arXiv* **2022**, arXiv:2206.07682.

27. Liu, Y.; Han, T.; Ma, S.; Zhang, J.; Yang, Y.; Tian, J.; He, H.; Li, A.; He, M.; Liu, Z.; et al. Summary of ChatGPT/GPT-4 Research and Perspective Towards the Future of Large Language Models. *arXiv* **2023**, arXiv:2304.01852.
28. Surameery, N.M.S.; Shakor, M.Y. Use Chat GPT to Solve Programming Bugs. *Int. J. Inf. Technol. Comput. Eng.* **2023**, *3*, 17–22. [CrossRef]
29. Borji, A.; Mohammadian, M. Battle of the Wordsmiths: Comparing ChatGPT, GPT-4, Claude, and Bard. *SSRN Electron. J.* **2023** . [CrossRef]
30. Wu, J. Literature review on vulnerability detection using NLP technology. *arXiv* **2021**, arXiv:2104.11230.
31. Thapa, C.; Jang, S.I.; Ahmed, M.E.; Camtepe, S.; Pieprzyk, J.; Nepal, S. Transformer-based language models for software vulnerability detection. In Proceedings of the 38th Annual Computer Security Applications Conference, Austin, TX, USA, 5–9 December 2022; pp. 481–496. [CrossRef]
32. Omar, M. Detecting software vulnerabilities using Language Models. *arXiv* **2023**, arXiv:2302.11773.
33. Sun, Y.; Wu, D.; Xue, Y.; Liu, H.; Wang, H.; Xu, Z.; Xie, X.; Liu, Y. When GPT Meets Program Analysis: Towards Intelligent Detection of Smart Contract Logic Vulnerabilities in GPTScan. *arXiv* **2023**, arXiv:2308.03314.
34. Cheshkov, A.; Zadorozhny, P.; Levichev, R. Evaluation of ChatGPT Model for Vulnerability Detection. *arXiv* **2023**, arXiv:2304.07232.
35. Feng, S.; Chen, C. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. *arXiv* **2023**, arXiv:2306.01987.
36. Ferraiolo, D.; Cugini, J.; Kuhn, D.R. Role-based access control (RBAC): Features and motivations. In Proceedings of the 11th Annual Computer Security Application Conference, New Orleans, LA, USA, 11–15 December 1995; pp. 241–248.
37. Yuan, E.; Tong, J. Attributed based access control (ABAC) for Web services. In Proceedings of the IEEE International Conference on Web Services (ICWS'05), Orlando, FL, USA, 11–15 July 2005. [CrossRef]
38. Pricing of GPT. Available online: https://openai.com/pricing (accessed on 20 July 2023).
39. OpenAI—Privacy Policy. 2023. Available online: https://openai.com/policies/privacy-policy (accessed on 25 September 2023).
40. Qiu, R. Editorial: GPT revolutionizing AI applications: Empowering future digital transformation. *Digit. Transform. Soc.* **2023**, *2*, 101–103. [CrossRef]
41. Shoeybi, M.; Patwary, M.; Puri, R.; LeGresley, P.; Casper, J.; Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv* **2019**, arXiv:1909.08053.
42. Ji, Z.; Lee, N.; Frieske, R.; Yu, T.; Su, D.; Xu, Y.; Ishii, E.; Bang, Y.J.; Madotto, A.; Fung, P. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* **2023**, *55*, 248. [CrossRef]
43. Moghaddam, S.R.; Honey, C.J. Boosting Theory-of-Mind Performance in Large Language Models via Prompting. *arXiv* **2023**, arXiv:2304.11490.
44. Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1877–1901. [CrossRef]
45. Wang, X.; Wei, J.; Schuurmans, D.; Le, Q.; Chi, E.; Narang, S.; Chowdhery, A.; Zhou, D. Self-Consistency Improves Chain of Thought Reasoning in Language Models. *arXiv* **2023**, arXiv:2203.11171.
46. What Is the Difference between the GPT-4 Models? Available online: https://help.openai.com/en/articles/7127966-what-is-the-difference-between-the-gpt-4-models (accessed on 20 July 2023).
47. Martin, R.C. Getting a SOLID Start. Robert C Martin-objectmentor.com. 2013. Available online: https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start (accessed on 26 September 2023)
48. Kokrehel, G.; Bilicki, V. The impact of the software architecture on the developer productivity. *Pollack Period.* **2022**, *17*, 7–11. [CrossRef]