

# Comparing Effectiveness and Efficiency of Large Language Models and Traditional Vulnerability Assessment Tools in IT Network Vulnerability Analysis

## Master's Thesis

in partial fulfillment of the requirements for  
the degree of Master of Science (M.Sc.)  
in Informatik

submitted by  
Oliver Dzaeck

First examiner: Prof. Dr. Matthias Thimm  
Artificial Intelligence Group

Advisor: Prof. Dr. Matthias Thimm  
Artificial Intelligence Group




## Statement

Ich erkläre, dass ich die Masterarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

	Yes	No
I agree to have this thesis published in the library.	<input checked="" type="checkbox"/>	<input type="checkbox"/>
I agree to have this thesis published on the webpage of the artificial intelligence group.	<input checked="" type="checkbox"/>	<input type="checkbox"/>
The thesis text is available under a Creative Commons License (CC BY-SA 4.0).	<input checked="" type="checkbox"/>	<input type="checkbox"/>
The source code is available under a GNU General Public License (GPLv3).	<input checked="" type="checkbox"/>	<input type="checkbox"/>
The collected data is available under a Creative Commons License (CC BY-SA 4.0).	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Löwen, 29.06.24  
(Place, Date)

  
(Signature)



## **Zusammenfassung**

Diese Masterarbeit untersucht die Effizienz und Effektivität von großen Sprachmodellen im Bereich der Schwachstellenanalyse von IT Netzwerken. Ziel ist es dabei festzustellen, inwieweit große Sprachmodelle für den Bereich der Cybersecurity angepasst werden können und unter welchen Bedingungen sie eine bessere Leistung erzielen. In dieser Arbeit wird das Pattern der "Retrieval Augmented Generation" untersucht. Ziel ist es zu bewerten in wie weit große Sprachmodelle als nützliche Alternative für die Identifizierung und Klassifizierung von Schwachstellen in Unternehmensnetzwerken dienen können.

Herkömmliche Tools wie Nessus stützen sich auf signaturbasierte Methoden und heuristische Ansätze. Im Gegensatz dazu bieten Sprachmodelle aufgrund ihrer Fähigkeit, natürliche Sprache zu verarbeiten und zu analysieren, potenzielle Vorteile hinsichtlich Flexibilität und Genauigkeit. Insbesondere bei der Analyse unstrukturierter Daten aus Netzwerk-Scans haben große Sprachmodelle gezeigt, dass sie kontextbezogene Informationen besser erfassen und relevante Schwachstellen genauer identifizieren können. Dies deutet darauf hin, dass große Sprachmodelle eine wertvolle Ergänzung zu bestehenden Werkzeugen zur Schwachstellenbewertung sein könnten.

## **Abstract**

This thesis investigates the efficiency and effectiveness of large language models in the area of vulnerability analysis of IT networks. The goal is to determine to what extent large language models can be adapted for the area of cybersecurity and under what conditions they perform better. In this thesis, the pattern of "retrieval augmented generation" is investigated. The goal is to evaluate to what extent large language models can serve as a useful alternative for identifying and classifying vulnerabilities in corporate networks.

Traditional tools such as Nessus rely on signature-based methods and heuristic approaches. In contrast, language models offer potential advantages in terms of flexibility and accuracy due to their ability to process and analyze natural language. In particular, when analyzing unstructured data from network scans, large language models have shown that they can better capture contextual information and more accurately identify relevant vulnerabilities. This suggests that large language models could be a valuable addition to existing vulnerability assessment tools.



# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Research objectives . . . . .	6
1.2	Method . . . . .	6
<b>2</b>	<b>Theoretical Framework</b>	<b>9</b>
2.1	Large Language Models . . . . .	9
2.1.1	Retrieval-Augmented Generation . . . . .	18
2.2	Cybersecurity . . . . .	23
2.3	State of research . . . . .	26
2.4	Vulnerability Assessment Tools in IT Security . . . . .	28
2.5	CVE . . . . .	28
2.6	CVSS . . . . .	32
2.7	Exploit DB . . . . .	37
2.8	Nmap . . . . .	39
<b>3</b>	<b>Method</b>	<b>41</b>
3.1	Datasources . . . . .	41
3.2	Models and technologies . . . . .	41
3.2.1	Models . . . . .	41
3.2.2	RAG . . . . .	43
3.3	Experimental Design . . . . .	49
3.4	Experiment 1 . . . . .	50
3.4.1	Experimental Objectives and Hypotheses . . . . .	50
3.4.2	Methodology . . . . .	50
3.4.3	Experiment Execution . . . . .	56
3.4.4	Measurement and Evaluation . . . . .	59
3.4.5	Results Analysis . . . . .	63
3.5	Experiment 2 . . . . .	71
3.5.1	Experimental Objectives and Hypotheses . . . . .	71
3.5.2	Methodology . . . . .	72
3.5.3	Experiment Execution . . . . .	76
3.5.4	Measurement and Evaluation . . . . .	78
3.5.5	Results Analysis . . . . .	80
3.6	Experiment 3 . . . . .	84
3.6.1	Experimental Objectives and Hypotheses . . . . .	84
3.6.2	Methodology . . . . .	84
3.6.3	Experiment Execution . . . . .	94
3.6.4	Measurement and Evaluation . . . . .	99
3.6.5	Results Analysis . . . . .	101

<b>4</b>	<b>Discussion</b>	<b>110</b>
4.1	Interpretation of the results . . . . .	111
4.2	Practical implications and applications . . . . .	116



## List of Abbreviations

<b>AI</b>	Artificial Intelligence
<b>AIG</b>	Artificial Intelligence Group
<b>AWS</b>	Amazon Web Services
<b>CSV</b>	Comma-Separated Values
<b>CVE</b>	Common Vulnerabilities and Exposures
<b>CVSS</b>	Common Vulnerability Scoring System
<b>DMZ</b>	Demilitarized Zone
<b>FTP</b>	File Transfer Protocol
<b>GPT</b>	Generative Pre-trained Transformer
<b>HTML</b>	HyperText Markup Language
<b>HTTP</b>	Hypertext Transfer Protocol
<b>JSON</b>	JavaScript Object Notation
<b>LLM</b>	Large Language Model
<b>MTA</b>	Mail Transfer Agent
<b>NLP</b>	Natural Language Processing
<b>Nmap</b>	Network Mapper
<b>NVD</b>	National Vulnerability Database
<b>OS</b>	Operating System
<b>RAG</b>	Retrieval-Augmented Generation
<b>REST</b>	Representational State Transfer
<b>SQL</b>	Structured Query Language
<b>SSH</b>	Secure Shell
<b>URL</b>	Uniform Resource Locator

**VM** Virtual Machine

**XML** Extensible Markup Language

# 1 Introduction

In 2022, the population's use of Large Language Models (LLMs) has measurably increased. The most well-known example of an LLM is the Chatbot Generative Pretrained Transformer or ChatGPT. Even though, according to the Luenendonk List, the most critical topic in 2022 was still cybersecurity <sup>1</sup>, current Google Trends data (as of July 2023) shows that interest in LLMs has significantly increased since the beginning of 2023, significantly surpassing the previous topic of cybersecurity in terms of search query volume.

The reason for the significant increase is the versatility that ChatGPT provides. Through interaction in natural language and extensive training with data, ChatGPT can conduct high-quality conversations in various existing domains like education, healthcare, reasoning, text generation, human-machine interaction, and scientific research [27]. This is because of the underlying architecture and the training used to develop ChatGPT (see 2.1).

This new technology opens up new possibilities for various industries to integrate artificial intelligence into their daily operations. A strength of ChatGPT is that it can query data creatively, interact with data, and suggest analyses, predictions, optimizations, and improvements based on specific data sets. A paper by Brown et al. [7] describes how GPT-3 can be used in various use cases and requires only tiny datasets for some tasks. Although this master's thesis deals with technical topics, the other domains served by ChatGPT should not be neglected. The use of LLMs extends through all known industries.

To maximize the benefits of AI technology and manage the associated responsibilities, companies are establishing specialized task forces to ensure that employees can efficiently work with AI-supported tools to create new operational excellence. Language models like ChatGPT surpass mere text generation, offering support in broader areas. Many companies are setting up ChatGPT instances to facilitate access to large language models while ensuring that no company internal information is shared with official models. This technology integrates into developers daily work by generating language and code from programming languages using the appropriate datasets. Such developments are on the rise, and it is foreseeable that most IDEs will include AI support, similar to GitHub Copilot in Visual Studio Code.

Moreover, companies are not just forming task forces but entire departments dedicated to exploring how AI can be deployed to streamline internal processes. Internal ChatGPT instances, enriched with company-specific knowledge, enable various use cases, from creative error code analysis to making company knowledge available that was previously scattered across various PDF documents. Microsoft has even

---

<sup>1</sup><https://www.luenendonk.de/aktuelles/themen-trends/die-top-5-der-it-investitionen-2023/>

launched a product named Microsoft 365 Copilot, integrated throughout almost the entire Microsoft product lineup. This product enhances these offerings with the capabilities of LLMs to boost user productivity. It incorporates the application of LLMs into tools such as Word, Excel, PowerPoint, Outlook, and Teams[28]. Within Microsoft's Azure Cloud, Copilot aggregates knowledge and data from hundreds of services to elevate productivity, decrease costs, and derive extensive insights. Microsoft Copilot for Azure is designed to field questions about Azure and provide information tailored to individual Azure resources and environments[35]. Moreover, Azure Copilot can be employed to evaluate the security of Azure resources and condense extensive data signals into crucial insights, thereby helping to cut through clutter, preempt cyber threats, and bolster security posture[31]. The impact on various industries is tremendous by tailoring LLMs to specific needs, integrating them with knowledge databases, and even providing internet access.

Beyond GPT, one of the most renowned language models, numerous providers have developed additional models, including Google with Gemini and Meta with LLaMA. According to Gartner, by 2026, it is expected that 100 million people will have AI-assisted "colleagues" to support their daily work. By 2027, the productivity value of AI will be recognized as a primary economic indicator of national power, and by 2028, due to labor shortages in manufacturing, retail, and logistics, there will be more intelligent robots than employees[18].

The cybersecurity sector is expected to experience significant changes as malicious actors leverage generative AI systems for cyber and fraud attacks. According to Europol, expert analyses[10] have determined that models like GPT-3.5 and those above it offer a wide range of criminal possibilities. Such models could greatly simplify information gathering in areas such as burglary, terrorism, cybercrime, and others without prior knowledge. Furthermore, LLMs are used for fraud, identity theft, and social engineering, as they can generate highly authentic texts. This makes detecting phishing attempts more challenging.

Europol also notes that such models may simplify entry into cybercriminal activities. Nonetheless, Europol advises evaluating the possibilities of LLMs for one's areas and acquiring the knowledge and competencies to, for example, use LLMs to combat cybercrime[10].

In light of the ongoing digitalization, IT security is an essential component of today's world and will undoubtedly become even more critical. Especially in turbulent times like the present, almost every day, we read about hacker attacks on companies and institutions<sup>2</sup>. On average, every second company fell victim to hacker attacks in 2022. The risks posed by unprotected systems are diverse, and the consequences can devastate companies and individuals. In this context, penetration tests, which are processes for identifying security gaps in IT systems, have taken on

---

<sup>2</sup><https://www.luenendonk.de/aktuelles/themen-trends/die-top-5-der-it-investitionen-2023/>

a crucial role.

The analysis of vulnerabilities in IT enterprise networks has been primarily based on heuristic techniques and dynamic and static analyses, which have been established and proven over the years. By supporting LLMs, potential gaps in accuracy could be closed.

Use cases for integration span many different areas of cybersecurity. LLMs could automate reporting and documentation to generate understandable reports about the network with custom classifications and risk assessments. LLMs could also help with pattern recognition of anomalies within the network scan and develop suggestions for improvement. When analyzing vulnerabilities within corporate networks, areas that would otherwise only be covered by current tools with the integration of machine learning or AI solutions could be covered. For example, LLMs could interpret CVE entries better than heuristic and dynamic analysis could by detecting vulnerable versions not explicitly defined in a CVE.

Involvement in IT security and LLMs motivates a deeper analysis of how LLMs can enhance the efficiency and effectiveness of vulnerability analyses or augment existing tools for more detailed work. As digital attacks continue to rise, particularly with countries like China and Russia in mind, integrating cutting-edge technologies with established security strategies forms a crucial bridge. This work aims to contribute valuable insights to the research topic and expand the collective understanding of it.

## 1.1 Research objectives

The goal of this master's thesis is to validate the hypothesis that LLMs can make measurable contributions to the analysis of vulnerabilities in networks. This thesis intends to verify, through systematic research and experiments, that LLMs are capable of achieving superior results in vulnerability detection compared to conventional tools based on methods like static heuristics. A central research focus is on evaluating different LLMs in terms of their efficiency and effectiveness by training them with specific datasets and comparing their performance in practical applications. The emphasis is on the detection and classification of network scans conducted across networks. Typically, the results include the detected hosts and the services running on them. In addition to comparing with conventional tools, this study aims to determine how an LLM can be best utilized for this specific domain.

This thesis seeks to answer the following research questions:

- **RQ1.** How does the efficiency of an LLM in detecting vulnerabilities in IT networks compare to established vulnerability assessment tools?
- **RQ2.** How do various LLMs compare in terms of efficiency for detecting vulnerabilities in IT networks, as measured by key performance metrics such as Precision, Recall, F1-Score, and False Positive/Negative Rates?

The following additional questions should be answered in order to gain a deeper understanding of the research topic. These questions are included, but not limited to:

- What are the limitations of using LLMs for this purpose compared to traditional vulnerability assessment tools?
- Can the use of LLMs in conjunction with traditional tools yield better results than using either individually?
- What kind of customization is required to improve the efficiency of an LLM in vulnerability detection?
- Can LLMs assess the severity of a vulnerability?

## 1.2 Method

By formulating research questions in this master's thesis to compare the efficiency and effectiveness of LLMs, the chosen quantitative methodology is the core of the

research. The aim is to collect empirical data in order to comprehensively evaluate the research questions. By conducting various experiments (see chapter 3.4.1, 3.5.1 and 3.6.1), the LLM to be used for subsequent investigations will be selected. Here, metrics of a binary classification model provide important insights into the performance of the LLMs.

The first experiment evaluates the effectiveness and efficiency of different LLMs in detecting known vulnerabilities in software, as cataloged in MITRE's CVE database. This involves comparing various models, including standard pre-trained LLMs and those with Retrieval Augmented Generation (RAG) techniques. To understand their capability in correctly identifying vulnerabilities it is particularly focusing on the use of natural language processing to understand the context and semantics of unstructured vulnerability descriptions. The experiments also investigate the models generalization capabilities and the contextual understanding also assessing how well they can recognize correlations in software versions. Also detecting vulnerabilities even when explicit version details are missing. The results of this experiment will determine which model is most effective for this specific case, guiding further research into optimizing LLMs for vulnerability detection in software.

The second experiment evaluates the effectiveness of the GPT-4 model with the Azure AI Search and the GPT-3 model with the RAG approach in classifying vulnerabilities from CVE descriptions. This study builds on their successful performance in prior tests and includes a cost-effectiveness analysis. The hypothesis posits that LLMs can accurately classify vulnerabilities vector scores using a RAG. Also for entries lacking CVSS ratings and potentially outperforming traditional assessment tools. This experiment draws on previous findings by Aghaei et al. [1], using their CVEDrill model as a benchmark to test whether a well-informed LLM can lead to superior performance in vulnerability classification. This contributes to the broader goal of enhancing network security assessment.

The third experiment is focuses on determining whether the best-performing LLM from previous experiments. The experiment test the LLMs to analyze and interpret network scans to identify and assess the severity of vulnerabilities. A simulated IT network environment will be set up specifically for this experiment to facilitate this. Previous experiments explored the models abilities to detect vulnerabilities in software and to evaluate their severity based solely on textual descriptions. This final experiment builds on the foundational understanding developed in earlier stages, using enhanced models capable of handling unstructured data from network scans to determine if a network or specific computer within it is vulnerable and to classify the criticality of these vulnerabilities.

Finally, the results of each experiment are analyzed in detail and transferred to the subsequent experiment. The experiments are followed by an overall review of all the

experiments conducted. This involves scrutinizing whether the selected approaches were correct and whether the research questions and additional hypotheses were answered. The hypotheses are critically examined, and a reflection on the research objectives and methodology is carried out.

Due to the complexity of the various Python scripts developed in this thesis, only portions of the source code are included within the main body of the work. The same applies to the results logs and Nmap scan results. Access to the complete source code as well as the result files can be obtained from the following Git repository: [https://github.com/Dzaecko/masterthesis\\_appendix](https://github.com/Dzaecko/masterthesis_appendix).



## 2 Theoretical Framework

This chapter introduces the topic of LLMs, including the current state of research and existing approaches to the use of LLMs in cybersecurity. Furthermore, existing scientific papers that deal with this topic are examined. It also examines which standard tools already exist to detect vulnerabilities in corporate networks and which different types of vulnerabilities are of interest in the course of this work.

### 2.1 Large Language Models

LLMs are advanced artificial neural networks specialized in interpreting and generating natural language [56]. These models are based on a deep learning architecture known as the "Transformer." Deep learning is an approach in machine learning where complex tasks are solved using artificial neural networks with a deep hierarchy of layers, as outlined by Goodfellow et al. [19].

Artificial neural networks (ANN) are models that are used in the field of machine learning.

The idea of an ANN is partly based on the research of Frank Rosenblatt in the 1950s and 1960s. Rosenblatt had the idea of developing an artificial neuron, the so-called perceptron. Inspired by the actual nerve cells that receive and process impulses [42]. The basic idea of the perceptron forms the basis of modern ANNs.

The example in Figure 1 shows a perceptron with three input variables. According to Nielsen [42], this could already be used to develop abstract decision-making.

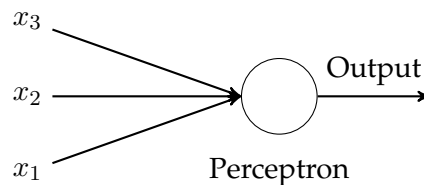


Figure 1: Diagram of a simple perceptron

Technically speaking, the perceptron represents a binary decision matrix in which each input value  $x$  can assume either the state 1 or 0. If each value  $x$  fulfills a condition that is a prerequisite for decision-making, a decision can be made. Rosenblatt has applied the concept of weightings to this so that each value has a specific weighting that makes the input value have a stronger or weaker effect on the output.

The mathematical representation was defined as follows:

A perceptron takes a series of input values  $x_1, x_2, \dots, x_n$ , each of which is multiplied by a weight  $w_1, w_2, \dots, w_n$ . The weighted sum of these inputs is then compared with a threshold value  $\theta$ . The threshold is used to compare the weighted sum and determine whether a positive or negative decision is made. In this case, it determines whether the threshold for recognizing a salient word has been exceeded to determine the output:

$$\text{Output} = \begin{cases} 1 & \text{if } \sum_{i=1}^n w_i x_i \geq \text{Threshold}, \\ 0 & \text{else.} \end{cases}$$

Here, the output is 1 when the weighted sum of the inputs meets the threshold, which means the perceptron identifies the pattern or feature it was trained to detect. The output is 0 when the weighted sum falls below the threshold, signifying that the perceptron does not recognize the pattern. This output enables the perceptron to make straightforward decisions based on the input data. An example is a perceptron that decides whether an email is classified as spam (1) or not (0). The features are the frequency of the words "free" ( $x_1$ ) and "millions" ( $x_2$ ). The weights could be  $w_1 = 0.2$  and  $w_2 = 0.5$ , and the threshold value could be 1.0.

The perceptron function could, therefore, be formulated as follows:

$$\text{Output} = \begin{cases} 1 & \text{if } 0.2 \cdot x_1 + 0.5 \cdot x_2 \geq 1.0, \\ 0 & \text{else.} \end{cases}$$

Suppose an email contains the word "free" three times ( $x_1 = 3$ ) and the word "millions" once ( $x_2 = 1$ ). The weighted sum would be  $0.2 \times 3 + 0.5 \times 1 = 1.1$ , that is greater than 1.0. The email is then classified as spam (output = 1).

Modern ANN are based on Rosenblatt's perceptrons but mainly use sigmoid neurons. According to Nielsen [42], a sigmoid neuron also has input values like a perceptron. But unlike a perceptron, it does not just return 0 or 1. The return is a number in the range between 0 and 1. This means that a sigmoid neuron can represent probabilities, unlike a perceptron.

Sigmoid neurons can be used to create deep neural networks. An ANN is constructed using multilayer networks. An ANN consists of three different layers, which can be formed from any number of neurons. Each of these layers can, in turn, contain sub-layers. The input layer contains  $n$  possible neurons that represent the input values  $x_1, x_2, \dots, x_n$ . Each neuron sends the input value to each neuron in the next layer. The value is multiplied by a weight each time it is passed on, and a bias is added. The bias is an additional parameter that offers various advantages

in neural networks, such as the adjustment of the activation function. The activation function is the actual function that uses the sigmoid function in an ANN with sigmoid neurons to calculate the output between 0 and 1 and pass it on to the next layer. The weighted sum is used as the input parameter for the activation function.

The next layer to which the values of the input layer are sent is the hidden layer. The described process of the activation function is repeated there and calculated in  $n$  hidden layers. The weighted sum in each neuron and in each layer of the hidden layer results in each time from the outputs of the activation function of the connected neurons, multiplied by the weights and added with the bias. The last layer in an ANN is the output layer. Here, the weighted sum of the last neuron is also taken, multiplied by the weight, and the bias is added.

The next layer to which the values of the input layer are sent is the hidden layer. The described process of the activation function is repeated there and calculated in  $n$  hidden layers. The weighted sum in each neuron and in each layer of the hidden layer results in each time from the outputs of the activation function of the connected neurons, multiplied by the weights and added with the bias.

For the  $j$ -th neuron in the  $l$ -th hidden layer, the weighted sum  $z_j^{(l)}$  is calculated as follows:

$$z_j^{(l)} = \sum_{i=1}^{n_{l-1}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)},$$

Where:

- $w_{ij}^{(l)}$  is the weight between the  $i$ -th neuron of the  $(l - 1)$ -th layer and the  $j$ -th neuron of the  $l$ -th layer,
- $a_i^{(l-1)}$  is the output of the  $i$ -th neuron of the  $(l - 1)$ -th layer,
- $b_j^{(l)}$  is the bias term of the  $j$ -th neuron in the  $l$ -th layer.

The output  $a_j^{(l)}$  of the  $j$ -th neuron in the  $l$ -th layer is then calculated by applying the activation function  $\sigma$  to the weighted sum  $z_j^{(l)}$ :

$$a_j^{(l)} = \sigma(z_j^{(l)}).$$

The last layer in an ANN is the output layer. Here, the weighted sum of the last neuron is also calculated, multiplied by the weight, and the bias is added. For the

$k$ -th neuron in the output layer, the weighted sum  $z_k^{(L)}$  is calculated as follows:

$$z_k^{(L)} = \sum_{j=1}^{n_{L-1}} v_{jk} a_j^{(L-1)} + c_k,$$

Where:

- $v_{jk}$  is the weight between the  $j$ -th neuron of the last hidden layer and the  $k$ -th neuron of the output layer,
- $a_j^{(L-1)}$  is the output of the  $j$ -th neuron of the last hidden layer,
- $c_k$  is the bias term of the  $k$ -th neuron in the output layer.

The output  $y_k$  of the  $k$ -th neuron in the output layer is then calculated by applying the activation function  $\sigma$  to the weighted sum  $z_k^{(L)}$ :

$$y_k = \sigma(z_k^{(L)}).$$

To explain the process clearly using a minimal example, let us think of an example that can be used in the development of an AI-controlled computer game.

**Definition 2.1. Pong Game:** In the game Pong, two bats have to be moved along the Y-axis to hit a ball. The ball has a position in the coordinates X and Y. Suppose you want to develop a neural network that is able to determine the Y-position of the bat. The neural network should calculate how far the bat must be moved along the Y-axis to hit the ball.

**Definition 2.2 (Neural Network Specifications).** A neural network could be developed with the following specifications:

- **Input Layer:** 3 neurons ( $x_{\text{ball}}, y_{\text{ball}}, y_{\text{beater}}$ )
- **Hidden Layer:** 4 neurons (with weights  $w_{ij}$  from each input to each hidden neuron and bias  $b_j$  for each hidden neuron)
- **Output Layer:** 1 neuron (with weights  $v_k$  from each hidden neuron to the output and bias  $c$  for the output neuron)

**Definition 2.3 (Weights and Biases).** • **Weights:**

- Between input and hidden layer ( $w_{ij}$ ): randomly initialized
- Between hidden and output layer ( $v_k$ ): randomly initialized
- **Biases:**
  - For hidden layer ( $b_j$ ): randomly initialized
  - For output layer ( $c$ ): randomly initialized

**Example 2.1.** In order to calculate a corresponding output value, the following is an example of a passageway calculation with values:

- Inputs:
  - $x_{\text{ball}} = 0.5$
  - $y_{\text{ball}} = 0.5$
  - $y_{\text{bat}} = 0.3$
- Weights from input to hidden ( $w_{ij}$ ):
  - $w_{11} = 0.2, w_{12} = -0.3, w_{13} = 0.4$
  - $w_{21} = 0.1, w_{22} = -0.2, w_{23} = 0.5$
  - $w_{31} = -0.1, w_{32} = 0.3, w_{33} = -0.4$
  - $w_{41} = 0.3, w_{42} = -0.1, w_{43} = 0.2$
- Weights from hidden to output ( $v_k$ ):
  - $v_1 = 0.5, v_2 = -0.4, v_3 = 0.3, v_4 = 0.1$
- biases:
  - Hidden layer ( $b_j$ ):  $b_1 = 0.1, b_2 = -0.1, b_3 = 0.05, b_4 = 0.2$
  - output layer ( $c$ ):  $c = 0.1$
- 1. **calculation of the hidden layer neurons:** Each neuron in the hidden layer calculates the weighted sum of its inputs and bias and applies the sigmoid activation function.

$$h_1 = \frac{1}{1 + e^{-(0.5 \cdot 0.2 + 0.5 \cdot 0.1 - 0.3 \cdot (-0.1) + 0.1)}} \approx 0.5546$$

$$h_2 = \frac{1}{1 + e^{-(0.5 \cdot (-0.3) + 0.5 \cdot (-0.2) + 0.3 \cdot 0.3 - 0.1)}} \approx 0.4350$$

$$h_3 = \frac{1}{1 + e^{-(0.5 \cdot 0.4 + 0.5 \cdot 0.5 + 0.3 \cdot (-0.4) + 0.05)}} \approx 0.5930$$

$$h_4 = \frac{1}{1 + e^{-(0.5 \cdot 0.3 + 0.5 \cdot (-0.1) + 0.3 \cdot 0.2 + 0.2)}} \approx 0.5890$$

2. **calculation of the output neuron:** The output is also calculated as the weighted sum of the hidden neurons plus bias:

$$y_{\text{new}} = \frac{1}{1 + e^{-(h_1 \cdot 0.5 + h_2 \cdot -0.4 + h_3 \cdot 0.3 + h_4 \cdot 0.1 + 0.1)}} \approx 0.6083$$

The result  $y_{\text{new}} = 0.61$  would be the new Y-value suggested by the network for the bat to hit the ball. To train the network effectively, one would measure the output error and optimize the weights and biases through many iterations of training data using backpropagation and gradient descent.

**Definition 2.4. Forward Propagation:** The network that has just been described and receives an input  $x$  and generates an output  $y$  is referred to as forward propagation according to Goodfellow et al. [19].

**Definition 2.5. Backpropagation:** Backpropagation attempts to minimize the difference between the output of the ANN and the desired target. The error is calculated using a loss function, and the network is run backward using gradient calculation to adjust the values for the weights and bias. With each run, an attempt is made to achieve an optimized output that approximates the target.

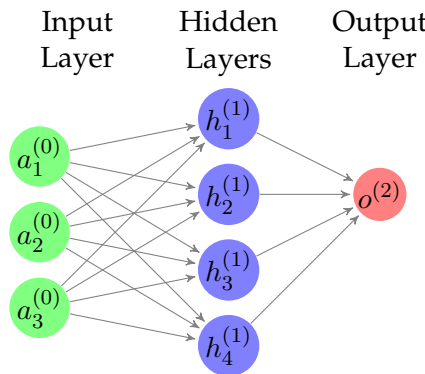


Figure 2: Neural network architecture with 3 input neurons, 4 hidden neurons, and 1 output neuron.

ANNs are very suitable for clear, context-independent tasks. However, when more complex tasks need to be solved, such as processing natural language or other sequence-based data, such as creating source code or audiovisual data, or detecting and analyzing vulnerabilities in corporate networks, as in this master's thesis, more specific approaches are required.

The Transformer architecture was specially developed for such tasks, which require context and also comprehension relationships over long distances within the data.

The Transformer architecture, first presented at the Neural Information Processing Systems (NIPS) conference in 2017 in the paper "Attention is All You Need", is divided into two main components: the encoder and the decoder. The encoder receives input in the form of a text sequence and processes it sequentially. For instance, the sentence 'In a democracy, the people' would be divided into tokens like ['In,' 'a', 'democracy,' 'the,' 'people']. Each of these vectors is represented in a multidimensional space. The encoder evaluates these vectors using the Self-Attention model, generating an "attention score" through more complex mathematical transformations, including normalization and the application of a Feed Forward Neural Network (FFNN).

The decoder operates similarly. It reprocesses the output of the encoder using the Self-Attention model to generate a response. This response depends on the specific model and the training data used. When the Transformer architecture was introduced in 2017, it was trained with datasets from the Workshop on Machine Translation (WMT) for English-German and English-French translations [56].

According to Vaswani et al., [56], both the encoder and the decoder are the main components of the Transformer architecture, as illustrated in Figure 3. The previously described example sentence ('In a democracy, the people') is converted into vectors in the input embedding as described. Then, the Attention Layer calculates how much attention one word should pay to another. In the example mentioned here, the word 'democracy' could direct attention to 'people.' New vectors with these attention relationships are created. The newly formed vectors subsequently go through a FFNN and generate new vectors. To stabilize and accelerate the process, normalizations are performed, and residual connections are added to each of these layers. The softmax function considers the determined attention score of the words from the input and calculates a probability distribution in which each value lies between 0 and 1 [19]. In the softmax function, exponential values of the attention scores of the individual words are calculated and divided by the sum of all exponential values. This function, applied to the example sentence ['In,' 'a', 'democracy,' 'the,' 'people'], would direct significant attention between the words 'democracy,' 'people' and 'the' depending on the calculation of the attention score.

$$\text{Softmax}(s_i) = \frac{e^{s_i}}{\sum_j e^{s_j}}$$

Depending on the pretraining, the model would then select the next word that is likely to fit into the learned context. This could be the word "have," for example. The whole process of the transformer would then be run through with the new input sentence plus the word "have" to generate the next word. This process would run until either the maximum tokens are used up or the sentence is finished. In the example given here, this could lead to the result with the generated sentence: "In a democracy, the people have the power to elect their leaders."

In a paper by Kaiming He et al. [12], these types of connections were introduced. These residual connections can offer several benefits to the processing, such as addressing the issue of vanishing gradients in intense networks and allowing a network to gain advantages from increased depth without adding complexity or difficulty to the training process.

This process is repeated in a certain number of layers. In the example of Vaswani et al., [56], the process was run through 6 encoder layers and six decoder layers. In GPT-3, for 96 layers each, and in GPT-4, for 120 layers each.

The output sequence generated thereby represents the original input with rich, learned context through the Transformer architecture. Depending on the chosen model, different types of outputs can be generated. In GPT-3 and GPT-4, whose goal is language modeling, for example, the example sentence would be completed, and the output could be: 'In a democracy, the people have the power to elect their leaders.' Other types of outputs include text translation, text classifications, sentiment analysis, and Named Entity Recognition.

In the realm of Natural Language Processing (NLP), LLMs, like the Transformer architecture, have become indispensable tools for a variety of tasks. The reason for their effectiveness lies in their ability to understand the sequential and contextual nature of language. Traditional machine learning algorithms often need help with handling the intricacies and nuances of natural language, such as idioms, phrasal verbs, and contextual meanings. LLMs, on the other hand, can capture these complexities through their deep learning architecture and large number of parameters. This enables them to outperform traditional algorithms in tasks like text summarization, machine translation, and sentiment analysis. As a result, LLMs have opened new avenues for research and practical applications in NLP, making the field more robust and versatile than ever before.

More advanced models, like GPT-3, build upon this architecture but vary in application. GPT-3, for instance, only uses the decoder and has a significantly higher



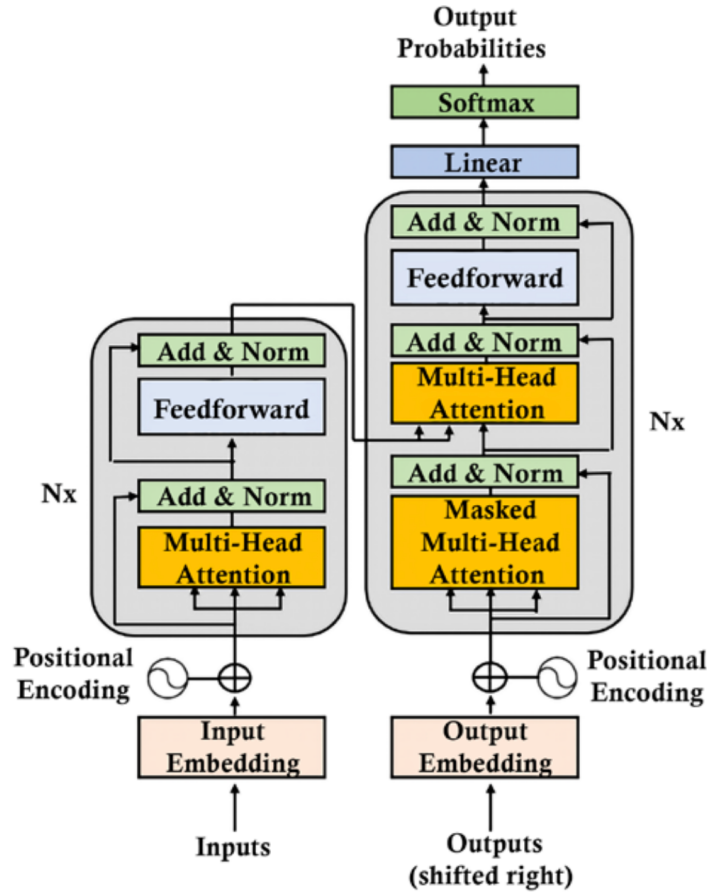


Figure 3: Transformer architecture from Vaswani et al. [56]

number of parameters compared to other models. To clarify the complexity, The model in "Attention is All You Need" had up to 213 million parameters. [56], GPT-3 has 175 billion, and GPT-4 even has up to 1 trillion parameters, as mentioned by Brown et al. [7].

A high number of parameters indicates the complexity of the underlying neural network. These parameters are calculated through the connections between different layers in a neural network (Input, Hidden, Output), along with their weightings and biases. For example, a neural network consisting of 3 input neurons, four hidden neurons, and one output neuron has, in comparison to large language models, a parameter count of 21. This allows the network to learn a game like Pong (see Figure 2).

The total number of parameters  $P$  in a fully connected neural network is calcu-

lated by the following formula:

$$P = \sum_{i=1}^{n-1} (N_i \times N_{i+1} + N_{i+1})$$

where:

- $N_i$  is the number of neurons in the  $i$ -th layer,
- $N_{i+1}$  is the number of neurons in the  $(i + 1)$ th layer,
- $N_i \times N_{i+1}$  represents the number of weights between layer  $i$  and layer  $i + 1$ ,
- $N_{i+1}$  represents the number of bias values in layer  $i + 1$ .

In comparison to the parameters within an LLM like GPT-4 with 1 trillion parameters, the complexity is quite evident. For the training of GPT-3, as mentioned by Brown et al., [7], a version of the CommonCrawl archive was used. Given that the data basis for GPT-4 only extends up to September 2021, the archive of that month was likely used for the training of GPT-4. As of the specified time, the archive comprised 2.95 billion web pages, amounting to 310 terabytes of uncompressed data. Prior to training, this data underwent a filtering and deduplication process. The training also incorporated extended versions of the WebText dataset and the literary datasets Books1 and Books2; however, the precise contents of these datasets were only partially revealed due to copyright constraints. The entirety of the English Wikipedia, up to September 2021, was also included. It should be noted that in contrast to open-source models such as BERT [8], OpenAI’s models are proprietary, leading to a need for more transparency.

### 2.1.1 Retrieval-Augmented Generation

LLMs such as GPT-4 or Llama are pre-trained models that have already been trained using a large corpus of knowledge. This last state of pretraining defines a static limit of knowledge that an LLM has [7]. Furthermore, there is a known problem with LLMs, especially when they are supposed to access knowledge that is outside the trained knowledge. This problem is called hallucinations. According to [58], hallucination occurs because models have learned specific data patterns during the training process and then tend to overgeneralize these patterns when they encounter new or different data.

During the evaluation of LLMs in the context of vulnerability detection, this behavior could be simulated. For this purpose, the GPT-4 model was asked whether it can detect vulnerabilities at different CVE IDs (see section 2.5). For this purpose,

a CVE entry from 2010, i.e., before the training of the model, as well as a CVE entry from the time of training and a CVE entry that was only discovered after the training of the GPT-4 model, were used for a random sample.

For the sample with vulnerability CVE-2024-22317 affecting IBM App Connect Enterprise 11.0.0.1 to 11.0.0.24 and 12.0.1.0 to 12.0.11.0, which was created after the training was created, the model correctly responded that this entry was created after the model was trained and therefore no information was available.

The vulnerability CVE-2010-3683, which affects Oracle MySQL 5.1 before 5.1.49 and 5.5 before 5.5.5, was not correctly identified by the LLM, but a vulnerability was plausibly described. However, this vulnerability does not exist and cannot be compared with an existing vulnerability. It is, therefore, assumed that the model hallucinated here, and the description was freely generated.

For entry CVE-2020-29662, the LLM correctly identified the vulnerability and described it in detail (correct description, correct CVSS score, etc.).

The results from experiment 1 (see section 3.3) and experiment 2 (see section 3.5) are used to examine this case in more depth, where different models are tested in the pre-trained state for the detection of vulnerabilities.

This sample was repeated several times, constantly with the same result. This leads to the assumption that the corpus for the training of GPT-4 contains samples of more recent CVE entries but not older entries and then declares a plausible vulnerability by hallucinating them, but this is not correct. Because CVE entries contain the year in their name, the model was able to understand the context to the extent that the CVE entry was only created after the pretraining was created, and it, therefore, assumes that it has no knowledge of it. This prevented hallucination.

The answers that are generated seem plausible but are then factually incorrect. The problem occurs mainly when models are confronted with new or unknown data. For example, data that was not available at the time of pre-training the LLM would be generated hallucinatorily. In an example, from [58], a model that was trained with data up to the year 2022 could not answer which country would host the Olympic Games in 2023, for example. This confirms the result of the samples from this work. The answer is to clarify that the model does not have this data, or it could hallucinate and incorrectly output any country as the venue.

RAGs build a bridge to this. According to [26], a RAG is a system for improving the accuracy and reliability of LLMs and generative AI models by adding facts from external sources. RAGs provide a solution by adding information that is either newer than what the model already had in pretraining or information that was not previously part of the model's pre-training. [26] distinguish between implicit parametric memory, the knowledge that a model has already obtained through pretrain-

ing, and non-parametric memory, the knowledge that a model obtains through external knowledge after pre-training. Referencing external knowledge significantly reduces the problem of incorrect content generation. RAGs are considered a key technology for the further development of chatbots and for improving the suitability of LLMs for real-world applications [17]. One advantage of using a RAG is that the model does not have to be fine-tuned in context, so the data does not have to be integrated directly into the model and can also be used across different models([58]).

When creating a RAG, the input text is converted by a process so that it can later be saved in a vector database. There are various approaches for this, the most widespread being the Byte Pair Encoding (BPE) approach, which is also used in the OpenAI models ([5]). First, the entire input text is converted to lowercase letters. An algorithm is then used to split the individual letters or words into tokens. The same procedure is also used in the Transformer architecture 2.1 when entering text.

As an example to link to the research in this thesis, the following CVE entry is used for the process within a RAG:

The CVE entry "CVE-2023-50870, In JetBrains TeamCity before 2023.11.1 a CSRF on login was possible, CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N, CWE-352" was broken down into 58 tokens using the "text-embedding-ada-002" model (see section 3.2.1). Each token represents a specific entry in the model's vocabulary. To understand this approach in more detail, a test was made with the online version of the TikTokenizer in this paper.<sup>3</sup> An array of indices was created using the sample text (see Array 2.1.1). The word CVE occupies exactly the place with the index 68902 in the vocabulary. For efficiency, not every word within the vocabulary of the model is represented by its index, but words are also made up of several tokens.

68902	12	2366	18	12	19869	2031	11
644	76224	8068	13020	1603	220	2366	18
13	806	13	16	264	79695	389	5982
574	3284	11541	53	1242	25	18	13
16	14	8253	38794	14	1741	70333	14
6616	38794	75983	25	49	11628	25	52
11547	70333	39251	38794	10576	38794	11541	12739
12	16482	720					

These indices would be created for each entry in a CSV file, for example, and stored with enriched metadata in a vector database. There are currently cloud solutions that provide vector databases, but there are also on-premise solutions such

<sup>3</sup><https://tiktokenizer.vercel.app/>

as LlamaIndex<sup>4</sup> or Langchain<sup>5</sup>. Cloud based solutions are offered by Azure<sup>6</sup> (see section 3.2.2), Databricks<sup>7</sup>, AWS<sup>8</sup> or Qdrant<sup>9</sup>.

With on-premise solutions such as Llamaindex, the vector databases are saved in JSON format 3.2.2.

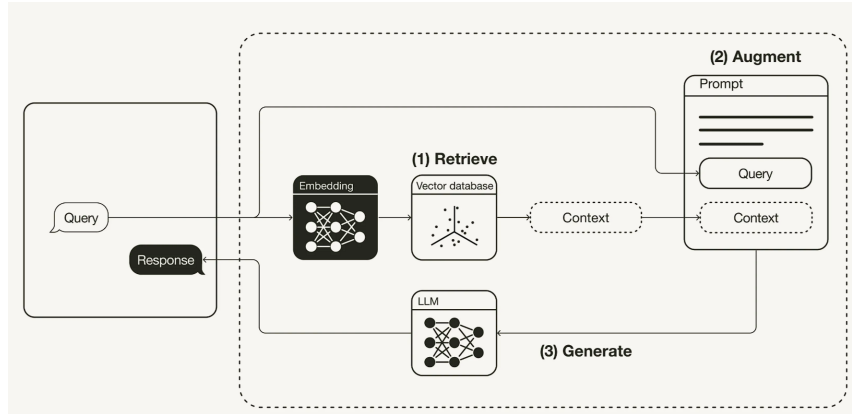


Figure 4: Retrieval-Augmented Generation Workflow. from <https://towardsdatascience.com/retrieval-augmented-generation-rag-from-theory-to-langchain-implementation-4e9bd5f6a4f2>

As illustrated in Figure 4, a RAG system consists of various processes, which are divided into Retrieve, Augment, and Generate. For the retrieval process, as explained above, the input text is converted into embeddings (tokenized), and the tokens contained in the input text represent the semantic content. Semantic search is applied to this content. Semantic search goes beyond traditional keyword-based search [21]. Semantic search aims to find specific information by analyzing the context and meaning behind the search queries. It uses the Nearest Neighbor (NN) algorithm, also known as parallel search or nearest point search. It is a nearest neighbor point search problem. The nearest neighbor extraction takes into account conditions related to the geometric effects of the objects [21]. This content is often processed by an Approximate Nearest Neighbor (ANN) algorithm or the k-nearest Neighbors (KNN) to find the nearest neighbors within the vector database and form a context.

Both algorithms use cosine similarity to calculate the distance between vectors.

<sup>4</sup><https://www.llamaindex.ai/>

<sup>5</sup><https://www.langchain.com/>

<sup>6</sup><https://azure.microsoft.com/en-us/products/ai-services/ai-search>

<sup>7</sup><https://www.databricks.com/>

<sup>8</sup><https://aws.amazon.com/de/bedrock>

<sup>9</sup><https://qdrant.tech/>

$$\text{Cosine Similarity}(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

For example,  $A$  forms the vector that is calculated by embedding the input, and  $B$  forms various vectors from the vector database.

Algorithms such as ANN and k-NN are suitable for semantic searches and have become established in the use of RAGs.

The k-NN provides exact results and is, therefore, ideal in situations where precision is crucial. K-NN guarantees that the k-nearest neighbors found actually have the shortest distances to the query point. ANN provides an approximate solution that only sometimes provides the absolute shortest distances but comes sufficiently close in most cases. This is useful in applications where a slight deviation from the exact solution is acceptable if it significantly improves performance [6].

The resulting context is generated into a prompt together with the actual input in the RAG process. With this enhanced prompt, the LLM can now work on the context to produce a coherent and informative response.

## 2.2 Cybersecurity

Since the introduction of computer technologies, there has been a threat that private, sensitive, or critical data could be compromised by unauthorized access. In the 1980s, this threat manifested itself in particular through the spread of the first various computer viruses, including the Elk Cloner[25], the Brain Virus<sup>10</sup>, and the Stoned Virus<sup>11</sup>, and despite the absence of the Internet as we understand it today, the dissemination of this malware took place primarily via portable storage media such as floppy disks and within the rudimentary internal networks of organizations. For the most part, these viruses represented experimental attempts by developers to enable research into potential vulnerabilities rather than posing an immediate threat to data security. The Elk Cloner, for example, was developed by Rich Skrenta as a high school joke to test the ability of software to self-replicate [25]. Even though the Elk Cloner was relatively harmless, for example, it displayed poems when starting a floppy disk, this virus has historical significance as it is considered the first self-propagating virus [25].

These early forms of malware were not yet technically capable of extracting data or giving outsiders access to internal networks. At that time, the real threat was less in the virtual domain and more in the physical domain. Due to the prevailing isolation or closed nature of computer systems, the focus of security efforts was not on defending against external network attacks but on protecting against physical access, data theft, and internal threats. Even within the ARPANET, the forerunner of the modern Internet, security concepts were considered secondary, as access was limited to an exclusive circle of people with a military or academic background [48].

Public interest was first aroused in 1988 when Robert Tappan Morris developed a program that, according to Robert's statements, actually intended to count the computers on the then-young Internet. Due to a program error, the infected computer replicated the software independently, which led to an enormous utilization of resources. At the time, around 6,000 computers were infected with the program, which at the time accounted for around 10% of the entire Internet. This program went down in history as the Morris worm, the first worm to spread on the Internet [23].

As a result of this incident, the first official team, the Computer Emergency Response Team (CERT), was formed to protect against, detect, and respond to an organization's cybersecurity incidents. The term *cybersecurity* was not yet widely used at that time. It was more commonly referred to as computer security or network security [9].

---

<sup>10</sup><https://medium.com/geekculture/brain-the-worlds-first-computer-virus-f3758323d894>

<sup>11</sup><https://www.f-secure.com/v-descs/stoned.shtml>

Due to the rapid increase in technologies and the spread of personal computers, as well as the further expansion of the Internet, the topic of cyber security became more and more present, so private companies and official authorities became more intensively involved with the topic. The idea of CERTs was transferred to other countries, and the state developed the first legislation and regulations [43, 49, 55].

Today, the field of cybersecurity covers a wide range of areas that protect systems, networks, companies, and private individuals from digital attacks or other security breaches. Cybersecurity covers the following areas in particular:

- Network security
- application security
- end device security
- data security
- Identity and access management
- Operational security
- Compliance

In recent years, more and more LLMs have also taken their place in cybersecurity. According to Nourmohammadzadeh et al., [44], the first documented use of LLMs was in 2018, where the AWD-LSTM model was used to generate a malicious spear phishing link [24]. The paper also mentioned various attack vectors through the use of LLMs (mostly GPT models). The attack vectors are divided into two particularly recognizable ones: One is in support of phishing attacks by creating convincing and easy-to-read phishing emails and web pages through the models' language capabilities. The other attack vector that LLMs can clearly exploit is the creation of malware or malicious program code. Since LLMs are capable of generating code in a variety of programming languages, they can be used to generate not only benign code but also malware.

In the same year that the first documented use of the AWD-LSTM model for attacks was reported, the first LLM was also used in the course of defensive applications. In 2018, the RNN model was used to detect abnormal behavior in network protocols. In the paper, the NIST Cybersecurity Framework (CSF) was used to classify in which areas there are already elaborations in the course of using LLMs as defensive applications. The CSF was developed to help individual companies and other organizations identify the cybersecurity risks to which they are exposed. Accordingly, the risks are divided into five functions defined by NIST:



1. **Identify:** Develop an organizational understanding to manage cybersecurity risk to systems, people, assets, data, and capabilities.
2. **Protect:** Develop and implement appropriate safeguards to ensure delivery of critical services.
3. **Detect:** Develop and implement appropriate activities to identify the occurrence of a cybersecurity event.
4. **Respond:** Develop and implement appropriate activities to take action regarding a detected cybersecurity incident.
5. **Recover:** Develop and implement appropriate activities to maintain plans for resilience and to restore any capabilities or services that were impaired due to a cybersecurity incident.

The research by Nourmohammadzadeh et al. has shown that LLMs have been mainly researched in the functions *Detect* and *Protect*. Among the research in *Protect* mentioned there, LLMs were used to investigate how well LLMs can correct software vulnerabilities in programs. For example, one of Pearce et al. [46] research questions was: "Can off-the-shelf LLMs generate safe and functional code to fix security vulnerabilities?" Pearce et al. were able to answer this research question by using experiments as the chosen method. The researchers used various commercial models, such as code-davinci-001 from OpenAI and a GPT-2-based model, which they had created themselves.

The research teams generated targeted vulnerabilities in software, including Python and C-based programs. These vulnerabilities were based on known Common Vulnerabilities and Exposures (CVE) entries. Based on two CVEs, CVE-89 (SQL Injection) and CVE-787 (Out-of-bounds Write), a database of vulnerable scripts was created. For this purpose, various LLMs were used to create scripts that contained a vulnerability of the respective CVEs. In order to generate as many programs as possible, the settings (Temperature and Top P) with which the most compilable programs could be generated were evaluated.

In the study, the OpenAI LLMs code-cushman-001 and code-davinci-001 were used to generate ten programs for each combination of the parameters "Temperature" ( $T$ ) and "Top P" ( $P$ ), with the values {0.00, 0.25, 0.50, 0.75, 1.00}. This resulted in a total of 250 suggestions for each model, giving a total number of 500 programs. The aim was to check the compilability of each generated program before evaluating it in terms of functional and safety aspects.

The results of the analysis can be calculated with this formula like, described be-

fore:

any combination  $(T_i, P_j)$ , with  $T_i, P_j \in \{0.00, 0.25, 0.50, 0.75, 1.00\}$ ,

led to the generation of  $N_{ij} = 10$  programs per model

This resulted in 250 suggestions per model and a total of 500 programs.

The analysis revealed that 95 functional and vulnerable programs were identified for CWE-787 and 22 corresponding programs for CWE-89 among the generated programs. These executable and vulnerable programs were then used as a database to discuss the extent to which code-cushman-001 and code-davinci-001 can repair these programs. The Temperature/Top P matrix was used again for this; instead of 10 programs per field, the corresponding generated programs were used. This led to the creation of 47,500 programs for the scenarios based on CWE-787 and, similarly, 11,000 programs for CWE-89. For CWE-89, with 10,796 valid programs, the success rate was 29.6% (3,197 programs). The test results illustrate the potential of LLMs, especially the Codex model, in identifying and repairing software vulnerabilities. The varying success rate between the different vulnerability types underlines the need for further research to optimize the efficiency of these models in various application scenarios.

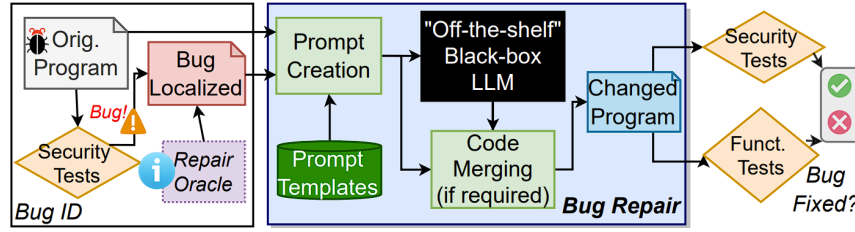


Figure 5: Examining Zero-Shot Vulnerability Repair with Large Language Models from Hammond Pearce et al.

### 2.3 State of research

The steadily increasing threat from cyber threats poses significant challenges to the security of various systems in light of the ongoing evolution of attack tactics and their rapid adaptation to new technologies. Ferrag et al. [13] argue that the need for advanced and effective detection mechanisms is of crucial importance.

Since the proliferation of LLMs, interest has increased in various security fields for using this modern technology to improve cybersecurity. As the topic of LLMs in the context of IT security is still new, no printed literature on this subject could be found during the initial literature search. However, several works have already

been published that deal with this issue. These are mostly pre-trained LLMs tailored to specific domains. Notably, BERT has been used multiple times in the field of IT security to develop a domain-specific LLM. Examples include CySecBERT [4], ExBERT [22], MalBERT [47], and CyBERT [2].

Further research approaches focus on the analysis and classification of software vulnerabilities. A paper from Chandra Thapa et al. [54] specifically addresses this topic. The focus is not on fine-tuning a particular model but on evaluating various models. Chandra et al. [54] were able to demonstrate that models like GPT-2 Large and GPT-2 XL outperformed models like CodeBERT [12] and DistilBERT in detecting vulnerabilities. The results show a general improvement in performance with increasing model size. The studies were limited to vulnerabilities in the programming languages C and C++.

Another documented approach is MalBERT [13], developed by Rahali and Akhloufi in 2021. Rahali and Akhloufi aimed to create a framework for detecting Android malware using a transformer-based approach, mainly using BERT. This process starts with the collection of data on Android applications from the public Androzoo dataset, which includes both malicious and benign apps. This data is filtered to include 12,000 benign and 10,000 malicious apps from 11 different categories, such as Adware, Spyware, and Ransomware. The downloaded APK files are then decompiled, and the AndroidManifest.xml files are extracted. BERT is used to fine-tune the training set to perform two types of classifications: a binary classification to identify malware and a multi-class classification to identify the malware category. The authors use four key metrics to assess model performance: Accuracy, Matthews Correlation Coefficient (MCC), F1-Score, and Cross-Entropy Loss. The results show that the BERT model outperforms all other models in both classification tasks. Specifically, BERT achieved an accuracy of 0.9761, an F1-Score of 0.9547, a loss value of 0.1274, and an MCC of 0.9559 [13].

Previous research has explored the applicability of LLMs in various areas of IT security. This includes the analysis of cybersecurity texts, where models like CySecBERT [4] and ExBERT [22] have been developed to analyze specific texts in the field of cybersecurity and detect threats or anomalies in cybersecurity. Another focus is on vulnerability detection in source code, where studies from Chandra Thapa et al. [54] have evaluated the efficiency of LLMs in detecting vulnerabilities in source code. Additionally, there are projects like MalBERT [13], which explore the capability of LLMs in detecting Android malware. While these research efforts provide valuable insights, they mainly focus on specialized use cases and leave broader contexts unexplored.

Despite LLMs promising applications in IT security, there are clear gaps that need further exploration. A key point is the comparative analysis of LLMs efficiency with traditional vulnerability detection tools in enterprise networks. While existing re-

search primarily focuses on specific applications and domains, the question remains open as to how LLMs perform compared to traditional tools in a broader context.

Addressing these research gaps could not only significantly contribute to science but also have practical implications for improving cybersecurity in enterprise networks. The answers to these questions could assist companies in making informed decisions about deploying LLMs in their IT infrastructure.

## 2.4 Vulnerability Assessment Tools in IT Security

Vulnerability Assessment Tools have long served as the cornerstone for identifying, classifying, and managing vulnerabilities within IT networks. These tools span a broad range of specialized software. From web application scanners like OWASP ZAP, Burp Suite, or Nikto, to comprehensive network scanners such as Nessus<sup>12</sup> and OpenVAS<sup>13</sup>. Network scanners generally offer broad scanning capabilities, including the identification of open ports, misconfigured firewalls, and known vulnerabilities specific to network services. Web application scanners are tailored for specific vulnerabilities like SQL injection, XSS, and CSRF.

Many of these tools rely on signature-based detection methodologies, utilizing databases of known vulnerabilities like CVE databases. Some advanced tools employ heuristic methods to discover vulnerabilities that may not yet be cataloged.

This background on vulnerability assessment tools will serve as a comparative foundation for evaluating the effectiveness and efficiency of LLMs in similar roles.

## 2.5 CVE

The idea of creating a Common Enumeration of Vulnerabilities came from David E. Mann and Steven M. Christey in 1999, when they explained in their paper [8] for the MITRE Corporation, a non-profit organization based in America that deals with cybersecurity, among other things, that the CVE system can be used to define a uniform collection of vulnerabilities. A CVE forms a logical bridge between a tool used and a database that contains the description of the vulnerability discovered in the tool, with enriched information on the version and possible implied CWE entries. Since 1999, the database of all tools with known vulnerabilities has been growing.

A CVE record can always be clearly identified by its CVE ID. A CVE record always

---

<sup>12</sup><https://www.tenable.com/>

<sup>13</sup><https://openvas.org/>

goes through a defined life cycle until it is generally published (see Figure 6) <sup>14</sup>. As of today (April 2024), there are 228,713 published CVE entries available for download.

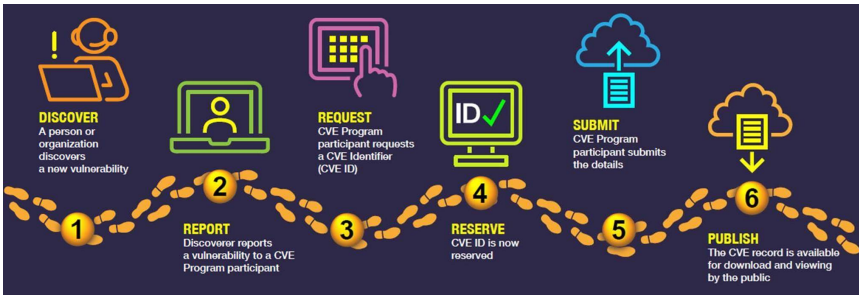


Figure 6: CVE Record Lifecycle

The CVE database is the leading database for software vulnerabilities and is constantly updated. It serves as a fundamental basis for classifying the most common vulnerabilities.

Consequently, the CVE database also forms a basis for this master thesis, supplemented by data from the Exploit DB (see Chapter 2.7) and the primary CVSS data from Chapter 2.6. The complete existing data sets are accessed for use in this thesis. MITRE offers several ways to download the dataset; in the traditional way, the data is available in CSV, HTML, text, and XML formats. In addition, all CVE entries are also available for download in JSON 5.0 format.

For research reasons, both the traditional CSV file format and the JSON 5.0 format are used in this paper. The structure of the CSV format is shown in table 1.

Table 1: Summary of the CVE Database Fields

Field Name	Description
Name	The unique identifier of the security vulnerability, known as the CVE ID.
Status	The status of the CVE entry, indicating whether it is a candidate or a recognized entry.
Description	A brief description of the security vulnerability detailing its nature and potential impact.
References	References to resources that provide additional information about the vulnerability. This may include URLs, advisories, or database entries.
Phase	The stage of the CVE entry in the CVE lifecycle, indicating its maturity and whether it has been reviewed.
Votes	Records of the voting results from the CVE board members, reflecting a consensus on the vulnerability.

*Continued on next page*

<sup>14</sup><https://www.cve.org/About/>

Table 1 – Continued from previous page

Field Name	Description
Comments	Additional remarks or discussions from the CVE board or analysts, which may include insights or clarification on the entry.

A significant problem that became apparent during the research phase also concerns a gap in the information retrieval of data from the corresponding datasets. As the missing link between the classification of CVSS scores (see 2.6) and the classification of existing exploits (see 2.7) shows, queries using a Python script revealed that, as listed in table 2, over half (58.56 %) of all CVE entries only contain a reference to the affected software and the corresponding version in the description. This suggests that a classic database query or a keyword search challenges the identification of vulnerabilities for classic vulnerability assessment tools, as an exact statement on the severity and identification of a vulnerability can be derived across various data sources.

Table 2: CVE Records Overview

Description	Number or Percentage
Total Number of CVE Records	238,151
CVE Records without Name or Version	139,458
CVE Records without Version	139,408
Percentage Missing Name or Version	58.56%
Percentage Missing Version	58.54%

It was also noticed during the research that versions are often not clearly defined in the description. Words such as "before", "after", "up to", "earlier", "lessThan", "lessThanEqual" and "or prior" are often used to describe a whole series of versions. An examination of the keywords during the research phase revealed that 43% of all current CVE records contain these keywords in the description. There is no uniform structure in the use of these terms; instead, they are used in an unstructured manner in different contexts. This makes it difficult to automatically transfer the information into relational database systems or other structured data systems. Furthermore, the CSV and JSON data formats are each structured differently.

For example, the CVE entry CVE-2008-3716 in CSV format has the following description: "Cross-site request forgery (CSRF) vulnerability in Harmoni before 1.6.0 allows remote attackers to make administrative modifications via a (1) save or (2) delete action to an unspecified component."

In JSON format, on the other hand, versioning is not handled according to a fixed structure. For example, for CVE-2020-9497, the version is stored together with the

software name in the "Version" field, with the information that all versions from 1.1.0 and older are affected.

Listing 1: CVE-2020-9497 Version

```
1 {
2   "versions": [
3     {
4       "status": "affected",
5       "version": "Apache Guacamole 1.1.0 and older"
6     }
7   ]
8 }
```

For CVE-2021-24383, version 8.1.12 for WP Google Maps is described as affected in the JSON format. The "lessThan" field also mentions version 8.1.12. This combination is not technically correct, as the description of the CVE states: "The WP Google Maps WordPress plugin before 8.1.12 did...". This description supports the assumption that the "version" field was mishandled.

Listing 2: CVE-2020-9497 Version

```
1 {
2   "versions": [
3     {
4       "lessThan": "8.1.12",
5       "status": "affected",
6       "version": "8.1.12",
7       "versionType": "custom"
8     }
9   ]
10 }
```

In the listing for CVE-2022-4289, instead of the fields "lessThan," etc., the versions are described using mathematical operators.

Listing 3: CVE-2020-9497 Version

```
1 "affected": [
2   {
3     "vendor": "GitLab,"
4     "product": "GitLab,"
5     "versions": [
6       {
7         "version": ">=15.3, <15.7.8",
8         "status": "affected"
9       },
10      {
11        "version": ">=15.8, <15.8.4",
12        "status": "affected"
13      },
14      {
15        "version": ">=15.9, <15.9.2",
16        "status": "affected"
17      }
18    ]
19  }
20 ]
```

The assumption here is that LLMs are better than traditional assessment tools at recognizing the context as well as the semantic nuances and contextualizations within the description. For example, if the software is installed with version 1.5.4, static tools may need help to make the connection between these versions. More in-depth research is considered in chapter 3.5. A fully structured form in the form of a

relational database was also not found during the research phase and during the literature review. As described in the following sections, other data sources such as the Exploit DB and CVSS scoring exist, which, via larger databases such as the National Vulnerability Database (NVD) from NIST, The National Institute of Standards and Technology (NIST) is a U.S. government agency focused on advancing measurement science, standards, and technology to promote innovation and competitiveness [41], offers a further challenge in providing an organized overview across different systems. Therefore, the assumption here is that LLMs that have access to this unstructured distributed data can more efficiently and effectively identify vulnerabilities through solutions such as vector databases.

## 2.6 CVSS

To assess and classify the vulnerabilities in this software, the Common Vulnerability Scoring System (CVSS) is used, which has established itself as a ubiquitous standard for quantifying vulnerabilities[14]. CVSS is an open framework developed by the Forum of Incident Response and Security Teams (FIRST), an international forum for incident response and cybersecurity teams, which published the first version of the CVSS standard in 2005. The CVSS score is measured on a scale of 0 to 10, with 10 being the highest level of risk. The calculated value can be transformed into qualitative categories (like LOW, MIDDLE, HIGH, and CRITICAL) to aid organizations in appropriately designing their vulnerability assessment and prioritization processes [15]. This classification into different qualitative representations can be categorized as shown in Table 3.

Table 3: CVSS Score and Qualitative Rating

CVSS Score	Qualitative Rating
0.0	None
0.1 – 3.9	LOW
4.0 – 6.9	MEDIUM
7.0 – 8.9	HIGH
9.0 – 10.0	CRITICAL

Various metrics are used to calculate one of three scores. The base score, the temporal score, and the environmental score can be calculated. CVE entries, for example, use the base score to evaluate individual CVE records. As the focus of this paper is partly on evaluating the security of a CVE, only the base score is considered in more detail. The possible presence of exploits is used later in the 3.3 section as additional information for entering the calculation and prediction by LLMs. According to the official CVSS version 3.1 specification [16], base scores are generally



generated by the organization responsible for the vulnerable product or by an authorized third party. Often, only the base metrics are released since they do not change over time and are relevant to all environments.

The metrics group for the base score is made up of eight different metrics (see table 4)

Table 4: CVSS Base Metrics Summary

Metric Group	Metric Value	Description
<b>Attack Vector (AV)</b>	Network (N)	The vulnerability can be exploited from anywhere across the Network.
	Adjacent (A)	The attack must be launched from the same local Network or a logically adjacent network.
	Local (L)	The attack requires local access or user interaction to succeed.
	Physical (P)	The attacker needs to physically access or manipulate the vulnerable component.
<b>Attack Complexity (AC)</b>	Low (L)	The attack does not require special conditions and has a high likelihood of success.
	High (H)	The attack relies on conditions outside the attacker's control, making it more difficult to exploit.
<b>Privileges Required (PR)</b>	None (N)	No privileges are required for exploitation.
	Low (L)	The attacker needs privileges that grant basic user capabilities.
	High (H) and significant privileges are required for exploitation.	
<b>User Interaction (UI)</b>	None (N)	No interaction from any user is required.
	Required (R)	Requires the user to perform some action before exploitation.
<b>Scope (S)</b>	Unchanged (U)	The exploit affects only the vulnerable component.
	Changed (C)	The exploit affects resources beyond the initial component.
<b>Confidentiality (C)</b>	High (H)	Total loss of confidentiality, leading to all resources being divulged.

Metric Group	Metric Value	Description
<b>Integrity (I)</b>	Low (L)	Partial disclosure of restricted information; not all data is disclosed.
	None (N)	No impact on confidentiality.
	High (H)	Complete loss of integrity; the attacker can alter all protected files.
<b>Availability (A)</b>	Low (L)	Partial modification is possible but does not impact the component critically.
	None (N)	No impact on integrity.
	High (H)	Complete loss of availability; the component becomes completely inaccessible.
	Low (L)	Performance degradation or interruptions occur but do not completely deny service.
	None (N)	No impact on availability.

In summary, FIRST has specified a vector that summarizes the classification of the metrics in the following format: 'CVSS:3.1/AV:N/AC:L/PR:H/UI:N/S:U/C:H/I:H/A:H'.

According to Jiamou Sun et al., [53], the mere description of a CVE record can already be used to make a suitable assessment of the classification into these metrics in order to calculate a CVSS score. To make the calculation more straightforward to understand, the following example illustrates the calculation of a CVSS score. FIRST defines the following formula for calculating the base score version 3.1:

The CVSS v3.1 Base Score is calculated using the following formula [16]:

$$\text{Base Score} = \text{Round Up} (\text{Minimum} [(\text{Impact} + \text{Exploitability}), 10]) \quad (1)$$

Where:

- **Impact** is calculated as:

$$\text{Impact} = 1 - (1 - \text{Conf}) \times (1 - \text{Integ}) \times (1 - \text{Avail}) \quad (2)$$

Here, **Conf**, **Integ**, and **Avail** are the impacts on confidentiality, integrity, and availability, respectively.

- **Exploitability** is calculated as:

$$\text{Exploitability} = 8.22 \times \text{AV} \times \text{AC} \times \text{PR} \times \text{UI} \quad (3)$$

Each of these factors (**AV**, **AC**, **PR**, **UI**) are weights based on the Attack Vector, Attack Complexity, Privileges Required, and User Interaction metrics.

- If the **Impact** sub-score is 0 (meaning no impact), the overall Base Score is also 0.

$$\text{If Impact} = 0, \text{ then Base Score} = 0 \quad (4)$$

Each metric value of a metric has a defined numerical value that is used for the calculation [16]."

**Example 2.2 (CVE-2022-37878). Description:** "Vulnerabilities in the ClearPass Policy Manager web-based management interface allow remote authenticated users to run arbitrary commands on the underlying host. A successful exploit could allow an attacker to execute arbitrary commands as root on the underlying operating system, leading to complete system compromise in Aruba ClearPass Policy Manager version(s): 6.10.x: 6.10.6 and below; 6.9.x: 6.9.11 and below. Aruba has released upgrades for Aruba ClearPass Policy Manager that address these security vulnerabilities."

#### CVSS Vector:

CVSS:3.1/AV:N/AC:L/PR:H/UI:N/S:U/C:H/I:H/A:H

#### Score 7.2 HIGH

- **Attack Vector (AV:N):** The vulnerability can be exploited over the Network, indicating a web-based management interface. Therefore, the attack vector is classified as Network (*Network*).
- **Attack Complexity (AC:L):** The attack complexity is low (*Low*), suggesting that no special conditions or hard-to-reach states are necessary for a successful attack. The attacker only needs to be authenticated.
- **Privileges Required (PR:H):** The attacker needs high privileges (*High*) in the system to exploit the vulnerability. This reflects the need to access the management interface as an authenticated user.
- **User Interaction (UI:N):** No interaction from another user is required (*None*) to exploit the vulnerability. The exploitation occurs completely through authorized user sessions.
- **Scope (S:U):** The Scope (*Scope*) of the vulnerability remains unchanged (*Unchanged*), as the vulnerability does not affect components outside the original security context.
- **Impact on Confidentiality, Integrity, and Availability (C:H/I:H/A:H):** The Impact on confidentiality, integrity, and availability is high (*High*). This is because

a successful attack gives the attacker full control over the system, leading to a complete loss of confidentiality and integrity, as well as availability.

Given the metric values:

Metric	Value
AV	Network (0.85)
AC	Low (0.77)
PR	High (0.27) for unchanged Scope
UI	None (0.85)
C, I, A	High (0.56 each)

Table 5: Metric Values

The calculated values are:

Calculated Value	Result
Impact	5.87
Exploitability	1.23

Table 6: Calculated Values

The formula for the Base Score is:

$$\text{Base Score} = \text{Round Up} (\min [( \text{Impact} + \text{Exploitability} ) , 10])$$

Applying the calculated values:

$$\text{Base Score} = \text{Round Up} (\min [5.87 + 1.23, 10]) = \text{Round Up}(7.1) = 7.2$$

Thus, the CVSS Base Score is **7.2**. This score can be interpreted with Table 3 to classify the qualitative rating as HIGH.

In this work, the CVSS score is relevant insofar as it describes a representative statement about software classified as critical. As a rule, a CVE record (see 2.5) describes the vulnerability that exists within the software but not the size of the impact of the vulnerability. To close this gap, this thesis will investigate whether LLMs are able to assess the severity of the impact through the text-based description within a CVE record. Research by Mustafizur R. Shahid and Hervé Debar [51] has already shown how a trained BERT model 2.3 can be very effectively adapted to the specific task through targeted fine-tuning and achieve high accuracy in predicting CVSS metrics based on vulnerability descriptions. [51] Building on this research, the CVSS score and metrics will also be extracted from the descriptions and tested

against different models (see 3.3) to provide a detailed statement on the measurement of efficiency and effectiveness in the course of vulnerability detection in this research area.

The latest standard is version **4.0** (as of April **2024**). In this work, however, the CVSS standard **3.1** is used. The reason for this is that during the investigation of the CVSS score, a comparison was developed using a Python script. The aim was to find out which CVSS version is currently used the most within the CVE database in order to work with it in the course of this work.

The comparison of all existing CVE entries as of April **2024** was carried out, and it was checked whether CVSS metrics were noted in the CVEs. The result of this (see table 7) shows that currently **69.39%** of all CVE entries that contain a score evaluation use version **3.1**. It is striking that version **4.0** was already released in **2023** but has so far had no apparent use on the creation of CVE entries from MITRE's official CVE database.

CVSS Version	Percentage
CVSS 3.1	69.39%
CVSS 3.0	26.14%
CVSS 2.0	4.55%

Table 7: Percentage of CVSS Versions

It was remarkable that in MITRE's official CVE database, **20.06%** of all CVE entries had an entry of one or more CVSS scores. That was **47,813** out of a total of **238,151** CVE entries. A comparison with the NVD, for example, shows that a CVE record with a base score of **7.5**, i.e. **HIGH**, was rated in the NVD database (**CVE-2020-23872**) and this has still not been added to the official CVE database after **3** years since the last update to the entry in the NVD database in **2021**. This confirms the discrepancy between the different sources in the synchronization of exploits in the assignment to CVE entries, which was also found by Jiamou Sun et al. [53]. Here, too, it was found that there is a gap between the timeliness of the official CVE database and the CVSS metrics recorded for it. This paper does not go into the size of the discrepancy or the comparison of the NVD database in any depth, but it is a research point that is still open for more in-depth analysis.

## 2.7 Exploit DB

Another data source included in this work is the database provided by ExploitDB. The entries within ExploitDB differ from the CVE entries in that they register and describe existing exploits for specific vulnerabilities. According to Jiamou Sun et al., [53], **69%** of the entries that were created as CVE entries and initially announced in

ExploitDB have a HIGH or CRITICAL severity rating (see chapter 2.6).

This can be explained by the fact that the actual existence of a confirmed exploit makes the vulnerability significantly more dangerous. In addition, it often takes a year or more for vulnerabilities for which exploits exist to appear in the official CVE database. More specifically, Jiamou Sun et al. [53] found that at the time of writing their paper, **39.6%** of all exploits had no associated CVE entry. **94.8%** of these exploits were already older than one year since their publication at that time. Based on these findings, in addition to the CVE entries, all ExploitDB records were used as a data set in this paper to extend the RAG system and investigate whether it can close the gap between missing CVE entries and the detection of vulnerabilities in software.

ExploitDB makes all exploit descriptions and papers on the exploits publicly available in its GitLab repository<sup>15</sup>. The data used in this paper refers to the CSV file with the descriptions of the exploits. Furthermore, the ExploitDB also provides the code for the exploits directly. For ethical reasons, these were not used in this work, but only the data available in the CSV file and described in table 8.

Table 8: Description of Exploit Database Fields

Field	Description
<b>id</b>	The unique identifier for each exploit entry.
<b>file</b>	The path to the file containing the exploit code or text.
<b>description</b>	A brief summary of the exploit or vulnerability.
<b>date_published</b>	The date when the exploit was published.
<b>author</b>	The name of the person or entity that published the exploit.
<b>type</b>	The type of exploit, e.g., denial of service (dos), web apps, etc.
<b>platform</b>	The platform that the exploit targets, e.g., Windows, Unix, Linux, etc.
<b>port</b>	The network port that the exploit targets, if applicable.
<b>date_added</b>	The date when the exploit was added to the database.
<b>date_updated</b>	The last date when the exploit entry was updated.
<b>verified</b>	A flag indicating whether the exploit has been verified.
<b>codes</b>	Associated codes such as CVE or OSVDB identifiers.
<b>tags</b>	Keywords or tags associated with the exploit.
<b>aliases</b>	Alternative names or identifiers for the exploit.
<b>screenshot_url</b>	URL to a screenshot related to the exploit, if available.

*Continued on next page*

<sup>15</sup><https://gitlab.com/exploit-database/exploitdb>

Table 8 – *Continued from previous page*

Field	Description
<b>application_url</b>	URL to the application or service that is affected by the exploit.
<b>source_url</b>	URL to the original source of the exploit information.

The further use and processing of the data from the ExploitDB are explained in more detail from the chapters in the experiments section 3.3

## 2.8 Nmap

The Network Mapper (Nmap) is an open-source tool used for scanning networks and gathering information about the systems and services present in a network. Orebaugh and Pinkard [45] describe network scanning as the process of discovering active hosts on a network and gathering detailed information about these hosts, such as their operating system, active ports, services, and applications [45]. Network scanning consists of the following four basic techniques according to Orebaugh and Pinkard [45]:

1. Network Mapping: Sending messages to a host to generate a response if the host is active.
2. Port Scanning: Sending messages to a specific port to determine if it is active.
3. Service and Version Detection: Sending specially crafted messages to active ports to generate a response indicating the type and version of the service running.
4. OS Detection: Sending specially crafted messages to an active host to generate a response that indicates the type of operating system running on the host.

Nmap offers various output formats and different scanning depths, which can be set using parameters when invoking the Nmap program. For the preliminary phase, the following Nmap parameters were used:

```
Nmap -A 192.168.178.107 --version-all
```

This command instructs Nmap to enable the detection of the OS, the version, script scanning, and the traceroute. A typical output of this command could look as follows:

Listing 4: Nmap scan on 192.168.178.107

```
1
2      PORT      STATE      SERVICE      VERSION
3      21/tcp    open       ftp          Pure-FTPd
4      25/tcp    filtered   smtp
5      63/tcp    filtered   domain
6      80/tcp    open       http         Apache httpd 2.4.10
7      |_http-server-header: Apache/2.4.10 (Debian)
8      |_http-title: 403 Forbidden
```

The output displays the ports open on the target server, the running services on those ports, and their versions. Some software providers attempt to hide version information. However, by reading HTTP headers, as enabled by Nmap, the version number can often still be revealed.

For the initial phase and the handling of Nmap outputs, the original Nmap output was used unaltered as an input parameter. The only modification was to manually split the scans based on each discovered port during the preliminary phase since some Nmap scans can exceed the maximum length of 4096 tokens. Tokens in GPT-3.5 are the basic units of text processing, similar to words or punctuation, and the limit of 4096 tokens is set due to memory and computational constraints to ensure efficient processing. Beyond this threshold, the resource requirements and complexity of the model's attention mechanism would increase exponentially. For the subsequent master's thesis, a Python script will be used to segment the scan results better and use them automatically as input parameters for the models.



## 3 Method

This chapter sets out the research question and the hypotheses. Experiments are used to provide empirical answers to the research questions. The design of the experiments from section 3.3, as well as the detailed procedure of the experiments, are described here. Detailed descriptions are used to define the databases selected here. Section 3.1 describes in detail the process of creating, selecting, and using the data sets specified for the experiments. The analysis in the summary sections evaluates the results obtained from the experiments after their execution. In order to test the most efficient LLM against a realistic scenario, it is evaluated in a simulation environment, and the research questions are answered.

### 3.1 Datasources

In order to define a database for the experiments to answer the research questions, data sources must be consulted that provide information on existing vulnerabilities and make them available in a categorized form. During the research, it looked in detail at two globally established standards for identifying and classifying vulnerabilities (see 2.5 and 2.6), as well as another comprehensive collection of collected use cases that security researchers use to improve the practical applicability of these standards in cybersecurity.

The selection of LLMs used for the experiments is also explained in this section. The focus here is on current LLMs that have already been generally trained and can be used immediately. The aim of this work is not to train a new model from scratch. Even though this is also a very research-intensive area in the field of cybersecurity improvement, this work aims to test how standard models can be used in vulnerability detection, both in the initial state and in the optimized state (see Section 2.1.1. The creation of data sets adapted for the execution of the experiments is described using Python scripts from section 3.3 onwards.

### 3.2 Models and technologies

#### 3.2.1 Models

Various models were used for the experiments to evaluate their efficiency against each other, particularly with regard to the detection of weak points. Different approaches were chosen to evaluate efficiency. The selection of the specified models was broad to obtain a comprehensive overview. All models are off-the-shelf LLMs that have already been pre-trained and are currently established on the market as

the most widely used and most capable. The models are tested in the initial state in the experiments, and selected models are enriched more precisely by using Retrieval Augmented Generation (RAG) together with embedding models. All models are consumed as a cloud service via REST APIs. The tests with the embeddings use both REST APIs and locally hosted RAGs that use the corresponding embedding models via REST APIs. The models are later used via Python scripts. By using a variety of models, the experiment can better reflect the complexity and versatility of actual use cases and thus provide more robust, generalizable findings. This helps to understand the areas of application of the individual methods and to provide an evidence-based statement about the LLM to be used further.

When selecting the models, the two models, GPT-3 and GPT-4, that were in widespread use at the time this paper was written, were chosen. OpenAI published the GPT-4 Omni model during the preparation of this thesis. This was not used for this work but should not be neglected for cost reasons since, as can be seen in the chapters of the experiment, a cost factor has an influence here. With the GPT-4 Omni model, the costs were reduced by 50% for the same performance.

Three different models were evaluated in advance for embedding the input data for the vector index within the RAG (see 2.1.1). The orientation towards OpenAI was based on the preliminary phase of the research. Due to the availability of the embedding models in Azure Open AI and the fact that the experiments and the structure of the RAG were to be standardized, i.e., with the same primary conditions, the embedding models were selected from those listed in the table 9. During the research, other models were examined in the literature, such as the Jina Embeddings 2 model, which was developed by Guenther et al. [20] for the BERT model. In work by Guenther et al. [20], the Jina models were compared with the text embedding Ada-002 model. The Jina model was able to achieve an average performance of 60.37%, whereas the text embedding Ada-002 model was able to achieve a better performance of 60.99%. However, the actual decision to select the embedding models was again influenced by the availability of the technologies used in this work.

For the models in Table 9, the newer *text-embedding-3* models should be used first, as the *text-embedding-3 large* model in particular achieved a value of 64.6% in the MTEB benchmark. The Massive Text Embedding Benchmark (MTEB), according to Muenninghoff et al. [40], aims to clarify how models perform across different embedding tasks and serve as a starting point for identifying universal text embeddings suitable for various tasks.

However, when working with the LlamaIndex, it has not been possible to build a vector index over the text embedding 3 model after several attempts. Therefore, the text embedding Ada 002 model was chosen for the following experiments.

The selection of models was also based on the pre-trained models from OpenAI.

Table 9: OpenAI Embedding Models <https://platform.openai.com/docs/guides/embeddings/embedding-models>

Model	Pages per Dollar	Performance on MTEB Eval	Max Input
text-embedding-3-small	62,500	62.3%	8191
text-embedding-3-large	9,615	64.6%	8191
text-embedding-ada-002	12,500	61.0%	8191

This is also because the experiments conducted with a vector index hosted by Azure are limited to OpenAI's models. In addition, it was observed from various benchmark analyses, such as that of Wang et al. [57], that the GPT-4-Turbo model, in particular, outperforms other current models such as Claude-3-Opus. Wang et al. [57] were able to achieve better results in the User-Centric Benchmark (URS) in 6 out of 7 tasks and thus also in the overall evaluation than the other models examined in the study. The URS benchmark consists of tasks from the following seven areas:

- Solve Problem
- Factual QA
- Text Assistant
- Ask for Advice
- Seek Creativity
- Leisure
- through API

In order to create diversity within the experiments, the Claude Opus Model 3 was initially to be used, but this is not yet available in Germany. Therefore, the Mistral Large model was chosen for another model, which, according to Mistral<sup>16</sup> has a similar performance to the GPT-4 model and is said to be more powerful than the Claude 2 model. The Claude Opus 3 model was not included in this review.

### 3.2.2 RAG

Two different frameworks for implementing a RAG were consulted for this work. On the one hand, Azure, the Microsoft cloud platform, was selected to pursue a cloud-based approach, and on the other hand, LlamaIndex was chosen as an on-

<sup>16</sup><https://mistral.ai/news/mistral-large/>

premise solution. The purpose of this selection was to perform a cost and performance analysis across different methodological approaches. In addition to the two selected frameworks, others currently exist that are suitable for implementing a RAG. It was not the aim of this research to comprehensively evaluate the various providers. During the exploratory phase of the research, work was carried out, particularly with the frameworks mentioned.

**RAG with Azure OpenAI** Microsoft has formed a partnership with OpenAI to accelerate the advancement of AI and make the resulting benefits accessible globally. This partnership, which has already been consolidated through investments in 2019 and 2021, continues to focus on the domains of supercomputing and AI research. As the exclusive cloud provider for OpenAI, Microsoft hosts all OpenAI workloads on the Azure cloud. To this end, Microsoft integrates the AI services and models developed by OpenAI into its consumer and enterprise products [30].

As part of this partnership, Microsoft is offering all models provided by OpenAI as an Azure service in a pay-as-you-go model as part of Azure AI Services, with consumption billed on a token basis. These include the Advanced Language and Legacy Language models, code interpreters, the OpenAI base models, image models, and embedding models, which are made available to end users [32].

Microsoft's AI-powered information retrieval platform, Azure AI Search, is used to construct the underlying RAG. According to [36], Azure AI Search enables developers to create rich search experiences and generative AI applications that combine large-scale language models with enterprise data. For this work, the Vector Store is used to create an index. When embedding the data, configurations can be used to select which embedding model should be used and whether a semantic search (see 2.1.1) should be performed for the queries on the RAG or whether a hybrid search should take place. A semantic search is carried out with a keyword search.

The workflow for indexing and querying for the vector search is implemented by Microsoft, as shown in Figure 4. For the vector search, Microsoft integrates the k-Nearest Neighbors (KNN) (see 2.1.1) and the Hierarchical Navigable Small World (HNSW) algorithm, a leading algorithm for Approximate Nearest Neighbors (ANN) ([34] [33]).

To set up a RAG system within the Azure Cloud, several conditions must be met. Firstly, a Microsoft Entra Tenant is required to be able to work with the Azure Cloud. An Entra Tenant represents an organization according to [37]. Each Microsoft Entra Tenant is independent and separate from other Microsoft Entra Tenants.

In order to be able to use the Azure OpenAI services, which are not available to all users by default, an activation must be requested from Microsoft (as of March

2024) so that these services are activated on a subscription (a unit within the Entra Tenant in which resources are provided) within the Tenant.

Azure offers several options for deploying resources in the Azure Cloud. These include manual creation via the Azure Portal<sup>17</sup> and provisioning via Infrastructure as Code (IaC). IaC uses the DevOp methodology and versioning with a descriptive model to define and deploy infrastructure such as networks, virtual machines, load balancers and connection topologies( [38]) via tools such as Bicep from Microsoft<sup>18</sup> or Terraform from HashiCorp<sup>19</sup> or also via the Azure Command-Line Interface (Azure CLI). For the reenactment of this work, it is explained here in detail how a RAG can be created in Azure via the Azure CLI.

To set up a RAG system in the Azure Cloud, basic structures must first be set up. The process includes creating a resource group, setting up an Azure OpenAI resource, and deploying the models required for the experiments to be performed (see 3.3 and 3.5). The required models include "GPT-4", "GPT-3.5" and the OpenAI embedding model "text-embedding-ada-002". In addition, an Azure AI Search must be set up in which the vector database is located to enable effective search functions.

The data required for vectorization comes from the sources mentioned in the sections 2.5 and 2.7 and is stored in Azure Storage. Azure Storage offers a highly available, massively scalable, permanent, and secure storage space for a large number of data objects in the cloud [29].

In Listing 5, the required resource group is created via the Azure CLI in the East US region. This resource group forms the basis on which all subsequent resources are created. Creating and using a resource group is cost-neutral in Azure, as there are no monthly fees for a resource group.

Listing 5: Create a new resource group in the East US region.

```
1 # Creates a new resource group in the East US region
2 az group create --name RAGRG --location eastus
```

The Azure OpenAI service is created in Listing 6. This service makes it possible to deploy models within the service and access the service via a REST API. The resource is provided within the previously created resource group. The Stock Keeping Unit (SKU) currently (April 2024) only exists in one version, as the costs for using the API for the individual model calls are billed via tokens. For the GPT-4 model, for example, the costs (as of April 2024) for input are 0.03\$ per 1000 tokens and 0.06\$ for output per 1000 tokens.

Listing 6: Create a Cognitive Services account.

<sup>17</sup><https://portal.azure.com/>

<sup>18</sup><https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/overview?tabs=bicep>

<sup>19</sup><https://www.terraform.io/>

```

1  # Creates an Azure OpenAI Search account in the previously created resource group
2  az cognitive services account create \
3  --name RAGOpenAIService \
4  --resource-group RAGRG \
5  --location east us \
6  --kind OpenAI \
7  --sku s0 \
8  --subscription <subscriptionID>

```

To communicate with the REST API, the endpoint provided by the Azure AI Search resource is required. After executing Listing 7, the endpoint is provided in the following form: <https://RAGOpenAIService.openai.azure.com>. This endpoint is the URL through which requests are sent to the provided models to perform various operations, such as retrieving responses or performing calculations based on the requested data.

Listing 7: Retrieve the endpoint URL of the Azure OpenAI Service.

```

1  # Retrieves the endpoint URL of the Azure OpenAI Service
2  az cognitive services account shows \
3  --name RAGOpenAIService \
4  --resource-group RAGRG \
5  | jq -r .properties.endpoint

```

To ensure that only authorized users have access to the REST endpoints, this is secured with an API key. This API key must be specified together with the endpoint during the calls. Listing 8 maps the Azure CLI command to retrieve the API key. This step is crucial to ensure the security and integrity of the data and services that are made accessible via the REST API.

Listing 8: Retrieve the access key for the Azure OpenAI Service account.

```

1  # Retrieves the primary access key for the Azure OpenAI Service account
2  az cognitive services account keys list \
3  --name RAGOpenAIService \
4  --resource-group RAGRG \
5  | jq -r .key1

```

For the later experiments from the chapters 3.3,3.5 and 3.6 an embedding model as well as the GPT-4 and GPT-3.5 model is required. Listing 9 shows the deployment for the embedding model "text-embedding-ada-002". The same step is carried out for the "GPT-4" model in the "GPT-4-turbo" version and "GPT-3.5" in the "GPT-3.5-Turbo-0125" version. No further costs are charged by Microsoft here.

Listing 9: Deploy a model in Azure OpenAI Service.

```

1  # Deploys a model in Azure OpenAI Service
2  az cognitive services account deployment create \
3  --name RAGOpenAIService \
4  --resource-group RAGRG \
5  --deployment-name ada002 \
6  --model-name text-embedding-ada-002 \
7  --model-version "1" \
8  --model-format OpenAI \
9  --sku-capacity "1" \
10 --sku-name "Standard"

```

For the later experiments in this work, specific vector databases are created for the use cases, and the creation of the RAGs required for the respective use cases is discussed in detail. These are created within the Azure AI Search from Listing 10. The service is set up in the same resource group as the previous services. The partition and replica settings were configured in a cost-saving manner and created in the Basic SKU with one partition and one replica. The partition describes the physical memory and the replica of the number of instances of the service. In addition to the Azure AI Search Service, the selected Basic SKU offers 15 GB of storage and costs \$73.73 per month (as of April 2024).

Listing 10: Azure AI Search

```
1 az search service create \  
2 --name RAGAIsearch \  
3 --resource-group RAGRG \  
4 --sku Basic \  
5 --partition-count 1 \  
6 --replica-count 1
```

Specific Azure configurations are made for further implementation in the experiments. These are explained in detail in the corresponding chapters. To create a basis for the storage of data, into which the data sources are later uploaded, an Azure Blob Storage is created at the end of this chapter, as mentioned at the beginning (see Listing 11).

### Listing 11: Azure Blob Storage

```
1 az storage account create \  
2 --name RAGBLOB \  
3 --resource-group RAGRG \  
4 --location east us \  
5 --sku Standard_ZRS \  
6 --encryption-services blob
```

In the further course, the RAG can now be set up with this configuration depending on the use case from the chapters 3.3 and 3.5.

**RAG with LlamaIndex** The **LlamaIndex** is an open-source framework for context-augmented, domain-independent data augmentation for text classification tasks and applications of LLMs. The LlamaIndex was originally developed by Meta (formerly Facebook). This context is used to leverage private or domain-specific data for specific use cases, such as Q&A bots or understanding documents and extracting information<sup>20</sup>. The LlamaIndex framework supports the RAG pattern, as shown in Figure 4. The LlamaIndex provides the framework for flexible connections to various providers. By default, LlamaIndex offers the option of embedding local data in a vector store and then working with a selected model on the data.

In addition, certain connectors or entire data sets can be obtained via the **LlamaHub**<sup>21</sup>. In this paper, we work with a self-created data set and with the "SimpleDirectoryReader" connector, which allows access to the contents of a local folder. There are also connectors that can be used to read PDFs, for example, or to access various cloud storage services as a data source.

LlamaIndex uses this local data to create a vector index. To do this, the documents are vectorized using an embedding model and stored in a corresponding vector storage. The vector storage can be either directly available as an in-memory solution; the vector index is loaded into the computer's main memory, and the queries are carried out directly on the in-memory storage, or the vector index can be stored in a cloud vector storage from a provider such as AWS, Qdrant, Postgres or MongoDB. In this work, the Simple Vector Store is used, which is offered directly by LlamaIndex and stored in memory. This enables the flexible and easy use of a vector index, but it is more suitable for experiments than for productive applications. In addition to the cloud and direct in-memory solutions, there are also other providers, such as Apache Cassandra, which offer a self-hosted solution. In contrast to Azure AI Search (see 3.2.2), this approach via the Simple Vector Store is not as complex as using Azure AI Search, where more advanced semantic search techniques such as HNSW are not possible. Therefore, it is assumed that the results in the following experiments will be more powerful and show better performance for the models

<sup>20</sup><https://docs.llamaindex.ai/en/stable/\#llamacloud>

<sup>21</sup><https://llamahub.ai/>



that use the Azure AI Search approach. However, it should be found out whether a RAG pattern using the LlamaIndex also provides satisfactory results. A distinction of the required time as well as the size of the respective indices is carried out in chapter 3.4.3.

To use the LlamaIndex locally, a Python script was developed for this work, which is described in more detail in Chapter 3.5.2. Files are loaded from a folder on the local storage and vectorized using the Text-Embedding-Ada-002 model. After vectorization, the vector index is generated by the LlamaIndex in the form of a JSON file and stored on the hard disk. Each time the LlamaIndex is used to work on the vector index, the content of the JSON file is loaded from the LlamaIndex into the in-memory and can then be queried via the LlamaIndex.

### 3.3 Experimental Design

To develop a thorough understanding of the methodology to be applied in this masters thesis, the plan is to conduct preliminary experiments during the preparatory phase of the work. The aim of these initial experiments is to investigate various approaches, which will later facilitate a comprehensive evaluation of different LLMs and tools used for vulnerability identification. From these preliminary studies, I expect to derive insights into how diverse datasets can be utilized for training the models or how a specialized database might be implemented within the scope of the master's thesis.

This includes understanding how the models interact with these datasets and how they can be trained to make robust predictions. Additionally, I intend to explore the necessary processing of input data for the model to handle specific types of inputs. An example of such specific inputs is output reports from Nmap scans, a popular network scanning tool. This is crucial to comprehend how the model needs to be adjusted to work effectively with real-world cybersecurity data. It is essential to emphasize that this preliminary work is being conducted on a limited scale, relying on a restricted number of data sources and datasets. This limitation is chosen to manage the effort and to test specific aspects of the methodology before applying it on a larger scale in the master's thesis.

Another focus of this phase is to compare the scan results of port data generated using the vulnerability tool Nessus with the findings from the models. This includes examining the running software identified by the models, their version numbers, and potential vulnerabilities. Through this comparison, I aim to gain an initial assessment of the accuracy and utility of the model predictions in comparison to established security tools.

## **3.4 Experiment 1**

### **3.4.1 Experimental Objectives and Hypotheses**

In this experiment, the aim is to determine which LLM (see 3.2.1) is best suited, in which constellation, and with which extension to recognize, identify, and classify the efficiency in the correct identification of an affected and known vulnerability. In addition, a cost comparison is carried out based on the tokens used.

The aim is to investigate in detail how effective LLMs are in identifying a specific known vulnerability that has already been classified in MITRE's CVE database. It analyzes how well LLMs with a RAG 2.1.1 can correctly classify corresponding vulnerabilities in contrast to pre-trained LLMs. In the course of preparing for the experiment, challenges arose that extended the experiment beyond the originally planned pure check for the same software and version number. As already described in 2.5, there are recognized deficits in the general normalization and uniformity of descriptions and the up-to-dateness between different vulnerability databases. As a result, the experiment should also test the generalization capability and context understanding of the different LLMs. The assumption here is that, in contrast to static vulnerability detection tools or pure database queries, LLMs also recognize correlations within the vulnerability description. The focus here is particularly on correlations in software versions.

The hypothesis is that LLMs are able to recognize vulnerabilities based on the software and version if this is done as a pure parameter input in this experiment. By using NLP to better understand and utilize the context and semantics within the unstructured data, it is hypothesized that LLMs using a vector index will outperform pure pre-trained models. The hypothesis further postulates that LLMs can also correctly recognize vulnerabilities through text understanding if these are not explicitly noted in the existing vulnerability. For example, the detection of a vulnerability even if the version is not explicitly stated.

### **3.4.2 Methodology**

The methodology of this experiment is designed to evaluate and compare the accuracy and reliability of the responses generated by the different models. This is done through a combined automated procedure and manual checks to gain a deeper understanding of the performance of this technology in the cybersecurity application domain. In addition to efficiency, a cost analysis per tested LLM is performed, as this is a relevant variable when compared with static evaluation tools. The LLMs used are the same as those already described in 3.3.

The database for this experiment comprises the complete CVE database with all CVE records up to April 2024. A subset of 100 randomly selected CVE records from all the CVE records is used to test the LLMs against the various LLMs. The selection of 100 records was made for cost and verification reasons. As part of the check is carried out manually, the subset should not be too large. In addition, during the preliminary phase, it was determined that a single run with 100 datasets against the GPT-4 model using a vector index in the Azure cloud costs around 20\$. A more detailed cost calculation will be explained later. Due to the likely iterations during the experiment setup, the cost should be kept low. These 100 CVE records were used in the JSON 5.0 data format.

In order to obtain a suitable subset of the 100 CVE records, only CVE records with an entry in the "Version" field were selected to ensure that these data records have a version. As already explained in section 2.5, this is not always the case, as the data is very unstructured. This can be used to the advantage of this experiment, as there may be a case such as in 2.5 where a version is incorrectly entered in the "Version" field, but this is not affected at all, as the actual description excludes this version.

For this experiment, this would mean that if an LLM queries this version, it is assumed that no vulnerability is detected by the LLM, which would lead to a false positive, as the version specified in the "Version" field is used as a parameter. For example, version 1.8.0 would be used as the version, but the description says "...before 1.8.0". In this case, the LLM would not have to detect a vulnerability in the version, which would be correct according to the description. Furthermore, a criterion was set during filtering that the CVE record must be published. This is defined in a field within the CVE record. Furthermore, the selection was limited to software types that are usually frequently affected by vulnerabilities (see table 11), as well as if they are installed in an environment, require a port, and can thus be detected via Nmap scans (see section 2.8). The selection was made based on the context of this work, as this type of software type will be the focus of this work later on.

This experiment transfers the software and version parameters to all LLMs.

A prompt was engineered for the input to the corresponding model, which transfers the parameters "product" for Software and "versions" for Version in a prompt. The prompt in Listing 12 is designed in such a way that it explicitly asks for the parameters for a vulnerability. The prompt includes the return being in a specified JSON format.

Software	Description
nginx	Web server and reverse proxy
MySQL	Relational database management system
apache	Web server software
phpmyadmin	Web interface for MySQL database management
MariaDB	Relational database system, a fork of MySQL
WordPress	Content management system for websites
drupal	Content management framework
samba	Free Software for file and print sharing in networks
PostgreSQL	Extensive, object-relational database system
postfix	Mail transfer agent (MTA) for routing and delivering email
OpenSSH	Set of programs for secure communication over insecure networks
BIND	Widely used Domain Name System (DNS) server
Exim	Mail transfer agent (MTA) for Unix systems
Squid	Proxy server and web cache daemon
Dovecot	IMAP and POP3 server software
vsftpd	Very secure FTP daemon for UNIX-like systems
proftpd	Modular FTP server for Unix and Unix-like operating systems
OpenVPN	Open-source Software for creating virtual private networks (VPNs)
gitea	Compact software for code management with Git

Table 11: Brief Description of Software Types

#### Listing 12: Input Prompt for the Model

```

1  "content": (
2  f"Has this software: {product} {versions} susceptible to vulnerabilities? Please provide only the JSON
   data for each affected cve number in the following format:"
3  '{"Software": "<software_name>", "Version": "<version_number>", "IsAffected": "<true/false>" , "CVE-ID":
   "<cve_id>"}'
4  'No further information on the CVE from your side. Pay particular attention to whether the specified
   Version is within the version range of the affected Software.'
5  'Check whether the name and version are within the scope of the CVE.'
6  'If you do not find information about the software and version, give a single JSON response in that
   format: {"Software": "<software_name>," "Version": "<version_number>," "IsAffected": "false,"
   "CVE-ID": "None"}.'
7  )

```

If the model cannot generate a result, the LLM cannot declare the Software and Version as affected. In a context where the model is used to predict vulnerabilities and cannot return a result, a vulnerability that is not found is treated as non-existent. The prompt explicitly points to the model that special attention should be paid to understanding the versions.

The model returns the following parameters:

### Field Descriptions:

- **Software:** The Software is interpreted by the model according to the input parameters.
- **Version:** The Version of the Software as interpreted by the model according to the input parameters.
- **IsAffected:** Indicates whether the model has declared this Software as being affected by a vulnerability.
- **CVE-ID:** The CVE Record identified by the model.

The evaluation of the results is checked for matches manually and using a GPT-4 model. GPT-4 was chosen because, according to the Hughes Hallucination Evaluation Model (HHEM) leaderboard on Huggingface<sup>22</sup>, it showed the best results in avoiding hallucinations. Until April 2024, GPT-4-Turbo remains at the top of the leaderboard with a hallucination rate of 2.5%. Each conspicuous record is reviewed and checked against the detected CVE record during the manual review. The manual check was carried out carefully and compared with the original data sets of the CVE records published by MITRE.

The complete CVE database with all entries up to April 2024 was used for the pre-trained models, which also have a RAG approach. Beforehand, a Python script filtered the JSON files for relevant parameters. A CVE record, as in Listing 13, includes data fields that are irrelevant to this experiment and were therefore filtered out. When filtering, care was taken to ensure that only published CVE records (status = published) and CVE records containing a description, a CVE ID, and a CVSS vector string were included. The CVSS vector string is irrelevant for this experiment but is used in other experiments.

The implementation used a Python script, which extracts only the CVE ID, Description, vector string, and a CWE description from the relevant entries and creates one per CVE entry in a CSV file. Since the Azure AI Search has a restriction that a file may not contain more than 16 MB and no more than 65,000 characters<sup>23</sup>, which are to be vectorized, the Python script generated 1261 individual CSV files that have the format from Listing 13.

---

<sup>22</sup><https://huggingface.co/spaces/vectara/leaderboard>

<sup>23</sup><https://learn.microsoft.com/en-us/azure/search/search-limits-quotas-capacity>

### Listing 13: CSV Record for RAG

```

1 CVE-ID, Description, vectoring, CWE-Description
2 CVE-2024-22317, IBM App Connect Enterprise 11.0.0.1 through 11.0.0.24, and 12.0.1.0 through 12.0.11.0
   could allow a remote attacker to obtain sensitive information or cause a denial of service due to
   improper restriction of excessive authentication attempts. IBM X-Force ID: 279143
   .,CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:H,CWE-307 Improper Restriction of Excessive
3 Authentication Attempts
   CVE-2024-22318, "IBM i Access Client Solutions (ACS) 1.1.2 through 1.1.4 and 1.1.4.3 through 1.1.9.4 is
   vulnerable to NT LAN Manager (NTLM) hash disclosure by an attacker modifying UNC capable paths within
   ACS configuration files to point to a hostile server. If NTLM is enabled, the Windows operating
   system will try to authenticate using the current user's session. The hostile server could capture
   the NTLM hash information to obtain the user's credentials. IBM X-Force ID:
   279091.",CVSS:3.1/AV:L/AC:H/PR:N/UI:N/S:U/C:H/I:N/A:N,CWE-200 Exposure of Sensitive Information to an
   Unauthorized Actor

```

This approach ensures that only relevant data is used in a compressed manner and that superfluous and irrelevant data is excluded. For the Azure AI Search, these data sets were manually uploaded to an Azure Blob Storage based on section 2.1.1. First, a container, i.e., a storage area within the Azure Blob Storage, must be created to which the JSON files are uploaded. In addition to the initial creation of the infrastructure for the RAG (see section 3.2.2), a container is created for this experiment using the Azure CLI (see Listing 14). First, the executing user is granted access rights as a contributor, and then a container for the experiment is created within the Blob Storage. To create the vector index, the index was created via the Azure portal using the Azure AI Search resource with the embedding model text-embedding-ada-002.

### Listing 14: Example generated CVE Record

```

1 {
2   az ad signed-in-user show --query id -o tsv | az role assignment create \
3     --role "Storage Blob Data Contributor" \
4     --assignee @- \
5     --scope "/subscriptions/XXX/resourceGroups/RAGRG/providers/Microsoft.Storage/storage accounts/RAGBLOB."
6
7   az storage container create \
8     --account-name RAGBLOB \
9     --name expl \
10    --auth-mode login
11 }

```

To generate the vector that is used on-premise via the LlamaIndex, the same data sets were used to generate a Simple Vector Store (see section 2.1.1). A Python script was developed for this purpose, which also reads the CSV files via the embedding model text-embedding-ada-002 and writes them to a local vector for later use in the store. The libraries required for this were the official LlamaIndex libraries<sup>24</sup>.

The results and token effort are recorded and saved in various CSV files. For each LLM, a CSV file is created for the results, see Listing 16, as well as a CSV file for the tokens used, see Listing 15, in order to be able to take a closer look at the costs.

### Listing 15: CSV for token consumption

```

1 Total Time, Call Count, Right, False, Input Tokens, Prompt Tokens, Input Tokens RAG, Prompt Tokens RAG,
2 Cost Input Total, Cost Prompt Total, Cost Total
  3066.08 seconds,156,53,48,5317,14419,241,2244,0.05558,0.49989,0.55547

```

<sup>24</sup><https://pypi.org/project/llama-index-core>

### Listing 16: CSV for LLM results

```

1 Software, Version, CVE ID, Reason, Software Match, Version Match, IsAffected, VersionIsExactlyNamed,
  LLMTrue, LLMSaysIsAffected
2 Icegram Email Subscribers & Newsletters Plugin for WordPress,4.4.8, CVE-2020-5767: The version and
  software name match the dataset, and the Version is affected as described. true, true, accurate,
  True, true
3 Icegram Email Subscribers & Newsletters Plugin for WordPress,4.4.8, CVE-2020-5768, The Software and
  Version in the assertion exactly match the Software and Version described in the data set as being
  affected by CVE-2020-5768., true, accurate, true, True, true

```

The performance of the LLMs is evaluated using a confusion matrix (for more information, see section 3.4.4). In addition to categories such as Software and Version, a reason is documented in the "reason" field that explains why the statement of the LLM is rated as correct or incorrect. In addition, it is recorded whether the LLM has made an accurate statement regarding the impact of a vulnerability on the specified Software and Version. The specific values and their significance within the experiment are discussed in detail.

To check this data, a Python script was developed, which creates a confusion matrix based on the results obtained. LLMTrue and LLMSaysIsAffected values are extracted from the results list and compared to derive the metrics required to create the confusion matrix. This is discussed in detail in the following chapters. From the output parameters after the programmatic evaluation 16, including LLMSaysIsAffected (the model has declared the Software and Version as affected by vulnerabilities) and LLMTrue (the check has shown that the LLM is correct), the value IsAffected is calculated in the script for generating the Confusion Matrix. This indicates whether the Software is affected by the vulnerability identified by the LLM.

The following formula is used in the Python script to decide whether a vulnerability exists: It is assumed that the values LLMSaysIsAffected and LLMTrue have been set correctly. In this case, the formula validates whether the LLM declared this Version affected. If the check shows that the LLM is correctly located, the Version discovered vulnerability affects the Version. The formula returns "TRUE" if the system (LLMSaysIsAffected) correctly recognizes that a condition applies (LLMTrue).

$$\text{IsAffected} = \begin{cases} \text{"TRUE"} & \text{if } (\text{LLMSaysIsAffected} = \text{"TRUE"}) \wedge (\text{LLMTrue} = \text{"TRUE"}) \\ \text{"TRUE"} & \text{if } (\text{LLMSaysIsAffected} = \text{"FALSE"}) \wedge (\text{LLMTrue} = \text{"FALSE"}) \\ \text{"FALSE"} & \text{if } (\text{LLMSaysIsAffected} = \text{"TRUE"}) \wedge (\text{LLMTrue} = \text{"FALSE"}) \\ \text{"FALSE"} & \text{if } (\text{LLMSaysIsAffected} = \text{"FALSE"}) \wedge (\text{LLMTrue} = \text{"TRUE"}) \\ \text{"FALSE"} & \text{otherwise} \end{cases}$$

As explained above, the parameter LLMTrue is validated using a GPT-4 model. The output parameters from the output by the prompt in Listing 12 and the correct Description of the detected CVE vulnerability in a prompt 17 are used as input parameters. The correct Description of the vulnerability is loaded into the script via

the CVE ID, and the Description is extracted. The model is instructed to compare the two inputs to verify the consistency of the context. The model is also instructed to monitor version coherence in particular.

The generated output parameters are added in the CSV file described above. The parameters *Software Match*, *Version Match*, and *Reason* are used for this, as well as the parameter *AssertionIsTrue*, which represents the value *LLMTrue* and is used to collect metrics. This process is used to verify the accuracy of the data set. As already described in section 2.5, there are no defined structures for validating the correctness of a vulnerability. The data is often inconsistent and difficult to interpret. An exclusively algorithmic processing would require the consideration of all existing auxiliary words such as *before*, *after*, etc., as well as mathematical operators and the understanding of textual relationships. Since completing this task requires text comprehension supported by AI, the GPT-4 model was chosen for verification. In addition, each result list is inspected manually, which requires a secondary evaluation, especially for conspicuous entries. This methodical approach to verification aims to generate the metrics to be collected in later sections.

Listing 17: Input Prompt for the Model

```

1  messages=[
2  {
3      "role": "user",
4      "content": f"This is an assertion: {json_data}. This is the correct data set: {entry}. The entry
                    describes a Vulnerability. Is my assertion regarding whether the Version is affected correct? Check
                    whether the assertion matches the dataset. Pay particular attention to whether the versions match
                    and are within the correct version range. Respond only in this format; no additional information is
                    needed.: '{{'Version': '<version>', 'Software': '<software>', 'Version Match': '<true/false>',
                    'Software Match': '<true/false>', 'AssertionIsTrue': '<true/false>', 'Reason': '<reason>',
                    'VersionIsExactlyNamed': '<true/false>'}}'"
5  },
6  ],

```

### 3.4.3 Experiment Execution

The experiment used specific specifications, scripts, and technologies. The description of the required components is followed by a detailed description of how the experiment was carried out and the methods used to determine the required results.

The hardware specifications used for this experiment are listed in Table 3.4.3.

component	specification
<b>Operating system</b>	Windows 11
<b>Processor</b>	Intel Core i7-10700K, eight cores, 3.8 GHz
<b>Memory</b>	128 GB
<b>Development environment</b>	Visual Studio Code - Insider Version 1.89.0
<b>Python version</b>	3.9.2



Various Python scripts were developed for the experiment, listed in Table 12. These scripts execute specific logic to test the hypotheses formulated in section 3.4.1. Each script fulfills a separate task and is called manually in sequential order.

Table 12: Scripts and Descriptions

Script	Description
<b>createRAGFiles.py</b>	Filters and creates compressed CSV files from JSON CVE records.
<b>runExperiment1.py</b>	Runs the experiment, processes 100 selected CVE records, and interacts with corresponding LLMs depending on the configuration.
<b>createConfMatrix.py</b>	Calculates the Confusion Matrix based on the experiment results.
<b>createLlamaIndex.py</b>	Creates a SimpleVector Store via the LlamaIndex.

Table 13: Technologies, REST APIs, and Data Formats

Table 13: Technologies and Usage

Technology/Service	Usage
<b>Azure Blob Storage</b>	Data Storage for CVE Database
<b>Azure AI Search</b>	Vector Database
<b>Azure OpenAI</b>	LLM models and embedding models
<b>REST API</b>	Interaction with Llama 2 70B, Mistral, Azure, OpenAI
<b>Data Formats</b>	JSON, CSV

The experiment was implemented in systematically planned steps. According to the methodology described in detail in section 3.5.2, the vector databases on Azure and the LlamaIndex were initialized. The required datasets were prepared using the script "createRAGFiles.py" to iterate through the datasets and then segment them into CSV files using specific filter processes. Using the described process, these files were then vectorized within the Azure AI Search vector database.

The script "createLlamaIndex.py," which uses the OpenAI embedding model "test-embedding-ada-002", was used for vectorization using the LlamaIndex. The duration and memory requirements of these processes were documented in detail.

Creating the vector index on the specified hardware for the LlamaIndex took 39 minutes and resulted in a vector index of 994 MB. The input dataset comprised 1261 CSV files with a total volume of 73.8 MB. In contrast, creating the vector index via the Azure Cloud took 845 minutes and required 1610 MB of storage space.

After creating the vector indices, the experiment was initiated using the script "runExperiment1.py", listed in table 12. The execution of the experiment was automated and continuously monitored. During the run, adjustments had to be made to the script to correct errors during runtime and restart the process. Before each run, the script had to be adapted for a different LLM from the list used. The output files for the results and the token calculation were modified manually, and the method for the run was changed accordingly.

Within the experiment execution script, each LLM has its method. The input parameters of each method required to use the LLMs were the same for each model and included the software and its version. In addition, the CVE ID was passed for the described check. Each LLM used the same parameters configured when calling the REST API to control the model to the corresponding endpoints. The temperature, the maximum number of response tokens (to limit the size of the response), and the top-p value were always set to 0 to achieve more repetitive and deterministic responses and to 1 to work conservatively and select only the most probable tokens. These values were not considered with different settings in the experiment.

The described check was also performed using the individual methods used for the calls to the LLMs. It was stored in working memory along with the specified output parameters of the LLMs during execution. After the 100 CVE records were iterated, CSV files were created from the data held in working memory. It is possible that multiple entries were generated per software and version, as the model was able to identify potential vulnerabilities multiple times.

The process was automated as described and randomly checked by running the process in debug mode. This method made it possible to view the outputs and checks during processing. This process required multiple adjustments to the code to refine the input prompts described in section 3.5.2 and respond to any errors. As explained in the later cost estimate, the runs, particularly those via Azure AI Search, were closely monitored due to the costs involved in proceeding efficiently and aborting erroneous runs early.

The result of each run was a CSV file with the LLM's predictions and the checks already carried out during the run, as well as another CSV file with the generated prompts. Each time the LLM was called, the tokens were divided into input and output tokens: Input tokens for the inputs passed to the model and Output tokens for the responses generated by the model.

After the script run was completed, a manual check was carried out as described above to validate the correctness of the results list. This thorough check took about two hours per result list to match possible detected vulnerabilities with the authentic CVE records.

After validating the result lists, they were loaded as input parameters into the script `createConfMatrix.py`. This script calculates the following metrics based on the values of the confusion matrix:

- **TP** (True Positives)
- **FN** (False Negatives)
- **FP** (False Positives)
- **TN** (True Negatives)
- **Accuracy**
- **Precision**
- **Recall**
- **F1-Score**
- **Matthews Korrelationskoeffizient (MCC)**

The metrics Accuracy, Precision, Recall, F1-Score, and the MCC are determined using the values from the confusion matrix in the script and output via the console. The results were then noted manually for each LLM. The following chapter discusses in detail the values determined and the results, including a cost analysis.

### 3.4.4 Measurement and Evaluation

The performance of the different models is first measured using a confusion matrix per LLM. According to Mohak Shah and Nathalie Japkowicz [50], the Confusion Matrix is a central element for evaluating the performance of classification algorithms. It is represented by the matrix elements  $C_{ij}$ , where  $i$  and  $j$  are the indices for the actual class and the class predicted by the classifier. In this specific experiment, the Confusion Matrix is used to evaluate the performance of LLMs in predicting vulnerabilities in software. Different performance evaluations are performed. In the case of the experiment, the performance evaluation is done for binary classification. The binary classification is one of the most common classifications in evaluating algorithms [50]. Two classes are defined: negative and positive. This results in a confusion matrix with four components: true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN). TP and TN indicate the number of test set examples that were correctly classified as positive or negative, respectively. In

contrast, FN and FP represent the positive and negative examples incorrectly classified as negative and positive respectively [50].

For the binary classification in this experiment, the four key figures can be described as follows:

**TP):** The LLM correctly predicts the presence of a vulnerability in the software.

*example:* The LLM receives the input parameters: Software = "MySQL Server", Version = "8.0.17". The checked output is CVE-ID = CVE-2019-2997; ISAffected = True, and LLMTrue = True. In this case, the number of TPs is increased by one ( $TP = TP + 1$ ) as the correct vulnerability has been identified.

**FN:** The LLM incorrectly predicts the absence of a vulnerability, although there is a known vulnerability in the software.

*example:* The LLM receives the input parameters: Software = "MySQL Server", Version = "8.0.17". The checked output is CVE-ID = CVE-2019-2997, IsAffected = True, and LLMTrue = False. In this case, the number of FNs is increased by one ( $FN = FN + 1$ ) as an existing vulnerability was not detected.

**FP:** The LLM incorrectly predicts the presence of a vulnerability, although no known vulnerability exists in the software or another vulnerability is present.

*example:* The LLM receives the input parameters: Software = "MySQL Server", Version = "8.0.18". The checked output is CVE-ID = CVE-2019-2997; IsAffected = False and LLMTrue = False (for example, the description was up to version 8.0.17), in which case the number of FP is increased by one ( $FP = FP + 1$ ).

**TN:** The LLM correctly predicts the absence of a vulnerability, which is correct because no known vulnerability exists in the software.

*example:* The LLM receives the input parameters: Software = "MySQL Server", Version = "8.0.18". The checked output is CVE-ID = CVE-2019-2997; IsAffected = False and LLMTrue = True (e.g., the description states up to version 8.0.17). In this case, the number of TNs is increased by one ( $TN = TN + 1$ ).

The confusion matrix is the basis for evaluating other metrics such as precision, accuracy, recognition, and F1 score. The *accuracy* indicates the proportion of correct predictions overall ( $TP + TN$ ) about all cases.

The *accuracy* is calculated as follows:

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

*Precision* measures how accurate the model predictions are when determining vulnerabilities.

$$\text{precision} = \frac{TP}{TP + FP}$$

*Recall* indicates how well the model captures all vulnerabilities.

$$\text{Recall} = \frac{TP}{TP + FN}$$

The *F1 score* represents a harmonic mean of Precision and Recall and aims to achieve a balance between precision and recall [50].

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The above guidelines address two specific scenarios: the accuracy with which an output matches an existing vulnerability in a given software version and the accuracy with which an LLM identifies a vulnerability in a given software.

During the initial phase of the study, it was found that LLMs can tend to hallucinate (referenced in section 2.1). This tendency leads to a situation in which LLMs make correct statements that are not congruent with the other parameters. An example of this is that an LLM correctly claims the existence of a vulnerability in software; however, this statement is based on a random hit and not on a well-founded analysis. Consequently, in subsequent experiments, it is essential to implement a multicategorical classification in addition to the binary classification. This also evaluates whether an LLM states the presence of a vulnerability and correctly identifies and names the specific vulnerability. Although randomly correct predictions can potentially distort the assessment of model performance, the hypothesis for the evaluation is based on the fact that by analyzing various classes within the Confusion Matrix, it could be shown that models are also prone to hallucinations about other aspects. Evaluating the entire Confusion Matrix and the metrics above enables a more precise understanding of model performance.

As the classes may be unbalanced, an additional value is added to assess the quality of the binary classifier: the *Matthews Correlation Coefficient (MCC)*. According to Baldi et al., [3], the MCC is a robust metric that considers true and false positives and negatives and provides a value between -1 and +1. In this context, +1 indicates a perfect prediction, 0 signifies performance no better than random guessing, and -1 represents complete divergence between the prediction and the actual observation.

$$\text{MCC} = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

To calculate the costs, the tokens required for each run were logged. A distinction was made between the input tokens and the output tokens.

The current costs for running the experiment and for calling the model are defined in table 3.4.4.

Model	cost per 1000 tokens (input)	cost per 1000 tokens (output)
Mistral Large	\$0.004	\$0.012
GPT-4-Turbo-2024-04-09	\$0.01	\$0.03
GPT-3.5-Turbo-0125	\$0.0005	\$0.0015

The following formula was used to calculate the cost per run. Specific parameters

were considered, including the tokens generated to check the model responses and those used to detect vulnerabilities. For example, the formula below for the GPT-4-Turbo model results in the following example calculation: 100 tokens for the verification input and 1000 tokens for the vulnerability check input, resulting in a requirement of 1100 tokens. These are then divided by the number 1000 (as the price is calculated per 1000 tokens) and multiplied by 0.01\$, resulting in a cost of 0.011\$.

In addition, in this example, the calculation of 300 tokens for the output of the check and 3000 tokens for the output of the vulnerability detection would result in a total of 3300 tokens. This is again divided by 1000 and multiplied by 0.03\$, resulting in 0.099\$. The total costs for an exemplary call, therefore, amount to 0.11\$. A detailed cost analysis is presented in the following chapter.

#### Variables and parameters:

- Input tokens (verification): input
- Output tokens (check): output
- Input Tokens (vulnerability analysis): inputRAG
- Output tokens (vulnerability analysis): outputRAG
- Costs per 1000 input tokens:  $C_{in}$
- Costs per 1000 output tokens:  $C_{out}$

$$\begin{aligned}
\text{Cost Input Total} &= \frac{(\text{input} + \text{inputValid})}{1000} \times C_{\text{in}} \\
\text{Cost Output Total} &= \frac{(\text{output} + \text{outputValid})}{1000} \times C_{\text{out}} \\
\text{Cost Total} &= \left( \frac{(\text{input} + \text{inputValid})}{1000} \times C_{\text{in}} \right) + \left( \frac{(\text{output} + \text{outputValid})}{1000} \times C_{\text{out}} \right)
\end{aligned}$$

### 3.4.5 Results Analysis

This chapter presents and discusses the experiment's results. The aim was to determine the efficiency of LLMs in identifying known vulnerabilities and to compare the costs of using them. Particular attention was paid to how pre-trained models perform in contrast to models that have access to a broad knowledge of software vulnerabilities and how well models can assess whether the software is affected by a vulnerability based on vulnerability descriptions and versions, even if this is not explicitly present in the vulnerability description.

Table 14: Performance Metrics of Different Models

Model	Accuracy	Precision	Recall	F1-Score	MCC
<b>GPT-4-Turbo with Vector-Index (Az. AI Search)</b>	<b>0.8191</b>	<b>0.9612</b>	0.8105	<b>0.8794</b>	0.5600
GPT-4-Turbo	0.0833	0.1026	0.2162	0.1391	-0.8155
Mistral Large	0.3401	0.2558	0.1447	0.1849	-0.3361
<b>GPT-3 with Vector-Index (Az. AI Search)</b>	0.7952	0.8220	<b>0.8151</b>	0.8186	<b>0.5836</b>
GPT-4-Turbo with Vector-Index (LlamaIndex)	0.3120	0.4853	0.3929	0.4342	-0.4344

Table 15: Confusion Matrix Values for Different Models

Model	TP	FP	TN	FN
<b>GPT-4-Turbo with Vector-Index (Azure AI Search)</b>	<b>124</b>	<b>5</b>	30	29
GPT-4-Turbo	8	70	1	29
Mistral Large	11	32	39	65
GPT-3 with Vector-Index (Azure AI Search)	97	21	<b>70</b>	<b>22</b>
GPT-4-Turbo with Vector-Index (LlamaIndex)	33	35	6	51

**GPT-4-Turbo (LlamaIndex & text-embedding-ada-02)** The LLMAIndex GPT-4 model, which has knowledge of all vulnerabilities up to April 2024 and uses this knowledge via the vector index Simple vector of the LlamaIndex, has shown a low overall performance in identifying vulnerabilities. According to the table 15, the model identified 33 true positive cases, i.e. correctly identified 33 vulnerabilities.

The accuracy of around 31.2% shows that the model could only correctly classify a limited number of positive and negative cases. This provides a suboptimal baseline performance and indicates that the model cannot effectively distinguish between the presence and absence of a vulnerability.

The precision of 48.5% indicates that the model has moderate reliability in positive predictions. This means that a significant number of the vulnerabilities identified by the model are not actual, with a relatively high number of false positives (35 cases).

The recall value of 39.3% shows that the model only captures a portion of the actual vulnerabilities. The model's weakness is reflected in the high number of false negatives (51), which suggests that the model has overlooked many actual vulnerabilities. The recall value shows that the model failed to detect around 60.7% of all actual vulnerabilities.

Overall, the results are weak, with moderate precision and even lower scores in other vital metrics such as recall. The F1 score of 43.4% shows that the balance between recall and precision is suboptimal, and there is clearly room for improvement. The MCC of -0.434 indicates that the model's predictions have a negative correlation



with the actual data. This calls into question the general predictive power of the model.

**GPT-4-Turbo (Azure AI Search & text-embedding-ada-02)** The GPT-4-Turbo model, which has knowledge of all vulnerabilities up to April 2024 and uses this knowledge via the Azure AI Search Service’s Vector Index, has shown high overall performance in identifying vulnerabilities. According to table 15, the model identified 124 true positive cases, meaning that 124 software and version inputs were correctly identified as affected by a vulnerability.

The accuracy of about 81.9% shows that the model can successfully classify the majority of positive and negative cases correctly. It provides a solid baseline performance, indicating that the model can effectively distinguish between the presence and absence of a vulnerability.

The precision of 96.1% shows that the model is highly reliable in making positive predictions. This means that almost all vulnerabilities identified by the model are actual vulnerabilities, although there were a small number of false positives (5 cases).

The recall value of 81% shows that the model captures a large proportion of actual vulnerabilities. The model’s weakness is particularly evident in the relatively high number of false negatives (29), which suggests that the model has overlooked actual vulnerabilities. The recall value shows that the model failed to detect around 19% of all actual vulnerabilities, which limits its effectiveness and reliability.

Overall, the results are mixed, with very high precision but lower values in other vital metrics such as recall. The F1 score of 87.9% shows that the balance between recall and precision is relatively reasonable, but there is still room for improvement. The MCC of 0.56 indicates that the model’s predictions have a moderate correlation with the actual data, confirming the model’s overall predictive power and reliability.

The costs incurred for this run can be summarized as shown in Table 16. A total of 32,954 input tokens and 647,852 output tokens were generated. If the prices from Table 3.4.4 are included, total costs of around \$19.77 for testing 100 software programs.

category	number of tokens	cost (in Dollar)
Cost input total	32954	0.32954
Cost request total	647852	19.43556
Total costs	-	19.7651

Table 16: cost of the prompts

**GPT-3 (Azure AI Search & text-embedding-ada-02)** The GPT-3 model, which has knowledge of all vulnerabilities until April 2024 and uses this knowledge via the Azure AI Search Service's Vector Index, has shown an overall acceptable performance in identifying vulnerabilities. According to the 15 table, the model identified 101 true positive cases, meaning that 101 software and version inputs were correctly identified as being affected by a vulnerability.

The accuracy of approximately 68.7% shows that the model successfully correctly classifies the majority of positive and negative cases. This provides a solid baseline performance and indicates that the model can effectively distinguish between the presence and absence of a vulnerability.

The precision of 65.6% indicates that the model has moderate reliability in positive predictions. This means that many of the vulnerabilities identified by the model are actual vulnerabilities, although there were a significant number of false positives (53 cases).

The recall value of 91.8% shows that the model captures the majority of actual vulnerabilities. The weakness of the model is reflected in the relatively low number of false negatives (9), which suggests that the model missed few actual vulnerabilities. The recall value shows that the model failed to detect only about 8.2% of all actual vulnerabilities.

Overall, the results are acceptable, with a high recall, but the precision could be improved. The F1 score of 76.5% shows that the balance between recall and precision is moderate and there is room for improvement. The MCC of 0.378 indicates that the predictions of the model have only a moderate positive correlation with the actual data.

The costs incurred for this run can be summarized as shown in Table 17. A total of 31333 input tokens and 695418 output tokens were generated. If the prices from Table 3.4.4 are included, this results in a total cost of around \$1.06 for testing 100 different software programs.

category	number of tokens	cost (in dollars)
Total cost input	31333	0.0156665
Total cost request	695418	1.043127
Total costs	-	1.0587935

Table 17: cost of the prompts

**GPT-4-Turbo (OpenAI)** The GPT-4-Turbo model has shown low overall performance in identifying vulnerabilities. According to the Confusion Matrix, the model

identified eight true positive cases, which means that eight software and version inputs were correctly identified as affected by a vulnerability.

The accuracy of around 8.3% shows that the model could only classify a small number of positive and negative cases correctly. This provides poor baseline performance and indicates that the model cannot effectively distinguish between the presence and absence of a vulnerability.

The precision of 10.3% shows that the model is very unreliable in making positive predictions. This means that almost all vulnerabilities identified by the model are, in fact, not real vulnerabilities, and there are a high number of false positives (70 cases).

The recall value of 21.6% shows that the model only captures a small proportion of actual vulnerabilities. The weakness of the model is particularly evident in the high number of false negatives (29), which suggests that the model has overlooked many actual vulnerabilities. The recall value shows that the model failed to detect around 78.4% of all actual vulnerabilities, which severely limits its effectiveness and reliability.

Overall, the results could be more robust, with very low precision and even lower values in other vital metrics such as recall. The F1 score of 13.9% shows that the balance between recall and precision is inferior, and there is clear room for improvement. The MCC of -0.816 indicates that the model's predictions have a negative correlation with the actual data. This calls into question the general predictive power and reliability of the model.

The costs incurred for this run can be summarized as shown in Table 18. A total of 11,866 input tokens and 45,228 output tokens were generated. If the prices from Table 3.4.4 are included, total costs of around \$1,47 for testing 100 software programs will be reduced.

category	number of tokens	cost (in dollars)
Total input costs	11,866	0.118
Cost request total	45,228	1.356
Total costs	-	1,474

Table 18: cost of the prompts

**Mistral Large** The Mistral Large model has shown moderate overall performance in identifying vulnerabilities. According to the Confusion Matrix, the model identified 11 true positive cases, meaning that 11 software and version inputs were correctly identified as affected by a vulnerability.

The accuracy of around 34.0% shows that the model was only able to classify a limited number of positive and negative cases correctly. This provides poor baseline performance and indicates that the model cannot effectively distinguish between the presence and absence of a vulnerability.

The precision of 25.6% shows that the model is unreliable in making positive predictions. This means that many of the vulnerabilities identified by the model are not real vulnerabilities, and there are many false positives (32 cases).

The recall value of 14.5% shows that the model only captures a tiny proportion of the actual vulnerabilities. The weakness of the model is particularly evident in the high number of false negatives (65), which suggests that the model has overlooked many actual vulnerabilities. The recall value shows that the model failed to detect around 85.5% of all actual vulnerabilities.

Overall, the results are moderate, with low precision and even lower values in other important metrics such as recall. The F1 score of 18.5% shows that the balance between recall and precision is not satisfactory, and there is clearly room for improvement. The MCC of -0.336 indicates that the model's predictions have a weak negative correlation with the actual data. This calls into question the overall predictive power and reliability of the model.

The costs incurred for this run can be summarized as shown in Table 19. A total of 17,865 input tokens and 59,905 output tokens were generated. If the prices from Table 3.4.4 are included, this results in total costs of around \$0.79 for testing 100 different software programs.

category	number of tokens	cost (in dollars)
Total input costs	17,865	0.071
Cost request total	59,905	0.718
Total costs	-	0.789

Table 19: cost of the prompts

**Summary** To summarize, the first experiment produced some interesting findings on vulnerability detection performance. The GPT-4-Turbo model using vector indices performed significantly better than other models in pure pre-trained status.

The model that used the vector index from Azure AI Search stood out. It was interesting to see that the GPT-3 model showed good efficiency close to the GPT-4 Turbo model. The GPT-4 model with the Vector index offers significantly higher precision and an improved F1 score compared to the GPT-3 model. However, the GPT-3 also offers acceptable performance and costs only a fraction compared to the GPT-4 Turbo model.

The conclusion here is that it makes sense to conduct further research with the GPT-3 model to improve the overall performance here through customized prompting, data sources, embedding, or configuration by parameters. As assumed in the hypothesis, the comparison of the performance of the different models mapped in Diagramm 8 is significantly different. The pre-trained models, as hypothesized, did not perform seriously enough to be suitable for use in vulnerability detection in the initial state. Likewise, while the GPT-4 Turbo model using the in-memory vector indices performed generally better than the pre-trained models, its performance does not come close to the models using Azure AI Search. As LLamaIndex itself describes 2.1.1, this should be used for testing purposes but is less suitable for critical applications such as vulnerability detection.

Through manual testing, it was observed that for many vulnerabilities that did not have a matching version description implied, the vulnerability could still be identified or declared non-convergent for the software and version. A precise measurement made the data base difficult and required manual verification. The high number of true negative cases indicates that the models correctly identified these inconsistent statuses.

In the initial situation, as explained in section 3.4.3, the version was extracted from the Version field. However, as already explained, this often differs from the CVE description. For example, the Version field contains the version number: 1.3.5, but the description says "...before 1.3.5", and the model has correctly recognized here that there is no vulnerability, which would mean that this was a case in which the model recognized that the version belongs differently to the version description. After analyzing the results for this experiment, this only includes the versions that are described with 'before,' 'through,' or 'prior to' in the description. Versions that are in a range between versions, as can also occur in the description of vulnerabilities, could not be tested due to the nature of the CVE records since, during the analysis, there is never a number in the version field that is in a range, but only versions that are declared in the text with 'before' or similar.

The following experiments take a closer look at these special cases. Thus, the hypothesis that LLMs are able to use text comprehension to recognize weak points could be proven in certain respects, but it remains to be seen whether text comprehension also recognizes weak points, as the versions are in certain ranges.

The cost analysis performed in the experiment shows that high costs are incurred when using the GPT-4 Turbo approach with Azure AI Search. The reason for this is that when using embeddings, the knowledge is not directly available in the model. However, the model first vectorizes the input signal, as described in section 2.1.1 by the RAG approach, and compares it with the vector database. The context found there can vary in size depending on the index but includes all connections to the input found in the index. This is passed to the model together with the input prompt,

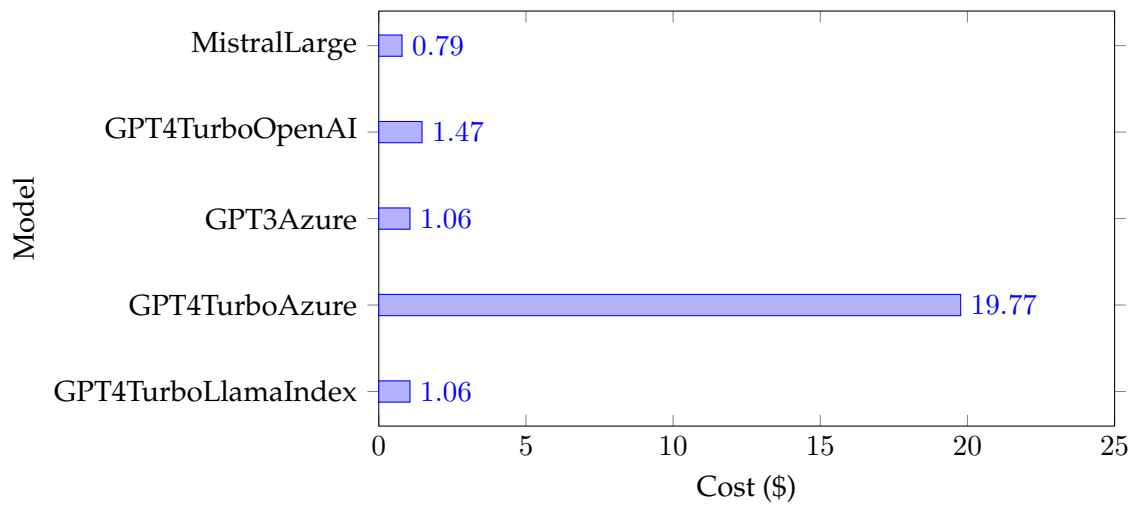


Figure 7: Comparison of Total Costs for Evaluating 100 Software Programs

which means that very high numbers of tokens are processed by the model. As the GPT-4-Turbo model has much higher consumption costs compared to the other models, higher costs are incurred even when matching 100 data records.

This experiment did not cover the yet-to-be-explored attempt to iterate different models with different RAG approaches.

In summary, this experiment demonstrated that models such as GPT-4 could provide added value in vulnerability detection. Further experiments will explore how these models can perform in detecting vulnerabilities and threat ratings and how they compare to traditional assessment tools.

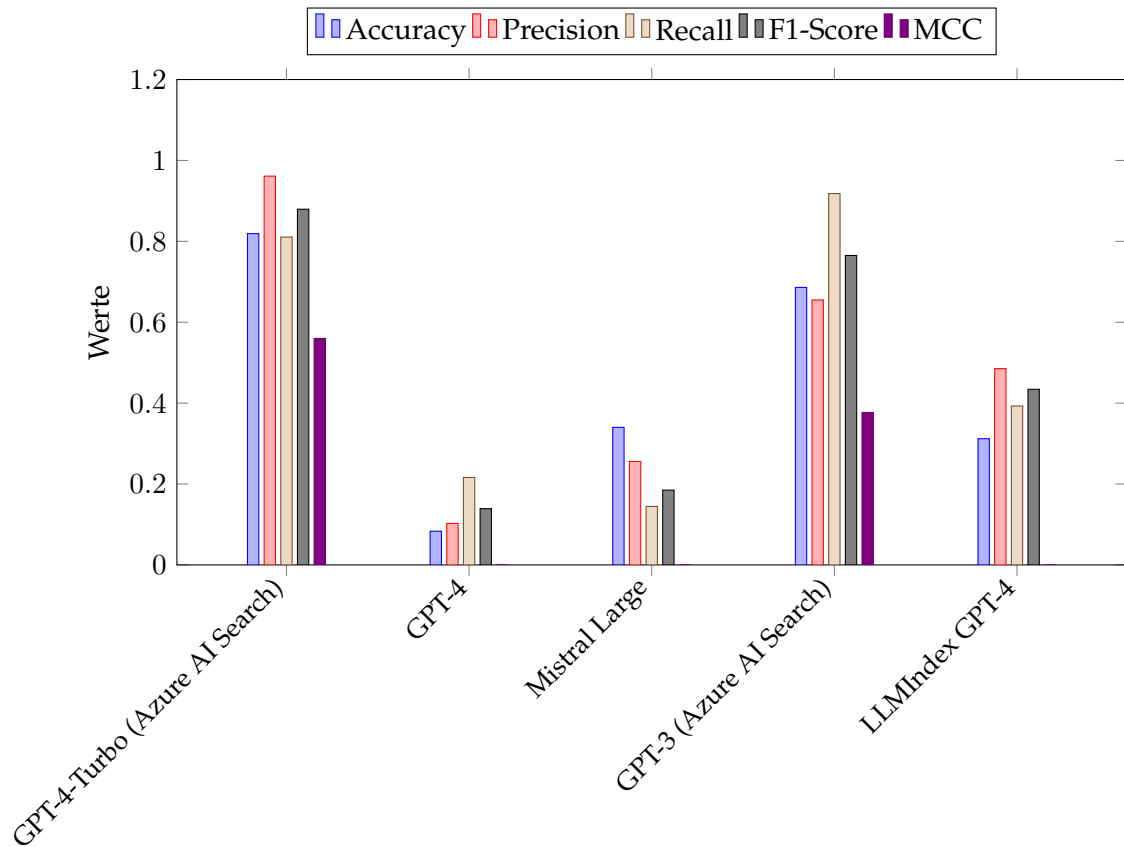


Figure 8: Comparison of the Models

## 3.5 Experiment 2

### 3.5.1 Experimental Objectives and Hypotheses

In this experiment, the aim is to determine whether the LLM that has shown the best performance in previous experiments, i.e., the GPT-4 model with the Azure AI Search approach, also shows good efficiency in classifying a vulnerability based on the CVE description. Furthermore, the GPT-3 model will also be included in the experiments with the RAG approach, as it was able to deliver acceptable results in the previous experiment. Since the cost analysis here calculated a significant difference between the models, the analysis for the GPT-3 model will be included here. Based on the findings from the sections 2.5 and 2.6, an essential gap between the coverage of vulnerabilities and score classifications via the CVSS score in the CVE database was identified. As stated in section 2.6, only 20.06% of all CVE records have a classification via the CVSS score. To answer the research question of how

the efficiency of an LLM in detecting vulnerabilities in IT networks compares to established vulnerability assessment tools, it is hypothesized that LLM models can classify the vector scores of vulnerabilities using an extensive existing database of vulnerability information, including their CVSS scores, even if they are not part of the existing knowledge base.

Answering this hypothesis is crucial to the course of this master’s thesis, as the goal of network vulnerability detection is to classify the riskiness of the vulnerability as a critical component in deciding to what extent a network is at risk. This experiment does not make a direct comparison to conventional assessment tools. However, at this stage, it is assumed that static tools are not able to classify vulnerabilities that do not have a CVSS rating. However, this hypothesis will be examined in greater depth in the later experiment.

This experiment benefits from the findings of Aghaei et al. [1]. In their research, Aghaei et al. [1] developed a model called CVEDrill, which is based on SecureBERT (see Section 2.3). This model accurately classified CVSS vectors using the description of CVE records. Their study assessed the prediction accuracy of various attack vectors (see Section 2.6) based on 100 randomly selected CVE descriptions as input parameters. As shown in Table 20, the prediction accuracy was compared with that of the ChatGPT model. The study does not specify which GPT model was used, but given that it was published in September 2023 and ChatGPT-4 was released in March 2023, it is assumed that the tests were conducted using the GPT-4 model.

Table 20: Comparative evaluation of predictive accuracy for CVSS vectors in a sample of 100 randomly selected CVEs disclosed in 2023.

Tool	AV	AC	PR	UI	S	C	I	A
ChatGPT	0.52	0.78	0.85	0.55	0.43	0.36	0.81	0.42
CVEDrill	0.82	0.86	0.76	0.91	0.87	0.81	0.81	0.77

Therefore, this experiment builds on the results of the study by Aghaei et al. [1] and tests the hypothesis that a model that has a broader knowledge of vulnerabilities can lead to better performance results, which were determined in the study of the pure ChatGPT (GPT-4).

### 3.5.2 Methodology

The methodology of this experiment is designed to investigate the ability of the GPT-4 and GPT-3 models using the RAG approach to identify vulnerabilities based on the description captured in a CVE record. Understanding the performance of these models will be integrated into Experiment X to gain a comprehensive picture of their efficiency and effectiveness compared to traditional vulnerability assess-



ment tools. The hypothesis that will be addressed in the later experiment is that traditional tools cannot classify vulnerabilities where a severity score has yet to be formally established. Therefore, the outcome of this experiment will provide an essential insight into whether LLMs are able to assist in this area of cybersecurity. This will build on the research of Aghaei et al. [1] to take advantage of their findings already and gain new insights not covered there.

In addition to measuring the match rates of various CVSS vector metrics (see 2.6), a cost analysis of the tokens used is performed, and it is checked whether the detection performance is close to the original result. This is evaluated based on the determined severity level (LOW, MEDIUM, HIGH, CRITICAL).

The database for this experiment comprises the complete CVE database with all filtered and processed CVE records up to April 2024. During the creation of the knowledge database in the form of the RAG, one parameter that was also passed was the CVSS vector string of the CVE vulnerability (see 2.6 and 3.3). The assumption is that this knowledge enables the model to make a more precise statement about a CVSS vector string and thus determine the CVSS metrics better than the pure pre-trained model, which has also been described as a vulnerability as a basis.

As in the last experiment, a database of 100 CVE records is used to test these against the model. This also overlaps with the number of test data used by Aghaei et al. [1] in the experiment. In order to rule out the possibility that the model draws on existing knowledge, only CVE records that were not yet available in the database at the time the RAG was created are used. An updated data set was therefore obtained from MITRE, which has a total of 233,151 CVE records in May 2024. When the RAG was created, there were still 228,713 published CVE entries. This results in a difference of 4,438 new CVE records created in the meantime. This ensures that the model does not simply fall back on existing knowledge but should use this knowledge to classify unknown CVE records. The possibility of a possibly equivalent description of an older vulnerability was disregarded.

The 100 CVE records used for the experiment are checked during processing to see whether they contain a CVSS vector string. Only if this is present is the data set used for processing. This allows the result of the model to be compared with the official CVSS vector string in order to determine a precise accuracy rating.

The setup for this project is different from the first approach used in Experiment 1. The experiment receives, as an input parameter, the description of a published CVE vulnerability, with no additional information provided. A CVE description is created by the individual or organization that discovered the vulnerability and reported it to MITRE. The lifecycle is described in Section 2.5. The requirements for a description follow the rules defined by MITRE<sup>25</sup>. The description must provide

---

<sup>25</sup>[https://www.cve.org/ResourcesSupport/AllResources/CNARules#section\\_8\\_](https://www.cve.org/ResourcesSupport/AllResources/CNARules#section_8_)

enough information for the reader to have a reasonable understanding of the affected products. Furthermore, the vulnerability type, root cause, and impact must be present in the text. Providing a version is not necessarily required.

For example, this CVE description of CVE-2021-24910 describes the existing vulnerability as follows:

"The Transposh WordPress Translation WordPress plugin before 1.0.8 does not sanitize and escape the parameter via an AJAX action (available to both unauthenticated and authenticated users when the curl library is installed) before outputting it back in the response, leading to a Reflected Cross-Site Scripting issue."

Compared to the rules described, the description fulfills the requirements for a CSV entry. The description clearly names the product concerned, namely the "*Transposh WordPress Translation WordPress plugin*" (information about the product). The type of vulnerability is identified as "*Reflected Cross-Site Scripting*" (vulnerability type). The description explains that the root cause of the issue is that the 'a' parameter is not sufficiently sanitized and secured before it is returned in the response (root cause). The description implies the type of vulnerability (XSS) that makes it potentially possible to execute malicious code in the context of a victim's browser (Impact).

The area of severity classification has already been addressed by several researchers, including Aghaei et al. [1], as described above, as well as by Shahid and Debar [52]. Both types of research were designed to determine the severity of CVSS based solely on the content of the CVE description. Both were able to achieve good results with their trained models. The results of Aghaei et al. [1] have already been shown in Table 20. Shahid and Debar [52] were able to achieve a relatively high accuracy ranging from 83.79 % to 96.07 %. Both researchers chose the approach of analyzing each vector metric individually.

In this paper's experiment, a pre-trained model is used to develop full-text understanding by inputting the description to generate a vector string for all metrics. The input, as described above, consists of the CVE description and a prompt designed from it, shown in Listing 18. The model is instructed to return a vector string from the given description after the text has been analyzed.

Listing 18: Input Prompt for the Model

```

1  messages = [
2  {
3    "role": "user",
4    "content": (
5      f"Generate a CVSS vector string from a CVE vulnerability description. Analyze the description and create
        the CVSS vector string based on this analysis. This is the CVE vulnerability description:
        {description}. Respond only with the CVSS vector string.
6      "
7    )
8  },
9  ],

```

The expected output is a vector string in the standard format from 2.6, in this form: "AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H". Once the vector string has been output by the model as a response parameter, it is used to perform further calculations. Further to the research of Shahid and Debar [52] and Aghaei et al. [1], this experiment not only looks at the vector string but also at the resulting severity level. The assumption here is that the severity level is also a meaningful metric for evaluating a vulnerability. Therefore, even if the vector string generated by the model as a response parameter and the original vector string officially associated with the CVE entry differ, an approximation of the generated base score and the official base score can lead to similar severity levels. In this experiment, therefore, not only is the ability of the model to generate the vector string measured but the serenity score is also used as a performance parameter.

**Example 3.1** (RAG System Process). As an example, a vector string predicted by the model would include the following metrics and their results: "AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:H". This would provide a calculated base score of 6.5, which would mean a Serenity level of \*MEDIUM\*. However, the official entry for this vulnerability would be "AV:N/AC:L/PR:L/UI:R/S:C/C:N/I:N/A:L" and, in contrast to the 6.5 base score, would have a base score of 4.1. Both base scores differ because, in this case, the availability impact was rated higher by the model than officially. This means that a security vulnerability more severely impairs the availability of an affected system or component.

Therefore, the two base scores are 2.4 points apart. However, both are at the same Serenity level \*MEDIUM\*. For the assessment of vulnerabilities, both vector strings match in terms of the serenity level.

Therefore, after receiving the response from the model, the vector string is ex-

tracted using a regex. This vector string is used to calculate the base score using the formula from 2.5. The base score is checked against the severity levels and classified according to the level of the score. The model’s response parameters are used to record the input and output tokens. The predictions of the model, the calculations of the base score, and the severity level are saved together with the original values from the CVE entry and the CVE ID for each response in a CSV file for later evaluation. The CSV is structured as in Listing 19.

Listing 19: Output Format for the Results

1	CVE ID, Generated Vector String, Calculated Base Score, Calculated Severity, Original Base Score, Original
2	Vector String, Original Severity, Input Tokens, Prompt Tokens
3	CVE-2024-25015,AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H,7.5,High,7.5,CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H,HIGH,77,350
	CVE-2024-25016,AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H,7.5,High,7.5,CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H,HIGH,103,480

After each run, the saved CSV file is passed to another script, which calculates the performance metrics. This script checks the predictive accuracy values for each CVSS metric and calculates the match rate for the severity. The following chapters describe this in more detail.

The limitation that arises is that a CVE entry only sometimes contains all the information in the description.

3.5.3 Experiment Execution

The experiment used specific specifications, scripts, and technologies. The description of the required components is followed by a detailed description of how the experiment was carried out and the methods used to determine the required results.

The hardware specifications used for this experiment are listed in Table 3.4.3.

Various Python scripts were developed to carry out the experiment, which are listed in Table 3.5.3. These scripts execute specific logic to test the hypotheses formulated in section 3.4.1. Each script fulfills a separate task and is called manually in sequential order.

Script	description
<b>cvssexp.py</b>	Runs the experiment, processes 100 selected CVE records, and interacts with corresponding LLMs depending on the configuration.
<b>cvssCalc.py</b>	Calculates the Confusion Matrix based on the experiment results.

Table 3.5.3 includes the technologies, REST APIs, and data formats used in this experiment.

<b>technology/service</b>	<b>usage</b>
<b>Azure Blob Storage</b>	Data Storage for CVE Database
<b>Azure AI Search</b>	Vector Database
<b>Azure OpenAI</b>	LLM models and embedding models
<b>REST API</b>	interaction with Azure, OpenAI
<b>Data formats</b>	JSON, CSV

This experiment is based on the database that was already created for the last experiment. This means that the dataset still contains the CVE records of all CVE entries up to April 2024 and has yet to be further modified for this experiment. A Python script called "cvssexp.py" was developed that iterates through the delta of the sourced CVE records, which cannot be known to the model either by the pretraining or by the dataset, and extracts the vector string from the response as described above. Since in the study by Aghaei et al. [1] it was not clear which GPT model was used in the evaluation, in this experiment, the GPT-3 and GPT-4 turbo models were used in the scripts, each with and without the RAG approach.

The delta is made up of the 4,438 CVE records described above. The script "cvssexp.py" iterates through the download folder of the CVE records and starts with the folders that were not yet present in the old data set. The script filters the entries for all CVE records that contain a CVSS version 3.1 vector string in the data. This is required for validation. The description is extracted from the data records and passed to the model.

Within the script, the parameters are adjusted manually after each run. This includes the model (GPT-4-Turbo or GPT-3) and whether or not the Azure extensions should be used for the RAG. In this experiment, all runs with the different models receive the same data. The first 100 CVE records that meet the criteria (CVSS vector score 3.1 is available) are selected from the delta rather than randomly per run.

The values required for the result analysis are formatted as above and saved in a CSV file.

This CSV file is manually passed as input in the script "cvssCalc.py" and run for each result file. In this experiment, the Python library "sklearn" was used for the required metrics<sup>26</sup>. The library contains methods for calculating metrics that are used mainly in the field of machine learning. The methods used in this experiment calculate the accuracy score. To do this, the library creates a confusion matrix in the background and calculates the required accuracy value, which it returns as an

<sup>26</sup><https://domino.ai/data-science-dictionary/sklearn>

answer. It receives each metric in a loop as an input parameter for each vector string. The metrics were first defined in an array.

The script's output parameters are the predictive accuracy values for each CVSS metric and the severity match rate. The results are copied manually from the console and saved to a text file. The following chapters provide a more in-depth look at the results, a cost analysis, and a final evaluation of the experiment.

### 3.5.4 Measurement and Evaluation

As in the first experiment, the performance of the models is evaluated using the metrics of a confusion matrix. The calculation of the Confusion Matrix was taken from the sklearn library described above. Further information on the Confusion Matrix is described in the experiment above. This experiment is based on the Confusion Matrix. In the last experiment, the Confusion Matrix was defined by a binary classification. In this experiment, however, a multi-class confusion matrix is used. The multi-class confusion matrix is used in this experiment because there are several classes involved in this part of the elaboration.

With binary classification, we only had the case where the model gave a prediction as to whether a vulnerability was present and which vulnerability it was. The check was both automated and manual there, but the performance was not measured to determine whether the model detected all existing CVE entries associated with that vulnerability. This would have required an even more profound examination of the existing data set, which was different from the focus of the experiment. Therefore, binary classification was sufficient for this approach.

In this experiment, the performance is based on the correct assignment of metrics and their values. Thus, a metric of the CVSS score can have several states, as already described in section 2.6. For example, the metric "Attack Complexity" can have the state "Low (L)" or "High (H)." The "Attack Vector" metric can have up to four different states. Therefore, when measuring performance and effectiveness, the Confusion Matrix is performed with a multi-class classification, whereby a comparison is performed for each class of the metric.

In this experiment, the values True Positives (TP), False Negatives (FN), and False Positives (FP) are used. True Negatives (TN) are also relevant, but TP, FN, and FP are decisive for calculating the metrics Accuracy, Precision, Recall, and F1-Score.

Each of the metrics of the CVSS score 2.6, which is measured within the experiment, is used to evaluate performance. The metrics Accuracy, Precision, Recall, and the F1 score are measured for this purpose.

In order to obtain a more general overall picture, the results of research that has already produced results in this subject area were used in the evaluation. Therefore, the results from this experiment are also compared with the results of Aghaei et al. [1] and Shahid and Debar [52] for comparison. This allows a broader view of this work, as the focus of the research described was on models specifically designed and trained for this use case.

The analysis of the results also evaluates the "Predictive Accuracy for CVSS Vectors" of the models from the research. The results should show to what extent the accuracy is different or closer compared to the approach with the pre-trained GPT-3 and GPT-4 models with and without RAG. The metrics described above are also considered via predictive accuracy in order to gain a better understanding of the performance of the models. For the metrics, the weighted average across the classes is considered, as certain classes of vector metrics have a higher presence in the data than others.

An analysis should have been carried out within this work. The assumption is supported by the results of Aghaei et al. [1]. In their research, the frequency of the classes was mapped in a table using a confusion matrix. For example, of a total number of 4791 data sets, the "Attack Vector" has the class "N" with a presence of about 76.62

A further value is measured for the models used and enriched in this work. As already explained above, the severity level is also considered in this paper. The assumption here is that a severity score can also reflect a classification of a vulnerable network/computer, even if deviations exist between the predicted and original CVSS vector strings. For this purpose, an accuracy calculation is also performed on the "Severity" field, and a comparison is made of how often vulnerabilities are at the same severity level. In this way, any deviations can be balanced out in order to obtain an assessment of the vulnerability.

### 3.5.5 Results Analysis

This chapter presents and discusses the experiment's results. The aim was to find out how efficient LLMs are at generating a correct CVSS vector string from a description of a CVE entry. Both pure pre-trained models and models with a rag approach were compared. The input data was the same for each run with different configurations. In each case, 100 CVE entries were run, which were also not available at the time the model was trained or within the knowledge database.

The results are measured using the metrics explained above. To enable a comparison with other research, the results of Aghaei et al. [1] are included in this results analysis.

**GPT-4-Turbo (Azure AI Search & text-embedding-ada-02)** The GPT-4 model, which uses the Azure AI Search approach and has access to a knowledge base with all existing CVE entries up to April 2024, achieved the values from Table 21 in the analysis. For the metric "Attack Vector (AV)" achieved a very high accuracy of 93%, which means reliable identification of the matching classes of the metric. The recall of 93% shows that the model was able to assign almost all classes from the CVE description correctly. The interplay between precision and recall shows the balance indicated by the F1 score. The precision of 93% is in the excellent range and achieves the highest value compared to all other models.

For "Access Complexity (AC)," the accuracy is 87%, which is lower than AV. The model effectively recognizes the metric from the CVE description but generates some false positives. With a recall of 87%, the model identifies the correct classes of the metric in most cases despite complex conditions.

Metric	Precision	Recall	F1-Score	Accuracy
AV	0.93	0.93	0.92	0.93
AC	0.84	0.87	0.84	0.87
PR	0.74	0.73	0.73	0.73
UI	0.82	0.82	0.82	0.82
S	0.78	0.77	0.76	0.77
C	0.72	0.70	0.70	0.70
I	0.69	0.68	0.68	0.68
A	0.62	0.64	0.57	0.64

Table 21: Summary of Metrics from Confusion Matrices for Various CVSS Metrics

For *Privileges Required (PR)*, the precision value drops to 74%, which shows that many contexts of the descriptions could not be correctly interpreted and classified by the model. The F1 score of 73% indicates that the differentiation of the *Privileges*



*Required* classes could be improved. The accuracy of 73% shows that the understanding of these classes needs to be optimized.

*User Interaction (UI)* has an average accuracy of 82% across all metrics, which is a solid performance in correctly classifying these metric classes.

The detection of the *Scope (S)* classes performs better than the classification of the PR metric but shows weaknesses in precision and accuracy with 78% each. Nevertheless, it can classify most cases correctly from the CVE description.

*Confidentiality (C)*, *Integrity (I)* and *Availability (A)* are the worst recognized metrics. Although all more than 50% of the classes were correctly assigned, only *Confidentiality* achieved precision and accuracy values of over 70%. The worst performing metric was *Availability* with a precision of 62% and an accuracy of 64%.

The agreement rate of almost 60% for the severity classification means that more than a third of the classifications are inaccurate. For use in safety-critical environments, more than such a classification is needed to meet the requirements.

The total cost of the GPT-4 model with the Azure approach (see table 22) was \$16.22.

category	number of tokens	cost (in Dollar)
Cost input total	15844	0.16
Cost request total	535404	16.06
Total costs	-	16.22

Table 22: cost of the prompts

**GPT-3 (Azure AI Search & text-embedding-ada-02)** The GPT-3 model has generally shown excellent performance in the classification of metrics. Overall, the GPT-3 model with Azure AI Search achieved better results than the GPT-4 model and approached the performance of the BERT (classifier) model. In the *attack vector (AV)* domain, the model achieved a precision of 92%, which is higher than the multi-classifier BERT model by Aghaei et al. [1]. The recall is also 92%, which means that the model was able to predict the class correctly in most cases.

*Access Complexity (AC)* also has a high precision of 90% and a recall of 89%. The model was even able to identify certain classes 100% correctly. For example, the class *High* was correctly classified in every given description. The F1 score shows a good balance between precision and recall, reaching 85%.

The weakest of all metrics was *Privileges Required (PR)*. Only a low precision of 60% was achieved here. The accuracy of 56% is the worst of all the models used.

*User Interaction (UI)* and *Scope (S)* again showed a strong performance with over 80%. UI achieved an accuracy of 85%, reflecting a high identification rate of correct classes.

The model shows the values for *Confidentiality (C)* and *Integrity (I)* in Table 23. A precision of 86% was achieved for C, which indicates good reliability, even if I has a solid value of 78%.

metric	precision	recall	F1 score	accuracy
AV	0.92	0.92	0.91	0.92
AC	0.90	0.89	0.85	0.89
PR	0.60	0.56	0.57	0.56
UI	0.88	0.85	0.85	0.85
S	0.82	0.80	0.80	0.80
C	0.86	0.84	0.84	0.84
I	0.78	0.78	0.78	0.78
A	0.61	0.69	0.64	0.69

Table 23: Summary of the weighted metrics from the confusion matrices of various CVSS metrics

**Summary** In this experiment, the capabilities of the GPT-4 Turbo and GPT-3 Turbo models were investigated, both of which were tested in pretrained mode and with an additional RAG approach. The task was to analyze CVE descriptions and classify correct CVSS vectors and the associated base score. Literary sources were used to build on previous studies in this context. The work of Aghaei et al. [1] and Shahid and Debar [52] were considered, and the results compared with those in this work. Both studies used different approaches to those chosen for this work. In this work, a large dataset of already known CVSS vectors was used. The other studies were carried out on the basis of more fine-grained metrics per CVSS vector. During the implementation phase, an approach was also used that was to be approximated by a pre-trained model in the aforementioned studies.

For this purpose, models were finetuned via the OpenAI page. Two different approaches were chosen: In one, the GPT-3 model was trained on a massive dataset with mixed CVSS vectors. The other approach was based on the strategy of Aghaei et al. [1], where a separate classifier was trained for each CVSS vector. For this purpose, the finetuning was carried out with 300 data sets per CVSS metric, with a separate model trained by GPT-3 for each vector metric and later queried via a Python script for each metric to the corresponding model and then combined into a CVSS vector string. Due to the complexity of this approach and the complexity of the rest of the experiment, this part of the finetuning has yet to be included in this work. The depth of the topic of finetuning would not have justified the effort

that would have been required for the work. Nevertheless, the first results of the finetuning were included in the overall view in Table 24.

Model	AV	AC	PR	UI	S	C	I	A	Severity
<b>BERT (with classifier)</b>	0.91	<b>0.96</b>	<b>0.83</b>	<b>0.93</b>	<b>0.95</b>	0.87	0.87	<b>0.88</b>	N/A
GPT-3 with Azure AI Search	0.92	0.89	0.56	0.85	0.80	<b>0.84</b>	<b>0.78</b>	0.69	<b>0.68</b>
CVEDrill	0.82	0.86	0.76	0.91	0.87	0.81	0.81	0.77	N/A
GPT-4 pretrained	0.90	0.89	0.68	0.89	0.75	0.66	0.65	0.61	0.61
GPT-3 finetuned (raw)	0.89	0.84	0.68	0.82	0.77	0.74	0.73	0.55	N/A
GPT-4 with Azure AI Search	<b>0.93</b>	0.87	0.73	0.82	0.77	0.70	0.68	0.64	0.59
GPT-3 finetuned (per metric)	0.89	0.84	0.60	0.88	0.71	0.62	0.59	0.43	N/A
ChatGPT (laut Aghaei et al. [1])	0.52	0.78	<b>0.85</b>	0.55	0.43	0.36	0.81	0.42	N/A

Table 24: Performance comparison of LLMs

Overall, this experiment showed that the GPT-3 model, in combination with Azure AI Search, performed better than the GPT-4 model with the same approach. Surprisingly, the GPT-3 model was able to achieve better performance in this work than the CVEDrill model. The CVSS-BERT model by Shahid and Debar [52] continued to show the best performance of all models listed in the table 24 compared to the models used in this work and was able to classify 5 of the 8 CVSS vector metrics with the highest performance.

The severity was also compared in this experiment to determine how well the average CVSS base score could be correctly calculated using the CVSS metrics. The GPT-3 model with the Azure AI approach demonstrated an accuracy of 68%. As was also noted in the other studies, some metrics can be ranked very well by models and metrics, whereas classification is measurably more difficult. All models, except for the GPT model tested by Aghaei et al. [1], were able to achieve an accuracy of over 80

In general, a GPT model can be used to classify CVSS vectors well and appropriately. The challenge here also lies in the description of the CVE records, as a model can find it difficult to classify if the description is inaccurate or very short.

## **3.6 Experiment 3**

### **3.6.1 Experimental Objectives and Hypotheses**

As a final experiment, the LLM that performed best in this work was selected to provide the basis needed to perform this experiment. The previous experiments were designed so that Experiment 1 tested the extent to which models can detect vulnerabilities in software that have already been officially cataloged. The limitations of CVE descriptions make it challenging to deal with versions that need to be clearly defined. The following experiment examined the extent to which models are able to perform a severity assessment when only the description of the vulnerability is provided. In other words, to use the textual understanding that a model brings with it to use the textual information extracted from it to assess the severity of the vulnerability. This was based on the results and research of Aghaei et al. [1] and Shahid and Debar [52].

The aim of this experiment is to determine whether models are able to interpret scans in IT networks to such an extent that a statement can be made as to whether this network or the corresponding computer in the network is affected by a vulnerability and how critical this is to be classified. The research approaches for classifying network scans that were researched during the research phase exist, but they needed to be present more to build on the results. For example, a paper by Moskal et al. [39] deals in part with the analysis of network scans in Nmap reports as a step in the execution of a multi-agent system.

The hypothesis that is being put forward is that models that have an extensive knowledge base of vulnerability data can interpret scan reports from network scans and identify and classify affected software versions. More specifically, the assumption is made that powerful models can be used in network scans to identify vulnerabilities in IT networks, assign them to existing threats, and determine the degree of danger. In particular, the research question “How does the efficiency of an LLM in detecting vulnerabilities in IT networks compare to established vulnerability assessment tools?” should be addressed. It is assumed that models specially trained in the field of cybersecurity, as in the experiments, can achieve similar or better results than conventional assessment tools. For this approach, the Nessus tool was chosen, as discussed in Chapter 2.4.

### **3.6.2 Methodology**

The methodology of this experiment is designed to measure the performance of the GPT-4 Turbo and GPT-3 Turbo models, each with the knowledge base as a RAG approach, to determine the extent to which these models are able to interpret network

scans and identify possible vulnerabilities in this context. The focus is on software versions that, as in Experiment 1, are not available in a fixed schema and can either not be recognized or recognized less well by conventional vulnerability analysis tools. The execution of the experiment will benefit from the results of the first two experiments.

To do this, a framework is created within the experiment (referred to as AI-SecNet in this paper), which performs a sequence of processes to generate a result for evaluating the analyzed IT network. The evaluation includes an output of vulnerabilities within the network and the classification of the level of danger. The framework was created using Python scripts that perform a Nmap scan, extract the software and versions for each port from the scan result, and perform a query against the GPT-4 model with Azure AI Search for each extracted software and version, which has shown the best performance in classifying vulnerabilities in an experiment. After that, the framework also uses the GPT-3 model with Azure AI Search, which in the last experiment showed the best performance of the models adapted in this thesis in the area of CVSS classification. The output parameters of the framework include a list of the vulnerabilities found and the CVSS rating of the vulnerability.

Due to the specific requirements and limited facilities of the server infrastructure, an anecdotal evidence base is used in this study. The decision to use anecdotal evidence comes from the need to conduct a deeper analysis in a real-world simulation environment that cannot rely on traditional large-scale data collection due to obstacles arising from its specific configuration and limited scope. Instead, the focused analysis of the results from this limited number of tests allows for a detailed and nuanced consideration of the performance of the models. To answer the research question of how models measure efficiency and effectiveness compared to conventional assessment tools, this must be done in a sandbox environment. For assessment tools such as Nessus, it is not possible to test Nessus on a large number of data sets due to technical limitations, as it is designed to check specific IP addresses or IP ranges. The technical solution is possible, but it goes beyond the scope of this work. One approach would be to provision a large number of computers with pre-installed software. Therefore, anecdotal evidence is used to gain specific insights into the performance and effectiveness of the models. This means that the primary data set is not very diverse, but a focused investigation and targeted manual evaluation can be carried out on the selected data sets.

The network that is being tested for this purpose is a simulation environment that was set up for this experiment. The environment consists of a router, a web server, an application server, and a database server. All computers were set up as virtualized computers with the Ubuntu operating system 22.04.3 LTS. The structure of this simulation environment primarily serves several purposes. The primary goal is to establish a controlled and secure platform for testing and evaluating various security aspects of a modeled corporate network. Due to the effort and configuration

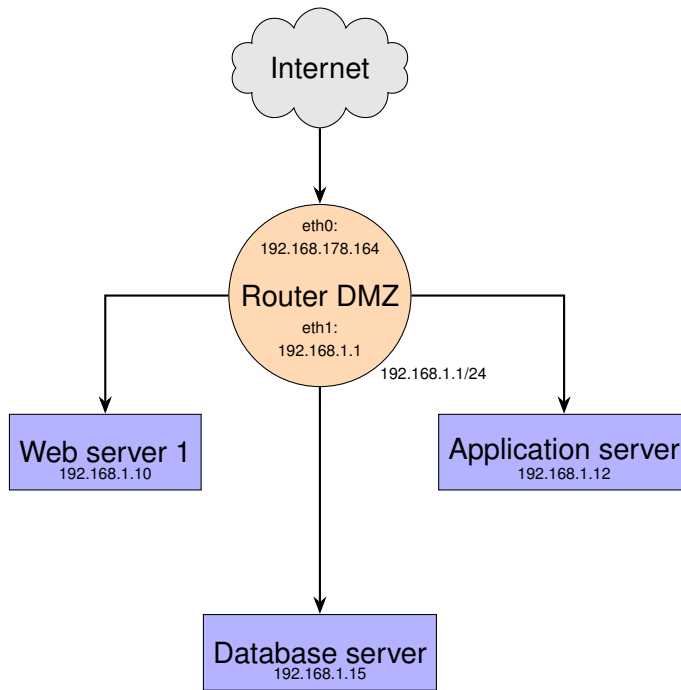
of the various systems in the network, the simulation is limited to a small company with an established network topology. Simulating actual network structures offers the opportunity to test hypotheses under almost actual conditions without the associated risks and costs in an actual business context. Such an environment is essential for conducting simulations during the experimentation phase in order to analyze specific use cases and potential threat scenarios. For the recipients, this simulation environment offers practical insight.

During the research, the GitHub repository Vulhub<sup>27</sup> caught our attention because it contains a collection of different software types as Docker images, and each of these software versions is associated with a known CVE entry. This enables flexible scaling of the various services to be scanned without having to install them with all their dependencies. The Docker images found there contain various types of software that can be classified, among other things, as software that is suitable in the context of web servers, application servers, and database servers. The respective software in the Docker image is directly linked to a known CVE vulnerability. During the research phase, it was noticed that older vulnerable software, in particular, cannot be easily installed on the Ubuntu system. It is particularly problematic if the version of the Ubuntu system is measurably younger than the service to be installed. During tests, it was noticed that outdated calls and program codes are often executed and can no longer be compiled on new Ubuntu systems. Therefore, the Docker images provide a good solution for flexibly installing vulnerable software.

The network topology, as shown in Figure 3.6.2, has been set up within the Microsoft Hyper-V virtualization environment on a Windows 11 machine. Each virtual machine was installed headless using the Ubuntu operating system with version 22.04.3 LTS, as mentioned above. The server configuration of the individual virtual machines is shown in Table 25.

---

<sup>27</sup><https://github.com/vulhub/vulhub>



The DMZ router is the entry channel for the actual network. It establishes the connection to the Internet via the Windows 11 host system and controls the network traffic to the outside. Furthermore, the DMZ router also serves as a Dynamic Host Configuration Protocol (DHCP) server to automatically distribute IP addresses from a defined address pool to the servers within the network. The DMZ router also acts as a firewall for the internal network. It ensures that network address translation (NAT) is carried out on a gateway, which enables computers in a private network to access the Internet under a shared public IP address by dynamically masking their internal IP addresses when forwarding packets.

Access to the internal computers from the outside is not possible and is controlled via port forwarding through the DMZ router. The setup is kept minimalistic so that the focus of the experiment is not on setting up the simulation environment. However, a corporate network should be simulated fundamentally. The basic configurations are documented in this work to help understand how the network was configured. The resource configuration and the operating systems installed are listed in Table 25.

During the setup of the Hyper-V environment, Ubuntu 22.04.3 Lite (without GUI) was downloaded as a server variant from the official Ubuntu site<sup>28</sup> as an image. A

<sup>28</sup><https://ubuntu.com/>

Table 25: Server Configurations

Property	Router DMZ	Web Server	Application Server	Database Server
OS	Ubuntu 22.04.3 LTS	Ubuntu 22.04.3 LTS	Ubuntu 22.04.3 LTS	Ubuntu 22.04.3 LTS
CPU	Intel i7-10700K	Intel i7-10700K	Intel i7-10700K	Intel i7-10700K
Memory	2 GB	5 GB	5 GB	5 GB
Storage	60 GB	50 GB	50 GB	50 GB
Interface 1	sandbox_ext. (eth0)	-	-	-
Interface 2	sandbox_int. (eth1)	sandbox_int. (eth0)	sandbox_int. (eth0)	sandboxsub_int. (eth0)

virtual machine was set up in Hyper-V with this image. For the virtual computers in the subnetwork of the DMZ router, a network card `sandbox_int` was configured in Hyper-V for internal communication and assigned to the virtual computers. The DMZ VM router was equipped with this network card as well as with another network card `sandbox_ext` to ensure communication with the internet via the host system.

Since the computers within the virtual network cannot be reached by the host computer, an SSH connection was established from the DMZ router, which can be reached by the host system, to the computers in the subnet to configure each computer. Termius was used to manage the SSH connections<sup>29</sup>.

The virtual computers within the subnet of the DMZ router were equipped with various software, such as Docker containers, as described above. To do this, Docker had to be installed on all computers first. In addition, an update and upgrade were carried out on the machines, and other required tools such as Nmap (for performing the network scan), unzip (for unpacking packages), and Docker were installed (see listing 20) and started.

Listing 20: Installing needed dependencies

```

1 sudo apt install nmap unzip
2 curl -s https://get.docker.com/ | sh
3 systemctl start docker

```

The selection of software for the various VMs was based on the domain of the respective server. For the web server, software products were selected in the preparation phase that was available in the GitHub repositories of Vulhub<sup>30</sup> and also received at least a “medium base score” in the CVSS classification.

<sup>29</sup><https://termius.com/>

<sup>30</sup><https://github.com/vulhub/vulhub>



Table 26: System Vulnerabilities Summary

Software	Version	Port	CVE
Webserver Software			
Drupal	7.57	8081	CVE-2018-7602
Jenkins	2.441	8091	CVE-2024-23897
PHP	7.0.28	8081	CVE-2018-10546 and more
Apache HTTP	2.4.54	8080	CVE-2006-20001 and more
jQuery	1.4.4	8081	CVE-2020-11022 and more
Joomla	4.2.7	8080	CVE-2023-23752
Confluence	8.5.3	8090	CVE-2023-22527
Grafana	8.2.6	3000	CVE-2021-43798
Application Server			
GitLab	13.10.1	8092	CVE-2021-22205
Apache Tomcat	9.0.30	8082	CVE-2020-1938 and more
phpMyAdmin	4.8.1	8070	CVE-2018-12613
Elasticsearch	v1.6.0	9200	CVE-2015-5531
JetBrains TeamCity	2023.05.3	8111	CVE-2023-42793
Oracle GlassFish	4.1.0	4848	CVE-2017-1000028
postfix	3.7.11	25	CVE-2023-51764
OpenSSH	8.9p1	22	CVE-2023-28531
Apache httpd	2.4.25	8070	CVE-2017-7659
Jetty	8.1.14	8080	CVE-2023-40167
vsftpd	2.3.4	21	CVE-2011-2523
Database Server			
PostgreSQL DB	10.5	5432	CVE-2019-9193
MySQL	5.5.23	3308	CVE-2012-2122
OpenSSH	8.9p1	22	CVE-2023-28531
Redis	5.0.7	6379	CVE-2022-0543

When selecting the software, care was taken to ensure that it was widely used and standard software. The selected software is described below. The CVE records registered for this purpose, as well as the version and port used on the corresponding server, are listed in Table 26. The various software versions cover older vulnerabilities from 2012 up to current vulnerabilities from 2024.

### Drupal

An open-source content management system (CMS), ideal for complex and customizable websites. It supports a wide range of web content such as texts, blogs,

surveys, and image galleries.<sup>31</sup>

### **Jenkins**

An automated continuous integration tool that helps developers automate tasks such as building, testing, and deploying software.<sup>32</sup>

### **JetBrains TeamCity**

A continuous integration and continuous delivery (CI/CD) server that supports the automation of build and test processes.<sup>33</sup>

### **Confluence**

A platform for team collaboration and knowledge management that facilitates the sharing and editing of documents within teams.<sup>34</sup>

### **Joomla**

A user-friendly content management system (CMS) is used to create websites and online applications.<sup>35</sup>

### **Grafana**

An open-source platform for monitoring and analytics, providing visualization and alerting for web applications.<sup>36</sup>

### **GitLab**

A complete DevOps platform, delivered as a single application, enabling the management of projects and the automation of workflows in the software development process.<sup>37</sup>

### **phpMyAdmin**

A free software tool for managing MySQL over the web. It allows the creation and editing of databases, tables, and content.<sup>38</sup>

### **Elasticsearch**

A search and analytics engine that provides fast and powerful search capabilities for large datasets.<sup>39</sup>

### **Oracle GlassFish**

An open-source application server that supports the Java EE platform, allowing de-

---

<sup>31</sup><https://www.drupal.org>

<sup>32</sup><https://www.jenkins.io>

<sup>33</sup><https://www.jetbrains.com/teamcity/>

<sup>34</sup><https://www.atlassian.com/software/confluence>

<sup>35</sup><https://www.joomla.org>

<sup>36</sup><https://www.grafana.com>

<sup>37</sup><https://www.gitlab.com>

<sup>38</sup><https://www.phpmyadmin.net>

<sup>39</sup><https://www.elastic.co/elasticsearch/>

velopers to create and deploy enterprise applications.<sup>40</sup>

### **PHP**

An open-source scripting language particularly builded for web development and can be embedded into HTML.<sup>41</sup>

### **Apache HTTP Server**

An open-source web server that provides a variety of features and extensions, being one of the most used web servers worldwide.<sup>42</sup>

### **jQuery**

A fast, compact, and feature-rich JavaScript library that streamlines HTML document traversal and manipulation, event handling, animation, and Ajax.<sup>43</sup>

### **Apache Tomcat**

An open-source platform implementing Java Servlet, JavaServer Pages, Java Expression Language, and Java WebSocket technologies.<sup>44</sup>

### **Postfix**

A mail transfer agent (MTA) that can send and receive emails on a computer and is often used as an alternative to Sendmail.<sup>45</sup>

### **OpenSSH**

An open-source tool that provides secure encryptions for remote access protocols, enabling secure communication over an unsecured network.<sup>46</sup>

### **Apache httpd**

A modular web server is known for its flexibility and extensive support for various operating systems.<sup>47</sup>

### **Jetty**

A lightweight, Java-based web server and servlet container used in both large and small applications.<sup>48</sup>

### **vsftpd**

A very secure and fast FTP server for Unix-based systems, known for its reliability

---

<sup>40</sup><https://www.oracle.com>

<sup>41</sup><https://www.php.net>

<sup>42</sup><https://httpd.apache.org>

<sup>43</sup><https://jquery.com>

<sup>44</sup><https://tomcat.apache.org>

<sup>45</sup><http://www.postfix.org>

<sup>46</sup><https://www.openssh.com>

<sup>47</sup><https://httpd.apache.org>

<sup>48</sup><https://www.eclipse.org/jetty>

and low resource requirements.<sup>49</sup>

### PostgreSQL

An open-source object-relational database management system that offers a wide range of advanced data processing features.<sup>50</sup>

### MySQL

A widely-used open-source database management system.<sup>51</sup>

### Redis

An open-source, in-memory data structure store utilized as a message broker database and cache.<sup>52</sup>

The installation of a Docker image from the Vulhub repository is shown here as an example. To do this, the entire repository was first downloaded and, in this case, the corresponding Docker image was started from the corresponding CVE folder of Drupal using the command `docker-compose up -d` (see listing 21).

Listing 21: Installing Docker Image from vulhub repo

```
1 wget https://github.com/vulhub/vulhub/archive/master.zip -O vulhub-master.zip
2 unzip vulhub-master.zip
3 cd vulhub-master
4 cd drupal
5 cd CVE-2018-7602
6 docker compose up -d
```

To answer the research question of how the efficiency of an LLM for detecting vulnerabilities in IT networks compares to that of established vulnerability assessment tools, a network scan with Nessus is performed on the same network during the experiment, as well as a run with the framework.

For this purpose, the accessible version of Nessus, which can be used for up to 12 IP addresses, is installed on the router within the network. The version installed is 10.7.2. During the installation, care was taken to use the latest version of Nessus at the time of the experiment in April 2024 to ensure that the internal knowledge database contains the most up-to-date vulnerability databases. For the Nessus installation, an SSH connection to the router DMZ was established, and Nessus was downloaded, installed, and started as a service according to Listing 22. Nessus must be used via the web portal, which is available by default on port 8834 after installation. Since the router DMZ is connected to the host network via the network interface eth0, Nessus can be accessed via the IP 192.168.178.164 and the corresponding port. The actual execution and start of a network scan via Nessus are described in more detail in the following chapter.

<sup>49</sup><https://security.appspot.com/vsftpd>

<sup>50</sup><https://www.postgresql.org>

<sup>51</sup><https://www.mysql.com>

<sup>52</sup><https://redis.io>

## Listing 22: Installing Nessus

```
1 dpkg -i Nessus-10.7.2-debian6_amd64.deb
2 systemctl start nessusd
```

The Nmap scan used in the execution was designed in advance as in listing 23. Care was taken to ensure that a particular depth of scan intensity was used to obtain as much information as possible about a computer. The parameters used for the scan are listed below:

- `nmap`: The Network Mapper tool for scanning networks.
- `-sV`: Version detection - Identifies the versions of running services.
- `-script "http-title,http-headers,http-methods,http-enum"`: Executes specific NSE scripts:
  - `http-title`: Retrieves the title of the webpage.
  - `http-headers`: Displays the HTTP headers.
  - `http-methods`: Determines the supported HTTP methods.
  - `http-enum`: Performs a simple enumeration of HTTP services.
- `-T4`: Sets the timing template to "Aggressive" to speed up the scan.
- `192.168.1.12`: Target IP address of the scan.
- `-p-`: Scans all 65535 ports.

## Listing 23: nmap scan command

```
1 nmap -sV --script "http-title,http-headers,http-methods,http-enum" -T4 192.168.1.12 -p-
```

During the preparation and execution of the experiment, it was discovered that there is a possibility that software does not directly display its version during a Nmap scan, even when using an intensive Nmap command. The Nmap scan command was executed on a test example and detected an open port on port 8112. The expected software there is TeamCity, but "unknown" is displayed because the web server running in the Docker image does not provide the information. Also, using the Nmap script to read out the HTTP headers or the HTTP title was not successful in this case. This can be due to various reasons, such as the main page on the port containing a redirect that Nmap does not resolve, thus making the required information unavailable.

To solve this problem, an attempt was made to retrieve the web page content using `curl`. `curl` is a command-line utility for data transfer that utilizes URL syntax.

When `curl` is called with the `-L` parameter, it follows the HTTP redirect, and the web page data of the target page can be displayed. At the time of writing, no Nmap script was available to perform this option. For statically processed network scans, the website content offers less added value, as information may be present in an unstructured manner within the HTML source code. These are difficult to recognize using static analysis, as the result for a query is not available at this point, and the search is therefore made more difficult, as it is not clear which keywords, such as the software name, should be searched for.

For processing via LLMs, which master text comprehension through NLP concepts, this information can be used advantageously for this experiment to obtain missing information in addition to the Nmap scan. In the test in the preliminary phase, the unambiguous version of the software could thus be defined from the context. Deeper insights into the results will be explained in the following chapters.

To use `curl` within the Nmap scan, a custom Nmap script was developed (see listing 24), which executes a `curl -L` command for each open port and adds the result to the Nmap scan result. The script for this was developed in Lua, a lightweight embedded scripting language used by Nmap for internal scripts. This approach has proven to be positive in several tests in the preliminary phase.

Listing 24: Nmap script for curl

```
1  local stdnse = require "stdnse"
2  local shortport = require "shortport"
3
4  description = [[
5  Perform an HTTP GET request on the specified port, following redirects using curl.
6  ]]
7
8  author = "Oliver Dzaeck"
9  license = "Same as Nmap--See https://nmap.org/book/man-legal.html"
10 categories = {"discovery", "safe"}
11
12 portrule = shortport.http
13
14 action = function(host, port)
15   local path = "/"
16   local ip = host.ip
17   local port_number = port.number
18   local command = "curl -L http://" .. ip .. ":" .. port_number .. path
19
20   -- Execute the curl command and capture the output
21   local handle = io.popen(command)
22   local result = handle:read("*a")
23   handle:close()
24
25   return result
26 end
```

### 3.6.3 Experiment Execution

The static analysis is carried out as described with Nessus. Nessus was installed on the router DMZ to do this, as described in the previous section. The analysis was carried out via the Nessus application interface. To do this, the address

`http://routermaster:8834` was called up from the host system.

To scan the entire network, the Basic Network Scan was selected as the scan template, and the IP addresses of the computers in the subnet were entered as parameters (192.168.1.10, 192.168.1.12, 192.168.1.11). After the network scan was completed, a report was created. The export should be in CSV format so that the results can be processed more efficiently. The fields that are crucial for this experiment were selected for export. These include the CVE entry, the CVSS v3.0 Base Score (the CVSS v3.1 Base Score was not yet available in Nessus at the time this work was written), the risk, the host, the protocol, the port, the name and the description of the vulnerability. Since Nessus also outputs general information about the scanned ports in the report, the CSV report file was filtered only for entries that contain a CVE entry and thus explicitly recognized and assigned a vulnerability. Furthermore, only entries associated with a valid port were filtered. Thus, all entries containing port 0 were not included in the analysis and the experiment. When examining the execution, further vulnerabilities were found in system components. However, these were not included in the analysis because the focus was on the detection of vulnerable software.

The developed framework (AISecNet) was also developed in Python. The code parts were reused from the other experiments and extended by the corresponding logic for the framework. The framework comprises several steps that are executed during a run of the process. The process is shown in Figure 9. The process begins with an initial scan of the network using a Nmap scan, which is executed as a subprocess in Python. The Nmap command described above is used, supplemented by the previously mentioned scripts and the self-developed Nmap script, to execute the Curl command on the software systems.

The result of this process is a comprehensive Nmap report. Such a report, depending on the number of installed software, contains, on average, over 2000 lines, as observed, and was not included in this document due to its length. The report is structured like the Nmap example from section 2.8 and contains the web page content of the page that was retrieved using Curl.

This large unstructured text is then passed to a purely pre-trained GPT-4-Turbo model within the framework to extract the relevant information from the text. For this purpose, a prompt (see 25) was written that instructs the model to extract the relevant information about possible software and versions from the context passed to it. The model is instructed to return the information in a specific JSON format. The software and version, as well as the port on which the software is installed, are to be used for further processing. The tokens used for this are also stored.

Listing 25: Prompt for text summarizing

```

1  "content": (
2  f"The context is an extended nmap scan. ToDo:\n"
3  f" - Find software, websites, webtechnologies, script language (like php, jquery, javascript, drupal,
4  joomla), software libraries etc. and the corresponding version in the given context of the nmap scan.\n"
5  f" - If a version is not directly recognizable, take a closer look at the corresponding output related to
the software and try to deduce the version from it.\n"
6  f" - Check the nmap scan several times for matches.\n"
7  f"Your answer should be look like in this json example:\n"
8  f"f'{{'Results': [\n"
9  f"  f'{{'software': 'jquery.js', 'port': 8080, 'version': '2.3.4'}}, '\n"
10 f"  f'{{'software': 'PHP', 'port': 22, 'version': '3.4'}}'\n"
11 f"  f']}}'. This is the nmap scan output:\n{nmapscan}"

```

The return is processed within the framework using an iterative method in which each entry in the last result list is passed to another model. The model used is the one that proved to be the more performant in Experiment 1 (GPT-4-Turbo with Azure AI Search). The software and version are embedded in a prompt (see Listing 26) and passed to the model as input. The model is defined to find corresponding CVE entries and exploits and to generate them in a predefined JSON format as a return. The output is stored in a result file and recorded for further performance measurement in comparison to Nessus. The tokens used are also stored. The expected result list, therefore, includes the software and version, as well as the corresponding CVE record and, if possible, other CVE records that also exist for this software.



Listing 26: Prompt for cve identfiy

```
1  "content": (  
2  f"""  
3  Find possible vulnerabilities in this software.  
4  List the corresponding CVE entries in a JSON with description and also exploits if there are any.  
5  Do not give me any information about software if the version is not directly affected by the vulnerability.  
6  
7  
8  Name: {name} Version: {version}  
9  
10 The output should be in this JSON format:  
11 {{  
12   "name": "{name} {version}",  
13   "software": [  
14     {{  
15       "cve_entry": {{  
16         "number": "CVE-2023-0001",  
17         "description": "Description of the vulnerability 1"  
18       }}  
19     }},  
20     {{  
21       "cve_entry": {{  
22         "number": "CVE-2023-0002",  
23         "description": "Description of the vulnerability 2"  
24       }}  
25     }},  
26     {{  
27       "cve_entry": {{  
28         "number": "CVE-2023-0003",  
29         "description": "Description of the vulnerability 3"  
30       }}  
31     }}  
32   ]  
33 }}
```

Subsequently, an assessment of the system's level of danger is made on the basis of Experiment 2. Therefore, the same methods are used to determine the CVSS score based on a description of a vulnerability. A CVSS vector string is determined using the GPT-3 model with Azure AI Search. This CVSS vector string is broken down into individual metrics and then calculated using the calculation explained in chapter 2.6. The result is also appended to the result file, as are the tokens generated by the call.

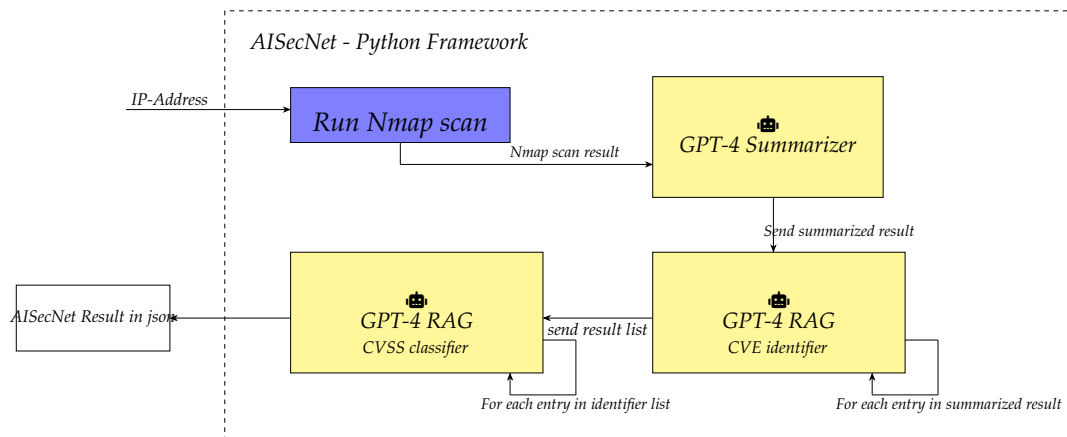


Figure 9: ASecNet Workflow

This process is run for each computer in the subnet. The computer with the IP 192.168.1.10 (application server) was run first, then the computer with the IP 192.168.1.11 (database server), and finally, the computer with the IP 192.168.1.12 (webserver). The result is a list of results for each IP in the network. A section of a result list is shown in listing 27. The output parameters in the file are as follows:

**software\_name** This is the name of the software to which the CVE entry pertains.

**number** This is the unique identifier of the CVE entry.

**cve\_description** This is the description of the vulnerability.

**software\_version** This is the version of the software that is affected by the vulnerability.

**cvss\_vector** This is the CVSS vector, which describes the characteristics and impact of the vulnerability.

**base\_score** This is the CVSS's base score, indicating the severity of the vulnerability on a scale from 0 to 10.

**severity** This is the vulnerability's severity rating (e.g., low, medium, high, critical), which is calculated based on the base score.

Listing 27: Example for the result output from the AISEcNet Framework

```
1 {
2   "number": "CVE-2024-23897",
3   "description": "Jenkins 2.441 and earlier does not disable a feature of its CLI command parser that
4     replaces an '@' character followed by a file path in an argument with the file's contents, allowing
5     unauthenticated attackers to read arbitrary files on the Jenkins controller file system.",
6   "software_name": "Jenkins 2.441",
7   "software_version": "2.441",
8   "cvss_vector": "AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N",
9   "base_score": 7.5,
10  "severity": "High"
11 },
```

After the scan has been completed, the results list contains all the recognized vulnerabilities, including the CVSS data, so that they can be compared with the reports generated by Nessus during the evaluation.

### 3.6.4 Measurement and Evaluation

The results of the Nessus scan were exported in the form of a CSV file, as described in the previous chapter. The filtering described above was applied to the data in order to extract only the relevant data. Care was taken to ensure that only software was taken into account. Vulnerabilities were also identified that are directly related to the connection to the system configuration of the scanned computers. For example, a vulnerability was found on the system with IP 192.168.1.12, which describes that IP forwarding is enabled. This aspect is a valuable insight for a comprehensive analysis of the vulnerabilities of a computer system. However, these vulnerabilities were filtered out for this work and experiment, as the developed framework is not designed to identify these vulnerabilities based on the various models, and their inclusion would create a less clear picture. However, it would be a valuable addition to further research approaches and the further development of the framework to classify the system even better.

In this experiment, the measurement was controlled manually and not automatically by scripts. This makes it possible to look at each individual affected vulnerability in more detail, based on the small data basis, in order to obtain a comparison between the static detection tools and the detection of the vulnerabilities via the framework.

The CSV report file from Nessus was compared with the software installed on the computer and the known vulnerabilities. A further CSV file was then used to define which software was recognized by Nessus and which was not. After careful examination, a note was made to indicate whether the vulnerabilities were recognized.

The following procedure was carried out manually for all entries. For clarity, the following is a description of the procedure.

The CSV file is filtered for the IP 192.168.1.12. The first entry found by Nessus is ElasticSearch on port 9200. The first thing to check is whether Nessus has recognized the correct version. In the case of ElasticSearch, Nessus was able to identify version 1.6.0 successfully. The result is recorded in a CSV file. The CVE entries found are then checked for correctness. Nessus has recognized CVE-2015-5377 with a CVSS v3.0 Base Score of 9.8. The CVE entry was checked manually using the official CVE database, as it does not match the CVE entry that was stored for the Docker image for ElasticSearch. The description of the recognized CVE reads: “Elasticsearch before 1.6.1 allows remote attackers to execute arbitrary code via unspecified vectors involving the transport protocol. NOTE: ZDI appears to claim that CVE-2015-3253 and CVE-2015-5377 are the same vulnerability”. This implies that the installed version 1.6.0 is also affected by the vulnerability. This finding is also recorded in the results CVE file.

Further testing for this example revealed that Nessus was unable to identify the installed phpMyAdmin software, even though phpMyAdmin version 4.8.1 was installed on the computer and included the CVE-2018-12613 vulnerability. This result is also recorded in the results CSV. This process was then carried out with each software program on each computer in the subnet. The structure of the results list is shown in Table 27 as an example.

Software	Version	CVE	Status
ElasticSearch	1.6.0	CVE-2015-5531	Detected OK
		Name CVE Detected	NOK
		Version detected	OK
phpMyAdmin	4.8.1	CVE-2015-5377	OK
		CVE-2018-12613	Detected NOK
		Name CVE Detected	NOK
		Version detected	NOK

Table 27: CVE Detection Status

The results of the model were manually extracted from the results list described in the last chapter and compared with the results of Nessus. For this purpose, they were entered in a comparison file, an Excel sheet, for validation. In the Excel file, a list of correctly installed software versions and CVE records reported via the Docker image was first created. The structure is shown in Table 27.

The accuracy of the results was also measured by comparing the recognized CVSS severity levels, the base score, and the CVSS vector string. A module within the framework was explicitly used to perform this classification. The vector string, the base score, and the severity level were also manually compared with the level en-

tered in the CVE records.

Another important aspect of the framework's evaluation is the cost analysis. If the framework is to be used in a productive environment or similar environment, this is a decisive factor for usability.

### **3.6.5 Results Analysis**

In this chapter, the results of experiment 3 are considered and discussed. The aim was to find out whether LLMs can be used in any way to detect vulnerabilities within IT networks. To confirm these hypotheses and assumptions, experiments were carried out, and a vulnerability detection framework was developed, as described in the last chapter. The results were recorded and prepared for evaluation. The results were analyzed manually and are evaluated in this chapter for each computer from the simulation environment. The aim was to compare which framework, Nessus or AiSecNet, was able to detect the most vulnerabilities. Based on these results, an evaluation of the performance was carried out.

It is important to emphasize that the data basis represents a small data basis with a total of 22 software products that were installed in the simulation environment. In order to obtain a deeper insight and more precise results, the experiment would have to be applied to a more extensive data basis. Due to the difficulties described in the other chapter with a mass test with, for example, Nessus, the framework had to be kept small. However, this provides an opportunity to assess the small database better and to develop a more detailed understanding of the results.

Furthermore, it must be mentioned that the AiSecNet framework, in contrast to Nessus, had to be carried out several times, as the results varied in part, and the prompt had to be adapted again and again. It must also be mentioned that only Nessus was selected for evaluation for this experiment. Due to the complexity of the other parts of this work, only one assessment tool was used for evaluation. Attempts were made with Nmap's internal vulnerability detection tool, but the results were not satisfactory, so they were not included in the analysis.

In order to get a bigger picture of how LLMs could achieve better results in a larger context using assessment tools, the experiments should be carried out using other assessment tools. Nevertheless, the results will provide added value based on the research details to assess how well LLMs could be used in cybersecurity and whether they can achieve better results than joint assessment tools.

The results include the described results of the detection of the software compared to Nessus. Furthermore, the results considered the extent to which the AiSecNet framework can classify the dangers of software using the CVSS score. The cost

analysis for a run per computer using the AiSecNet framework is also included in the analysis, as it provides a decisive statement on the economic implementation of a framework under the current conditions of the price models of the models.

**Application Server (192.168.1.12)** The first analysis of the results was carried out on a computer loaded with various vulnerable software as an application server in the simulation environment.

Surprisingly, Nessus was able to detect 3 out of 11 vulnerable software types installed and correctly assign a vulnerability. Nessus was also able to identify vsftpd and Postfix with the correct version but could not assign a vulnerability, according to the report. All other software types, except for the correctly identified ones, could not be recognized as to which software was running on the scanned port. This results in a pure detection rate of the software, with or without vulnerability, of 45.45%. The results from Table 28 show that Nessus was able to correctly detect, identify, and classify the vulnerabilities in Elasticsearch, Oracle Glassfish, and Apache Tomcat. In this small-scale experiment, Nessus was able to achieve a vulnerability detection rate of 27.27% when scanning the application server.

To check which data Nessus should at least be aware of, an extended Nmap scan was carried out on the application server using the command: `nmap 192.168.1.12 -sV -p-`. The parameters `-sV` ensure that the software version is specifically searched for, and the parameter `-p-` ensures that all ports are scanned. It was noticeable that the Jetty web server and OpenSSH should also have been identified by a purely static search (see listing 28).

Listing 28: Extended nmap scan on 192.168.1.12

PORT	STATE	SERVICE	VERSION
21/tcp	open	ftp	vsftpd 2.3.4
22/tcp	open	ssh	OpenSSH 8.9p1 Ubuntu 3ubuntu0.6 (Ubuntu Linux; protocol 2.0)
25/tcp	open	smtp	Postfix smtpd
4848/tcp	open	ssl/http	Oracle GlassFish 4.1 (Servlet 3.1; JSP 2.3; Java 1.8)
8070/tcp	open	http	Apache httpd 2.4.25 ((Debian))
8080/tcp	open	http	Jetty 8.1.14.v20131031
8088/tcp	open	http	Apache Tomcat 9.0.30
8092/tcp	open	http	nginx
8112/tcp	open	unknown	
9200/tcp	open	http	Elasticsearch REST API 1.6.0 (name: Frederick Slade; cluster: elasticsearch; Lucene 4.10.4)
9300/tcp	open	elastic search	Elasticsearch binary API

As shown in Table 28, AiSecNet was able to correctly identify 10 out of 11 installed software types as vulnerable. Each detected software was correctly identified as vulnerable software, as well as in the appropriate CVE entry and the correct version. For the correctly detected software, the installed TeamCity version could be marked as vulnerable, but the correct CVE entry could not be recognized.

Neither Nessus nor AiSecNet were able to identify GitLab correctly.

Based on the context recognition of the Nmap scan from listing 28, which was also part of the AISecNet process in an extended version, AISecNet was able to identify vsftp, OpenSSH, Oracle GlassFish, Apache HTTPD, Jetty, Apache Tomcat and Elasticsearch. GitLab, TeamCity, and phpMyAdmin were not included in the Nmap scan result. The contextual information that the model used to obtain information about the software was extracted from the extension via the curl script. During the manual review, TeamCity and the version were found in the extracted HTML content of the curl result obtained, as were phpMyAdmin and the version. GitLab was named, but it did not provide a direct reference to a software version.

In a direct comparison between the assessment tool and the AISecNet framework, the framework, which uses models specially adapted to the cybersecurity domain, was able to deliver a 233% better result in this case than Nessus.

192.168.1.12			Nessus			AISecNet		
Software	Ver.	CVE	Vul. Det.	CVE Det.	Ver. Det.	Vul. Det.	CVE Det.	Ver. Det.
ElasticSearch	1.6.0	CVE-2015-5531	OK	OK	OK	OK	OK	OK
phpMyAdmin	4.8.1	CVE-2018-12613	NOK	NOK	NOK	OK	OK	OK
Apache httpd	2.4.25	CVE-2017-7659	NOK	NOK	NOK	OK	OK	OK
Jetty	8.1.14	CVE-2023-40167	NOK	NOK	NOK	OK	OK	OK
Oracle Glassfish	4.1.0	CVE-2017-1000028	OK	OK	OK	OK	OK	OK
Apache Tomcat	9.0.39	CVE-2020-1935	OK	OK	OK	OK	OK	OK
Gitlab	13.10.1	CVE-2021-22205	NOK	NOK	NOK	NOK	NOK	NOK
vsftp	2.3.4	CVE-2011-2523	NOK	NOK	OK	OK	OK	OK
TeamCity	2023.05.03	CVE-2023-42793	NOK	NOK	NOK	OK	NOK	OK
postfix	3.7.11	CVE-2023-51764	NOK	NOK	OK	OK	OK	OK
OpenSSH	8.9p1	CVE-2023-28531	NOK	NOK	NOK	OK	OK	OK

Table 28: Vulnerability Detection Comparison

Nessus did not incur any extra costs during execution except for the operation of computer power.

AISecNet caused costs of 1.53\$ through the use of the models and the increased number of tokens through the use of the context from the RAG. The consumed tokens per model are listed by input and output tokens in table 29.

Modell	Prompt-Tokens	Completion-Tokens	Gesamtkosten (\$)
GPT-4 Turbo	149,308	5834	1.49
GPT-3.5 Turbo	28595	510	0.02
Total Cost (\$):			1.51

Table 29: Tokenanzahl und Kosten für GPT-Modelle

In the area of classifying vulnerabilities via the CVSS metrics, AISecNet was only able to generate 2 out of a possible 11 vector strings from the CVE description, as can be seen in Table 30. One vector string was generated for Apache Tomcat, which was flagged as correct after checking the NVD database. The other vector string was generated for Oracle Glassfish. The generated vector string for Oracle Glassfish

was not correct and was not categorized as "Critical." The vector string stored in the NVD database has a calculated base score of 7.5 and is therefore assigned to the severity level "High." Overall, the model performed poorly in this example scenario. It was striking that the model returned "No match found" for all other weak points. This would suggest that the model tried to include the vector string via the RAG in these cases and, when this was not found, did not look at the CVE description. The result obtained thus contradicts the results obtained in the second experiment. The same setup and the same prompt were used for this experiment as for experiment 2. No further work will be done on this result in this paper, but a revision of the prompt could lead to better results.

Product	CVSS Vector	RAG Status	CVSS Score	Severity	Correctly Classified
Apache HTTP	No match found				
ElasticSearch	No match found				
phpMyAdmin	No match found				
Apache httpd	No match found				
Jetty	No match found				
Oracle Glassfish	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N	Not in RAG	9.1	Critical	No
Apache Tomcat	AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:L/A:N	Not in RAG	5.3	Medium	Yes
vsftp	No match found				
TeamCity	No match found				
postfix	No match found				
OpenSSH	No match found				

Table 30: Security ratings of software products from application server 192.168.1.12

**Database Server (192.168.1.11)** The next step in our thorough assessment was to examine the database server, which was also installed in the simulation environment with vulnerable software. Our focus here was purely on database systems, ensuring a comprehensive evaluation of their vulnerability.

Regrettably, Nessus was unable to identify any of the 3 installed database systems with the correct version, even after repeated scans. This critical gap in vulnerability detection is evident in the fact that although the database systems were recognized in the Nessus report, the corresponding version of the database system was not. The listing 29 shows the Nmap scan that was also carried out for the computer. It can be seen that MySQL was not recognized with a version number, but a version was already available directly in the Nmap scan for the PostgreSQL DB and Redis. This results in a 100

Listing 29: Extended nmap scan on 192.168.1.11

1	PORT	STATE	SERVICE	VERSION
2	22/tcp	open	ssh	OpenSSH 8.9p1 Ubuntu 3ubuntu0.6 (Ubuntu Linux; protocol 2.0)
3	3308/tcp	open	mysql	MySQL
4	5432/tcp	open	postgresql	PostgreSQL DB 10.2 - 10.5
5	6379/tcp	open	redis	Redis key-value store 5.0.7
6	Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel			
7				



In comparison, AISecNet was able to identify all three database systems and OpenSSH with a 100% detection rate, both based on the appropriate CVE records and on the version. During the manual review, it was recognized that the MySQL version was available via the HTML content, and therefore, it is assumed that the framework was able to obtain the version from the context. This was also the case for the other two database systems, as these were also directly available in the Nmap result context.

The comparison of the two systems is listed in table 31. As already described, it can be seen that Nessus was unable to detect any of the four vulnerabilities. AISecNet, on the other hand, was able to identify all vulnerabilities perfectly.

192.168.1.11			Nessus			AISecNet		
Software	Ver.	CVE	Vul. Det.	CVE Det.	Ver. Det.	Vul. Det.	CVE Det.	Ver. Det.
PostgreSQL	10.5	CVE-2019-9193	NOK	NOK	NOK	OK	OK	OK
MySQL	5.5.23	CVE-2012-2122	NOK	NOK	NOK	OK	OK	OK
Redis	5.0.7	CVE-2022-0543	NOK	NOK	NOK	OK	OK	OK
OpenSSH	8.9p1	CVE-2023-28531	NOK	NOK	NOK	OK	OK	OK

Table 31: Vulnerability Detection Comparison for 192.168.1.11

As with the other servers, the Nessus scan is free of charge, except for the resources used to run Nessus. AiSecNet incurred costs of 0.34\$ for a single run. The individual cost breakdowns, broken down by the models used, are listed in Table 32.

Modell	Anzahl der Prompt-Tokens	Anzahl der Completion-Tokens	Gesamtkosten (\$)
GPT-4 Turbo	29802	1088	0.33
GPT-3.5 Turbo	11604	144	0.012
Total Cost (\$):			0.34

Table 32: Tokenanzahl und Kosten für GPT-Modelle

When considering the classification, AISecNet was able to correctly classify 75% of all vulnerabilities within the CVSS scoring system. It is noticeable that in this experiment, the CVSS vector string for OpenSSH was correctly classified using the CVE description. This was not the case for the application server. The "No match found" result also occurred when classifying MySQL. This leads to the assumption that, compared to experiment 1, the success rate is high on a more significant amount of data but that the model works very inconsistently in a framework that was not measured in this study. The classification of the recognized vulnerabilities according to a CVE description gives excellent results; all generated vector strings were identical to the vector strings stored in the NVD database, except for minor differences. For example, the vector string for the PostgreSQL vulnerability was "CVSS:3.0/AV:N/AC:L/PR:H/UI:N/S:U/C:H/I:H/A:H" with a base score of 7.2 and a severity level of HIGH according to the NVD database. In this case, the model had rated the vector metric for privileges requirements as LOW, while the actual NVD rating for privileges requirements was HIGH.

Product	CVSS Vector	RAG Status	CVSS Score	Severity	Correctly Classified
PostgreSQL	AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H	Not in RAG	8.8	High	Yes
MySQL	No match found				
Redis	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H	Not in RAG	9.8	Critical	Yes
OpenSSH	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H	Not in RAG	9.8	Critical	Yes

Table 33: Security ratings of software products from web server 192.168.1.11

**Web Server (192.168.1.10)** The final analysis of the results is carried out on the computer that was loaded as a web server in the simulation environment with various vulnerable software types.

Nessus was able to detect 3 out of 8 installed vulnerable software types and correctly assign a vulnerability. It was able to correctly identify PHP, Apache HTTP, and an outdated version of jQuery as vulnerable. Nessus was also able to identify Grafana and the installed version but not the vulnerability associated with the Grafana version. Nessus did not identify any other web applications (see table 34). This corresponds to a detection rate of vulnerable software of 37.5%.

192.168.1.10			Nessus			AISecNet		
Software	Ver.	CVE	Vul. Det.	CVE Det.	Ver. Det.	Vul. Det.	CVE Det.	Ver. Det.
PHP	7.0.28	CVE-2018-10546	OK	OK	OK	OK	OK	OK
Apache HTTP	2.4.54	CVE-2006-20001	OK	OK	OK	OK	OK	OK
Grafana	8.2.6	CVE-2023-22527	NOK	NOK	OK	OK	OK	OK
Drupal	7.57	CVE-2018-7602	NOK	NOK	NOK	NOK	NOK	NOK
Jenkins	2.441	CVE-2024-23897	NOK	NOK	NOK	OK	OK	OK
Joomla	4.2.7	CVE-2023-23752	NOK	NOK	NOK	OK	OK	OK
Confluence	8.5.3	CVE-2023-22527	NOK	NOK	NOK	NOK	NOK	NOK
jquery	1.4.4	CVE-2020-11022	OK	OK	OK	OK	OK	OK

Table 34: Vulnerability Detection Comparison

To check the results, an extended Nmap scan was carried out on this server with the IP 192.168.1.10. The result is shown in listing 30. The scan result shows that none of the installed software is mentioned explicitly in the scan result for the web server. Therefore, in this case, it can only be assumed that this information had to be extracted for the AiSecNet framework by using various Nmap scripts and the Curl script.

The scan analyses that Nessus performs show that the detection of software is challenging here. The extended Nmap scan result shows that without the use of scripts, the software listed by the Docker images with installed web servers is not directly identified by name. No further information is displayed for port 3000. Grafana is installed on this port. Nessus successfully identified Grafana there using the internal algorithms and analyses used by Nessus and assigned the appropriate version, but Nessus did not recognize the vulnerability that existed there.

Listing 30: Extended nmap scan on 192.168.1.10

PORT	STATE	SERVICE	VERSION
22/tcp	open	ssh	OpenSSH 8.9p1 Ubuntu 3ubuntu0.6 (Ubuntu Linux; protocol 2.0)
80/tcp	open	http	Apache httpd 2.4.52 ((Ubuntu))
3000/tcp	open	php?	
8080/tcp	open	http	Apache httpd 2.4.54 ((Debian))
8081/tcp	open	http	Apache httpd 2.4.10 ((Debian))
8084/tcp	open	http	Apache httpd 2.4.52
8088/tcp	open	http	nginx 1.14.2
8089/tcp	open	http	nginx 1.14.2

In this experiment, AISecNet correctly identified and classified 6 out of 8 vulnerable software types on the web server. This corresponds to a detection rate of 75% for vulnerable software. The two software types that were not recognized, Drupal and Confluence, were correctly recognized by AISecNet, but no correct version could be assigned. The pure detection rate of the installed software is 100%. AISecNet did recognize the major release number 7, but not the corresponding sub-version.

In a performance comparison for the detection of vulnerabilities, AISecNet showed a measurably better performance.

Nessus did not incur any additional costs during execution except for the operation of the computer performance.

AISecNet caused costs of 1.53\$ by using the models and the increased number of tokens by using the context from the tag. The tokens used per model are listed in Table 35 according to input and output tokens.

Model	Number of Prompt Tokens	Number of Completion Tokens	Total Cost (\$)
GPT-4 Turbo	126707	4938	1.41
GPT-3.5 Turbo	75034	1235	0.12
Total Cost (\$):			1.53

Table 35: Token counts and costs for GPT models based on the measurement of the web server 192.168.1.10.

When classifying the vulnerabilities according to the CVSS scoring system, AISecNet created a CVSS vector string for all recognized vulnerabilities. The model was able to generate vector strings for Grafana, Jenkins, PHP, Joomla, and Apache HTTP. A closer examination of the CVSS vector strings known to the model revealed that a classification of the CVSS vector strings was available in the vector database for Grafana, as well as for the jQuery vulnerability and the PHP vulnerability. It can be assumed that the information was obtained directly from the RAG. The model generated the other CVSS vector strings by interpreting the CVE description. Despite the presence of the vector string for the PHP vulnerability, it was not correctly classified by the framework. The comparison was carried out using the NVD database. This contained a CVSS vector string for these vulnerabilities. The CVE database used by MITRE did not contain this score classification, which can be explained by

the lack of correspondence between the various vulnerability databases from Chapter 2.6. In the comparison, the generated vector string was compared with the vector string on the NVD database for the vulnerability. A "No match found" result, as in the classification of the vulnerabilities in the application server and database server, did not occur when checking the web server.

Product	CVSS Vector	RAG Status	CVSS Score	Severity	Correctly Classified
Apache HTTP	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H	Not in RAG	9.8	Critical	No
Grafana	AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:N/A:N	Is in RAG	4.3	Medium	Yes
Jenkins	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H	Not in RAG	9.8	Critical	Yes
Joomla	AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N	Not in RAG	5.3	Medium	Yes
jquery	AV:N/AC:H/PR:N/UI:R/S:C/C:H/I:L/A:N	Is in RAG	6.9	Medium	Yes
PHP	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H	Is in RAG	9.8	Critical	No

Table 36: Security ratings of software products from web server 192.168.1.10

Overall, AISecNet achieved a higher success rate for the web server than Nessus, with AISecNet detecting 6 out of 8 vulnerabilities compared to Nessus's 3 out of 8. In terms of cost, the AISecNet framework is costly, as a single scan costs 1.53\$, in contrast to the unrestricted use of Nessus. In the classification of vulnerabilities, AISecNet performed in the middle range. Of a total of 6 vulnerabilities, four were correctly identified, and two were incorrectly identified, resulting in a detection rate of 66.67%.

**Summary** In this experiment, the performance and efficiency of two security tools were examined: Nessus and AISecNet. Nessus is an established vulnerability detection tool, while AISecNet is a framework that was developed on the basis of other experiments to investigate various software and server configurations in a simulated environment. The approaches from Experiment 1 and Experiment 2 were taken up, and the results of both experiments regarding the performance of the best model variants were used as a basis for developing the AISecNet framework.

The analysis focused on the aspects of detection rates, the cost of classification accuracy, and the overall performance in identifying and assessing vulnerabilities in the server setup. The investigation aimed to conduct an evaluation and to discuss and answer the research questions. The strengths and weaknesses of both tools were measured and compared. An evaluation of the respective results per server was carried out in the corresponding sections of the servers.

During the experiment, various hypotheses were formulated. These were intended to answer the question of whether models with an extensive knowledge base of vulnerability data can interpret scan reports from network scans and identify and classify affected software versions. The study also investigated whether powerful models can identify vulnerabilities in network scans, assign them to existing threats, and determine the level of risk. It also investigated whether models that have been

specially trained in the field of cybersecurity achieve similar or better results than conventional evaluation tools.

The first hypothesis was that models with an extensive knowledge base of vulnerability data could interpret scan reports from network scans and identify and classify affected software versions. This hypothesis was confirmed. The results show that AISecNet has a high detection rate in identifying vulnerabilities in software from the extended Nmap scan reports based on the text context. The results confirmed a high detection rate of **86.96%** for the various servers.

The second hypothesis was that powerful models can identify vulnerabilities in network scans, assign them to existing threats, and determine the level of risk. This hypothesis was already addressed in Experiment 2, where good results were achieved with the GPT-3 model using a knowledge database via the RAG pattern. In this experiment, the same approach was used, but the developed process was now part of the entire AISecNet framework. The results varied greatly depending on the server: a classification rate of **75%** was achieved for the database server, while the detection rate for the application server was only about **9.09%**. Overall, AISecNet was able to correctly classify **38.1%** of all vulnerabilities to a CVSS score. The hypothesis was thus partially confirmed.

The third hypothesis was that models that have been specifically trained in the field of cybersecurity can achieve similar or better results than conventional evaluation tools. The Nessus vulnerability tool was included in this analysis. The investigation was carried out in a simulation environment in a direct comparison between the AISecNet framework and Nessus. The result was that this hypothesis could be confirmed. In a direct comparison, the AISecNet Framework showed a significantly higher detection rate of **86.96%** compared to **26.09%** for Nessus.

Although the results generally support the hypotheses, there is room for improvement, especially in the consistency of the classification of the level of vulnerability. At a total cost of **3.37 \$**, this is a crucial factor for the practical feasibility of vulnerability analysis using this framework. During the creation of this work, OpenAI released another model, GPT-4 Omni, which costs only **50%** of the current cost while providing the same performance as the GPT-4 Turbo model. If the AISecNet framework were to be iterated over a more extensive network in its current form, it would be necessary to examine to what extent the costs justify the benefits. The Nessus version used in this experiment was free of charge and limited to 12 computers. Currently, the cheapest annual license for Nessus costs **4,836 €**. This allows unlimited scans. It should be mentioned that this experiment focused exclusively on the detection of installed software components. Nessus also offers other security mechanisms, such as vulnerability detection at the system and protocol level, as well as the creation of advanced reports and a user-friendly administration interface.

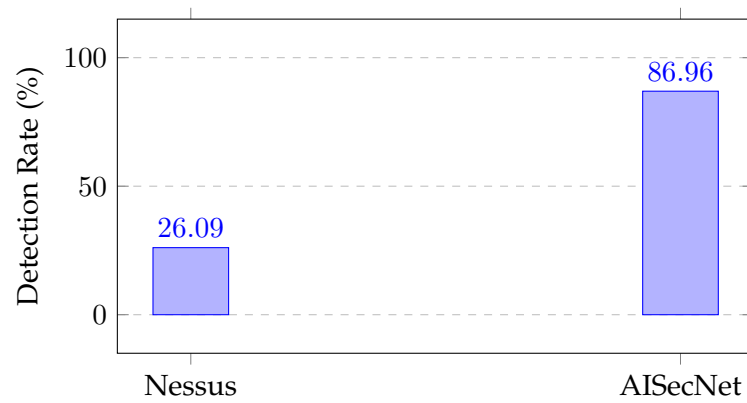


Figure 10: Comparison of detection rates between Nessus and AISEcNet

## 4 Discussion

In this master's thesis, LLMs such as GPT-4 and GPT-3, as well as Mistral Large, were compared with traditional heuristic tools for vulnerability detection. The motivation for this work resulted from the analysis of existing literature and studies that support AI models in cybersecurity. These studies are relatively new, as this field has only been intensively studied since the emergence of LLMs. The changes within the current research are enormous, and during this elaboration, various contexts and new models have emerged and been published (e.g., GPT-4 Omni). In addition, during the preparation of this work, studies have appeared, such as the research by Fang et al. [11], which deals with the compromise of websites by models.

To date, there are no established works of literature that address vulnerability detection through LLMs. This work was, therefore, primarily based on research and studies by other scientists. Various approaches were used in the analysis that link LLMs to the context of cybersecurity. Various sub-areas of cybersecurity were considered, such as the research by Pearce et al. [46], which deals with the identification and repair of software vulnerabilities. This research provides important insights into how models could be used to eliminate vulnerabilities in vulnerable code.

Other research, such as that of Aghaei et al. [1] and Shahid and Debar [52], focused specifically on the area of CVE vulnerabilities, which has provided this work with essential insights for further research, especially in terms of classifying vulnerabilities according to the CVSS scoring system, where the CVSS vector string was created by analyzing the CVE description. Other research, such as MalBERT [47], examines AI-assisted Android software for vulnerabilities and code that could indicate malware. Comparison with the existing literature shows that the results are consistent with previous studies that have examined the effectiveness of LLMs in

various application areas of IT security. This work extends these findings to the area of vulnerability detection. It shows that LLMs are not only able to identify known vulnerabilities but also to classify unknown vulnerabilities (within the scope of the CVE description, even if the actual vulnerability is not known).

According to Nourmohammadzadeh et al., [44], LLMs are mainly used in the area of detection in various use cases. Most of the studies in this area appeared in 2023, which underlines the timeliness of the topic addressed in this thesis and the importance of research in the field of cybersecurity in the context of AI, especially LLMs. Many of these research projects have achieved very positive results. The noticeable gap here should be addressed in this work. A direct vulnerability check of installed software was not present in previous research and could not be identified during the research for this work. There are research papers such as those by Fang et al. [11], Aghaei et al. [1], and Shahid and Debar [52] that overlap with this work. The results of this work thus cover an area that still needs to be addressed in the context of these research papers.

The developed AISecNet framework would also be classified in the areas of detection classified by CSF (see Chapter 2.2) like most of the frameworks analyzed by Nourmohammadzadeh et al. [44].

## **4.1 Interpretation of the results**

At the beginning of this work, several research questions were formulated. This work aimed to answer these questions. To answer the research questions, experiments, and literature research had to be carried out. The findings obtained from this provide a comprehensive overview of the models' performance and allow conclusions to be drawn as to the extent to which LLMs could be used in cybersecurity to improve detection rates and thus ensure more accurate and precise monitoring of critical infrastructures such as corporate networks.

First, the research question RQ1 is considered: "How does the efficiency of an LLM in detecting vulnerabilities in IT networks compare to established vulnerability assessment tools?"

To answer this question, we investigated how vulnerabilities in a network can be measured and how these results can be integrated into a specially developed solution so that models can work on this data. In this work, the use of Nmap scans was discussed (see chapter 2.8) in order to use them in a comparative experiment and to compare the performance of an LLM with established assessment tools. Nessus was chosen as the comparison tool.

To achieve the most accurate results possible, the first two experiments evalu-

ated how models can detect vulnerabilities with and without RAG patterns. These results were used to conduct Experiment 3 (see Chapter 3.6.1), which served to answer RQ1.

The results of this experiment show that LLMs can be used to perform vulnerability analyses. The results have shown that LLMs are not only able to identify vulnerabilities in IT networks but can also outperform established evaluation tools in a high percentage of performance. In a direct comparison from Experiment 3, the developed AiSecNet framework, which works with multiple layers of LLM models, was able to achieve a detection rate of 86.96%. In contrast, Nessus only achieved a detection rate of 26.09%. The implications of these results are manifold. In practical terms, this means that LLMs can be integrated into existing security infrastructures to improve detection and response times to security incidents. Theoretically, the results provide new insights into the application of machine learning and artificial intelligence in cybersecurity.

However, it must be taken into account that Experiment 3 was carried out on a small database of 22 installed software programs, which limits the validity of the results. It remains to be seen whether Nessus would have achieved better results than in the small experimental framework with a more significant number of computers to be scanned or an entire infrastructure. Based on the other experiments conducted, the AiSecNet framework would have achieved similarly good results with a more extensive data set. This assumption is supported by the results in Experiment 1, which showed that the specially trained GPT-4 Turbo model with Azure AI Search as an RAG approach was able to deliver very good to almost perfect metrics. The precision was 96.1%, and the accuracy was 81.9% (see chapter 3.3). Answering the research question leads to the conclusion that LLMs trained specifically for vulnerabilities could provide significant added value in cybersecurity when used productively.

Static analysis tools such as Nessus offer many other methods in addition to the scanning option used, which enable more targeted security scans. Among other things, Nessus can perform system-level scans or generate reports that contain valuable information and descriptions of the vulnerabilities detected. These capabilities were not examined in this work in relation to the implementation of AI-based methods in the context of LLM. Nevertheless, there are already recognizable research fields that leave open how user-friendly reports can be created and what countermeasures could be taken.

However, it should be noted that the LLM only recognizes vulnerabilities that have already been described in the CVE database. Therefore, the RAG approach must always be kept up to date in order to identify new vulnerabilities correctly. New vulnerabilities unknown to the LLM could not be recognized even with this support. To do so, combinations of several process steps would have to be carried



out, such as examining the source code for potential vulnerabilities based on the research of Nourmohammadzadeh et al. [44].

The models that were not pre-trained were not able to deliver the best performance in recognizing vulnerabilities. The research showed that the pure GPT-4 model was only accurate to a small extent of 8.3% in assigning software to an existing vulnerability. In particular, the contextual understanding of text that the language models guarantee through their given architecture has been shown to be able to identify vulnerabilities to a high degree. This was demonstrated by the pure examination of the Nmap scan, and the extension of the Nmap scan results through the implementation of extended scripts. It was initially somewhat surprising that Nmap does not provide a script that executes a Curl command. The assumption here is that there has yet to be any added value so far in displaying the content of the website installed on the port for a scan. However, using the NLP capabilities of LLMs gives a decisive advantage for this experiment in clearly identifying software and version. The accuracy of whether a statement is correct, whether software and version are really affected, was also increased in the experiments, as the assumptions were confirmed that LLMs also correctly assign software and version if the version is not exactly as described in the vulnerability description of CVE. For example, if a version of a software is in a particular range.

In summary, research question RQ1's effectiveness and efficiency not only approach but also greatly exceed those of established vulnerability assessment tools.

The second research question that was posed was: "How do various LLMs compare in terms of efficiency for detecting vulnerabilities in IT networks, as measured by key performance metrics such as Precision, Recall, F1-Score, and False Positive/Negative Rates."

To this end, a diverse selection of different standard pre-trained models was selected in an experiment in order to compare them with each other. The selection of models included the models developed by OpenAI, GPT-3, and GPT-4. Other models were evaluated, as described in this paper, and to introduce diversity, the Mistral Large model was also added to the consideration. The first important finding was that models in a pre-trained state were able to identify known vulnerabilities, but only to a minimal extent.

In contrast, models that were specifically trained for the domain of cybersecurity, in particular for vulnerabilities in software, were compared. In Experiment 1 (see 3.3), it was evaluated how a pre-trained LLM can perform with the enrichment of large knowledge databases that were previously unknown to the model (or only to a small extent), in contrast to purely pre-trained models. This experiment had an enormous impact on the further implementation of this work due to the in-depth preliminary analysis and research on the topic of RAG patterns (see chapter 2.1.1).

The possibility of using a pre-trained model and enabling it to access all existing CVE vulnerabilities via an RAG pattern proved to be the critical factor in achieving a very high performance in the identification of vulnerabilities in Experiment 1. The combination of this knowledge database and the contextual text understanding that the models bring with them had a decisive impact on the positive results of this work. This was clearly demonstrated in the comparisons based on key performance metrics such as recall, precision, F1 score, and false positive/negative rates.

In particular, the RAG pattern using the Microsoft Azure solution, especially Azure AI Search, was able to offer significant added value compared to the on-premise solution from LlamaIndex. The focus was on evaluating these models and RAG approaches. The results based on the metrics show that the GPT-4 Turbo model, which used Azure AI Search, performed best. This was followed by GPT-3 with the same RAG approach. All other models without the RAG approach performed poorly and were not measurable in the context of cybersecurity. The reasons for this were described in detail in the experiment. One of the main reasons for this was that a general pre-trained model was not explicitly trained for the field of cybersecurity but instead contained a large knowledge corpus from other domains, and CVE entries only make up a small part of the overall knowledge. The interpretation of this is that it is not the trained knowledge base that matters but rather how the model can be trained for a specific domain. This allows an entirely different perspective on this work and the context to flow. The real advantage that the models provided was the NLP understanding of how to extract parameters from texts and interpret them correctly and intelligently.

One gap and open question that arises is how other models and other RAG approaches from other providers would perform. This experiment was focused on Microsoft technologies due to the preliminary phase of this research and the available resources. Other providers, which were mentioned by name in chapter 2.1.1, would also have to be evaluated for further research approaches. However, it can be assumed that the RAG patterns offered by Google, AWS, and others would achieve similar results since the RAG pattern is basically consistently implemented in the same way. Google and AWS also offer hybrid and semantic searches on the index.

In summary, Experiment 1 was crucial for the further development and implementation of the subsequent experiments. Therefore, the RQ2 can be answered by saying that, as shown in Table 14, the GPT-4 Turbo model, which was the most up-to-date at the time the experiment was carried out, was able to achieve the best results with 124 TP values when extended by an RAG (see Table 15).

Furthermore, additional questions were asked at the beginning of this work that were to be answered in the course of this work by the results obtained:

### **What are the limitations of using LLMs for this purpose compared to traditional vulnerability assessment tools?**

During this work, it was found that LLMs are very dependent on the quality and completeness of the training data. If the training data is complete and correct, the performance of the model is positively affected. If a model does not have the necessary knowledge base, it may hallucinate and give incorrect answers. Especially in a sector like cybersecurity, these points must be considered critically, as misjudgments in sensitive areas can have critical consequences (see chapter 2.1.1).

Furthermore, during the research for this thesis, it became apparent that there needs to be more standardization of vulnerabilities. LLMs may have difficulties in developing a uniform CVE database. It must be said that this thesis is based on the CVE database from MITRE (see chapter 2.5). It was only during the work that it became apparent that the NVD database has the same up-to-date status as the CVE database from MITRE (it is the source of the CVE records). In contrast to the CVE database from MITRE, the NVD database is better maintained and more up-to-date in the areas of CWE, CVSS, and exploit references.

Another major factor in considering models' limitations in this area is the costs incurred. For example, a cost analysis was carried out in the chapters of the experiments, which, using the current AISEcNet framework, resulted in costs of 3.37\$. In contrast, Nessus was free of charge (if the computer resources were not taken into account).

### **Can the use of LLMs in conjunction with traditional tools yield better results than using either individually?**

Yes, the combination of traditional tools can yield better results with the integration of LLMs. During the research for this thesis, a deeper insight into the functioning of traditional tools, in this case Nessus, could be determined. Nessus and other tools are well-established, widely used, and already known to users. They have a large community and an extensive range of additional plugins. According to Nessus (footnote <https://www.tenable.com/blog/introducing-nessus-expert-now-built-for-the-modern-attack-surface>), Nessus is the leader in the field of assessment tools and offers Nessus in all installation forms and on various operating systems. It can be assumed that the large providers, in particular, are currently investing in the integration of LLMs themselves since, as can be seen in this paper, the performance with the text context understanding of LLMs offers a high added value in the quality of the already very well developed assessment tools.

### **What kind of customization is required to improve the efficiency of an LLM in vulnerability detection?**

It was clearly shown in this thesis that the use of a separate knowledge base in

the form of an RAG offers excellent added value. This was described in the last chapters and the chapters of the experiments (see chapters 3.3, 3.5 and 3.6). What was not considered in this thesis is the fine-tuning of models in this domain. Within this work, attempts were made to fine-tune the models to the field of cybersecurity. The integration was planned for Experiment 2 in order to classify vulnerabilities based on the description. However, the procedure and the experiments were not documented; the results are recorded in Table 24.

The experimental setup was based on two different approaches. On the one hand, the GPT-3 model (GPT-4 was not yet fine-tunable as of May 2024) was trained with a more significant number of pre-defined prompts about CVE vulnerabilities and the associated CVSS metrics. The other approach was carried out in line with Aghaei et al. [1] research. In this approach, a separate model was trained for each metric of a CVSS vector string. In the experiment, the vulnerability description was then iterated through each model, and a statement was generated for each vector metric.

The results were not convincing, so the implementation focused specifically on the RAG pattern. Due to the workload for this work, which arose for the further topic of fine-tuning, this area has not been further investigated. Therefore, it does not provide any usable results and thus offers no added value for research. Overall, further in-depth research in this area would be worthwhile.

### **Can LLMs assess the severity of a vulnerability?**

The results of Experiment 2 (see Chapter 3.5) show that the use of LLMs to detect the severity of a vulnerability can be used well. The results obtained in this experiment show a high level of accuracy in the interpretation of textual descriptions of vulnerabilities. This area of vulnerability detection in the support of language models has, as already mentioned above, also been considered in the interest of other research. In this work, as well as in the work of Aghaei et al. [1], and in the work of Shahid and Debar [52], various approaches described in the chapters of Experiment 2 were carried out. Both the CVSS-BERT, CVEDrill, and the implementation via the GPT-3 model with the RAG approach via Azure AI Search were able to achieve an accuracy of over 80% in most of the CVSS metrics. These findings from the other research bring significant added value to cybersecurity. According to Aghaei et al. [1], it is assumed that CVEDrill (or a generalized similar approach) will play a central role in improving proactive cybersecurity measures, strengthening digital landscapes against emerging threats and contributing to standardization and general practices. This work supports this thesis.

## 4.2 Practical implications and applications

The use of LLMs can significantly improve the detection rate of vulnerabilities in IT networks. This work has shown that LLMs can process contextual and semantic information, enabling detailed analysis. AISEcNet was able to create more specific vulnerabilities than Nessus. This leads to a more comprehensive security check and reduces the risk of undetected threats. In practice, the use of a framework like AISEcNet could be extended to take over possible routine tasks such as regular risk reports and documentation.

In this work, the framework was built on the results of the various experiments extracted by the methods developed for the experiments and integrated into the AISEcNet framework. Using the framework for a scan costs an average of 1.50\$. The framework developed here is based entirely on Python and can only be started via the console; it has no user interface. The AISEcNet framework was more of a proof of concept (PoC) that carried out a feasibility study in this work to answer specific hypotheses and research questions. Therefore, the context of the framework was tailored to a sub-area.

When comparing the costs of Nessus and AISEcNet, only the license fees were compared with the costs incurred by the corresponding models. If an AI-based framework were to be used productively, it could cover additional areas that Nessus already covers. In this case, the final cost-benefit analysis would have to take into account additional factors, such as development costs and maintenance costs, that a development project would require to implement a productively usable framework.

As described, the GPT-4 Omni model was released during the development of this work, which only costs 50% of the actual cost. For the cost calculation, this would have already caused an average price of 0.75\$ per scan. The focus of this experiment was mainly on the pre-trained models from OpenAI. Mistral Large was included in the analysis but was only able to achieve a low accuracy of 34%. This work should have covered how the model could be measured. If Mistral Large had also had the advantage of a knowledge base in the form of an RAG, it would have been possible to achieve a higher accuracy. Therefore, the research part remains open as to whether there may be other models that are cheaper than the OpenAI models and could also achieve good performance. The approach of fine-tuning was also not directly considered in this work. It was mentioned that partial developments in this area had taken place for this work, but that due to the complexity and the results achieved for this, they needed to be better to be able to work on them further. However, this only refers to the part within this research, and more profound research for the context of cybersecurity with the inclusion of fine-tuning could promise more successful results.

Since the major established assessment tools are already well developed in the

field of cybersecurity, the inclusion of LLMs would be essential for future developments and will help to define a standard.

A productive expansion of the AISecNet framework or a similar framework is also conceivable for productive use. For example, this could involve working with a local instance of a RAG, which would require increased computing resources, as well as the most up-to-date possible updates of CVE records and other important information in the field of cybersecurity, such as the ExploitDB and scientific papers on specific areas and other sources that could contribute to vulnerability analysis.

In the course of the research, a Proof of Concept (PoC) was set up to extend the AISecNet framework via an interface. A rough architecture was developed to test the extent to which productive use would be possible. The architecture includes a Docker image to be installed within the network, which contains two applications. On the one hand, a frontend component was developed that accepts an IP address via an input field for the PoC and sends it to a Python Flask backend. Both components were packaged in a Docker container and can be installed on an independent operating system. The Python script receives the IP from the front end, performs a Nmap scan against the target, and sends the Nmap scan report to a .Net web API in the Azure cloud. There, the API call is processed by the Nmap scan against the GPT-4 Omni model. The result is then sent back to the client and displayed on an output page.

The description of the PoC is not detailed here, as it is not part of this work. However, it should show that it would be feasible to implement the process that was carried out in the various experiments in this work in a productive scenario. The feasibility of executing this in a user-controlled manner via a user interface with a backend that is not installed locally on the client but is operated in the cloud offers the advantage that users could be provided with a daily updated RAG. Possible monetization could also be offered as pay-as-you-go models, where billing could be based on use or in a subscription model.

The interface, which is run locally as a Docker image, could be expanded accordingly to enable a user-friendly interface like Nessus. The search algorithms could then go beyond pure Nmap scans and, for example, also use other tools such as Nikto<sup>53</sup>, a web server scanner that detects many potentially dangerous files and CGIs, as well as outdated software versions and other security problems, to obtain detailed information about the installed web tools or to perform advanced scans at the operating system level. The results could then be interpreted using a corresponding model, thus generating detailed damage reports in text form, categorizing the network according to risks, and obtaining further creative, text-context-related information about the risk status of the corresponding systems. Approaches that have already been explored in the research of Fang et al. [11] could also be used to

---

<sup>53</sup><https://cirt.net/Nikto2>

develop one's exploits based on the error description in order to check whether the affected system is really affected by the vulnerability. Of course, ethical considerations would have to be taken into account since such frameworks could not only be used in the context of the positive aspects of cybersecurity but also, on the other hand, to use these tools to gain unauthorized access to systems.

## References

- [1] Ehsan Aghaei et al. „Automated CVE Analysis for Threat Prioritization and Impact Prediction“. In: *arXiv preprint arXiv:2309.03040* (Sept. 2023). arXiv: 2309.03040 [cs.CR].
- [2] Kimia Ameri et al. „CyBERT: Cybersecurity Claim Classification by Fine-Tuning the BERT Language Model“. In: *Journal of Cybersecurity and Privacy* (2021). DOI: 10.3390/jcp1040031. URL: <https://doi.org/10.3390/jcp1040031>.
- [3] Pierre Baldi1 et al. „Assessing the accuracy of prediction algorithmsfor classification: an overview“. In: <https://www.researchgate.net/> (2000).
- [4] Markus Bayer et al. „CySecBERT: A Domain-Adapted Language Model for the Cybersecurity Domain“. In: *Proceedings of the Conference Acronym 'XX*. New York, NY, USA: ACM, 2022. URL: <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>.
- [5] Martin Berglund and Brink van der Merwe. „Formalizing BPE Tokenization“. In: *Electronic Proceedings in Theoretical Computer Science (EPTCS)* (2023). arXiv:2309.08715 [cs.FL]. URL: <https://doi.org/10.48550/arXiv.2309.08715>.
- [6] Nitin Bhatia and Vandana. „Survey of Nearest Neighbor Techniques“. In: *International Journal of Computer Science and Information Security* 8.2 (2010). Corresponding author: Nitin Bhatia, n/a.
- [7] Tom B. Brown et al. „Language Models are Few-Shot Learners“. In: *arXiv:2005.14165* (2020). Available at <https://arxiv.org/abs/2005.14165>.
- [8] Steven M. Christey David E. Mann. „Towards a Common Enumeration of Vulnerabilities“. In: (1999). URL: <https://www.cve.org/Resources/General/Towards-a-Common-Enumeration-of-Vulnerabilities.pdf>.
- [9] eSecurityPlanet. *History of Computer Viruses and Malware . What Was Their Impact?* 2023. URL: <https://www.esecurityplanet.com>.
- [10] Europol. *The Impact of Large Language Models on Law Enforcement*. 2023. URL: <https://www.europol.europa.eu/cms/sites/default/files/documents/Tech%20Watch%20Flash%20-%20The%20Impact%20of%20Large%20Language%20Models%20on%20Law%20Enforcement.pdf>.
- [11] Richard Fang et al. *LLM Agents can Autonomously Hack Websites*. License: arXiv.org perpetual non-exclusive license. Feb. 2024. arXiv: 2402.06664v1 [cs.CR]. URL: <https://arxiv.org/abs/2402.06664v1>.
- [12] Zhangyin Feng et al. „CodeBERT: A Pre-Trained Model for Programming and Natural Languages“. In: *arXiv preprint arXiv:2002.08155* (Sept. 2020).



- [13] Mohamed Amine Ferrag et al. „Revolutionizing Cyber Threat Detection with Large Language Models“. In: *arXiv preprint arXiv:2306.14263* (June 2023).
- [14] *Common Vulnerability Scoring System (CVSS)*. <https://www.first.org/cvss/v3-1/>. 2021.
- [15] FIRST. *Common Vulnerability Scoring System SIG*. URL: <https://www.first.org/cvss/>.
- [16] FIRST.Org, Inc. *Common Vulnerability Scoring System v3.1: Specification Document*. <https://www.first.org/cvss/v3.1/specification-document>.
- [17] Yunfan Gao et al. „Retrieval-Augmented Generation for Large Language Models: A Survey“. In: *arXiv arXiv:2312.10997* (2024). URL: <https://arxiv.org/abs/2312.10997>.
- [18] Gartner. *Gartner Experts Answer the Top Generative AI Questions for Your Enterprise*. 2023. URL: <https://www.gartner.com/en/topics/generative-ai>.
- [19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- [20] Michael Günther et al. „JINA EMBEDDINGS 2: 8192-Token General-Purpose Text Embeddings for Long Documents“. In: *arXiv preprint arXiv:2310.19923* (2024). Available from Jina AI GmbH: <https://jina.ai/embeddings>.
- [21] S. Haripriya, A. Brahmananda Reddy, and A. Prashanth Rao. „A Review on Semantic Approach using Nearest Neighbor Search“. In: *IOSR Journal of Computer Engineering (IOSR-JCE)* (2016). ISSN: 2278-0661 (e-ISSN), 2278-8727 (p-ISSN). DOI: 10.9790/0661-1803054653. URL: <https://www.iosrjournals.org>.
- [22] Ben Hoover, Hendrik Strobelt, and Sebastian Gehrmann. „EXBERT: A Visual Analysis Tool to Explore Learned Representations in Transformers Models“. In: *arXiv preprint arXiv:1910.05276* (Oct. 2019).
- [23] IBM. *The history of malware: A primer on the evolution of cyber threats*. 2023. URL: <https://www.ibm.com/blogs>.
- [24] Philip Tully John Seymour. „Generative Models for Spear Phishing Posts on Social Media“. In: *arxiv:1802.05196* (2018). URL: <https://arxiv.org/pdf/1802.05196.pdf>.
- [25] Scott Levy and Jedidiah R. Crandall. *The Program with a Personality: Analysis of Elk Cloner the First Personal Computer Virus*. arXiv:2007.15759v1 [cs.CR]. 2020.
- [26] Patrick Lewis et al. „Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks“. In: *arXiv preprint arXiv:2005.11401* (2020). URL: <https://arxiv.org/pdf/2005.11401.pdf>.

- [27] Yiheng Liu et al. „Summary of ChatGPT-Related Research and Perspective Towards the Future of Large Language Models“. In: *Meta-Radiology* 1 (2023). DOI: 10 . 1016 / j . metrad . 2023 . 100017. URL: <https://www.keaipublishing.com/en/journals/meta-radiology>.
- [28] Microsoft. *Introducing Microsoft 365 Copilot – your copilot for work*. 2023. URL: <https://blogs.microsoft.com/blog/2023/03/16/introducing-microsoft-365-copilot-your-copilot-for-work/>.
- [29] Microsoft. *Introduction to Azure Storage*. <https://learn.microsoft.com/en-us/azure/storage/common/storage-introduction>. Accessed: 2024-04-23. 2023.
- [30] Microsoft. *Microsoft and OpenAI Extend Partnership*. <https://blogs.microsoft.com/blog/2023/01/23/microsoftandopenaiextendpartnership/>. [Accessed: 20.04.2024]. Jan. 2023.
- [31] Microsoft. *Microsoft Copilot for Security*. 2024. URL: <https://www.microsoft.com/en-us/security/business/ai-machine-learning/microsoft-copilot-security>.
- [32] Microsoft. *Pricing - Cognitive Services | OpenAI Service*. <https://azure.microsoft.com/en-us/pricing/details/cognitive-services/openai-service/>. Accessed: 05.04.2024. 2023.
- [33] Microsoft. *Relevance in Vector Search*. Apr. 2024. URL: <https://learn.microsoft.com/en-us/azure/search/vector-search-ranking> (visited on 04/22/2024).
- [34] Microsoft. „Vectors in Azure AI Search“. In: *Microsoft Azure Documentation* (Apr. 2024). URL: <https://learn.microsoft.com/en-us/azure/search/vector-search-overview#approximate-nearest-neighbors>.
- [35] Microsoft. *Was ist Microsoft Copilot für Azure*. 2023. URL: <https://learn.microsoft.com/de-de/azure/copilot/overview>.
- [36] Microsoft Corporation. *AI Search*. <https://azure.microsoft.com/en-us/products/ai-services/ai-search>. Accessed: 21/04/2024. 2024.
- [37] Microsoft Corporation. *Quickstart: Set up a tenant*. <https://learn.microsoft.com/en-us/entra/identity-platform/quickstart-create-new-tenant>. Accessed: 01/11/2024. Jan. 2024.
- [38] Microsoft Corporation. *What is Infrastructure as Code (IaC)?* <https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code>. Accessed: 20/04/2024. Nov. 2022.
- [39] Stephen Moskal et al. „LLMs Killed the Script Kiddie: How Agents Supported by Large Language Models Change the Landscape of Network Threat Test-

- ing". In: *arXiv preprint arXiv:2310.06936* (2023). URL: <https://arxiv.org/abs/2310.06936>.
- [40] Niklas Muennighoff et al. „MTEB: Massive Text Embedding Benchmark“. In: *arXiv preprint arXiv:2210.07316* (2023). Code and leaderboard available at <https://github.com/embeddings-benchmark/mteb> and <https://huggingface.co/spaces/mteb/leaderboard>.
  - [41] National Institute of Standards and Technology. *National Institute of Standards and Technology Website*. <https://www.nist.gov/>.
  - [42] Michael A. Nielsen. *Neural Networks and Deep Learning*. 2015. URL: <http://neuralnetworksanddeeplearning.com/index.html>.
  - [43] *NIST Cybersecurity Program History and Timeline*. National Institute of Standards and Technology (NIST). 2023. URL: <https://csrc.nist.gov>.
  - [44] Farzad Nourmohammadzadeh et al. „Large Language Models in Cybersecurity: State-of-the-Art“. In: *arXiv:2402.00891v1* (2024). Available at <https://arxiv.org/abs/2402.00891>.
  - [45] Angela Orebaugh and Becky Pinkard. *Nmap in the Enterprise: Your Guide to Network Scanning*. Syngress, 2011. ISBN: 9780080558745.
  - [46] Hammond Pearce et al. „Examining Zero-Shot Vulnerability Repair with Large Language Models“. In: *arXiv:2112.02125* (2023). Available at <https://arxiv.org/pdf/2112.02125.pdf>.
  - [47] Abir Rahali and Moulay A. Akhloufi. „MALBERT: Using Transformers for Cybersecurity and Malicious Software Detection“. In: *A Preprint* (Mar. 2021).
  - [48] SentinelOne. *A Brief History of Cybersecurity*. 2023. URL: <https://www.sentinelone.com>.
  - [49] SentinelOne. *A Brief History of Cybersecurity*. 2023. URL: <https://www.sentinelone.com/blog/a-brief-history-of-cybersecurity/>.
  - [50] Mohak Shah and Nathalie Japkowicz. *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge University Press, 2011. ISBN: 978-0521196000.
  - [51] Mustafizur R. Shahid and Hervé Debar. *CVSS-BERT: Explainable Natural Language Processing to Determine the Severity of a Computer Security Vulnerability from its Description*. Available at arXiv. France, 2021. URL: <https://arxiv.org/pdf/2111.08510.pdf>.
  - [52] Mustafizur R. Shahid and Herve Debar. „CVSS-BERT: Explainable Natural Language Processing to Determine the Severity of a Computer Security Vulnerability from its Description“. In: *arXiv preprint arXiv:2111.08510* (2021).
  - [53] Jiamou Sun et al. „Generating Informative CVE Description From ExploitDB Posts by Extractive Summarization“. In: *arXiv* (2021). Jiamou Sun, Zhenchang Xing (Research School of Computer Science, CECS, Australian National University, Canberra, Australia), Hao Guo, Xiaohong Li (College of Intelligence

- and Computing, Tianjin University, Tianjin, China), Deheng Ye (Tencent AI Lab, Shenzhen, China), Xiwei Xu, Liming Zhu (Data61, CSIRO, Australia). arXiv: arXiv:2101.01431 [cs.LG].
- [54] Chandra Thapa et al. „Transformer-Based Language Models for Software Vulnerability Detection“. In: *arXiv preprint arXiv:2204.03214* (Sept. 2022).
  - [55] *U.S. Passes New Cybersecurity Laws in June 2022*. EC-Council. 2022. URL: <https://www.eccouncil.org/u-s-passes-new-cybersecurity-laws-in-june-2022/>.
  - [56] Ashish Vaswani et al. „Attention Is All You Need“. In: (2017).
  - [57] Jiayin Wang et al. „A User-Centric Benchmark for Evaluating Large Language Models“. In: *arXiv preprint arXiv:2404.13940* (2024). Benchmark dataset and code available at <https://github.com/Alice1998/URS>.
  - [58] Hongbin Ye et al. „Cognitive Mirage: A Review of Hallucinations in Large Language Models“. In: *arXiv* (2023). arXiv: 2309.06794. URL: <https://arxiv.org/abs/2309.06794>.