Escuela de Computación
Unidad de Posgrado

TEC | Tecnológico de Costa Rica

ACTA DE APROBACION DE TESIS

GPT-Based Identification of Publicly Known Vulnerabilities

Por: Andrés Felipe Vargas Rivera

TRIBUNAL EXAMINADOR

Herson E. V.

Dr. Herson Esquivel Vargas
Profesor Asesor

MSc. Jeferson González Gómez
Profesor Lector

Dr. Thijs van Ede
Lector Externo

Dra.-Ing. Lilliana Sancho Chavarría
Presidente, Tribunal Evaluador Tesis
Programa Maestría en Computación

SINAES POST-GRADO ACREDITADO COSTA RICA

14 de mayo, 2024

Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Programa de Maestría en Computación

GPT-Based Identification of Publicly Known Vulnerabilities

A thesis submitted in partial fulfillment to opt for the degree of

Magister Scientiæ in Computer Science

Author

Andrés Felipe Vargas Rivera

Supervisor

Herson Esquivel Vargas

May 22, 2024

Dedication

This thesis is dedicated to my parents, who have been my rock and source of strength through every challenge. To my sister, whose encouragement and unwavering belief in my abilities have been a constant source of inspiration. To my friends, who stood by me with words of support and moments of laughter, making this journey more enjoyable. I would also like to express my deepest gratitude to the Vicerrectoría de Investigación Estudiantil (VIE), the School of Computing, and the Instituto Tecnológico de Costa Rica (TEC). Their resources, guidance, and support provided the foundation for my research and education. A special thanks goes to my thesis advisor, Herson Esquivel, whose insightful guidance and constructive feedback shaped my work. His patience and commitment to excellence were crucial during key moments in my research. Finally, I extend my sincere appreciation to those who funded and supported this work. Your contributions not only made this thesis possible but also helped advance knowledge in this field. Thank you to everyone who has been part of this journey. This work would not have been possible without your encouragement, support, and belief in my potential.

Dedicación

Dedico esta tesis a mis padres, quienes han sido mi apoyo y fuente de fortaleza en cada desafío. A mi hermana, cuyo aliento y fe inquebrantable en mis habilidades han sido una constante fuente de inspiración. A mis amigos, que estuvieron a mi lado con palabras de apoyo y momentos de risa, haciendo este camino mucho más llevadero. También quiero expresar mi más sincero agradecimiento a la Vicerrectoría de Investigación Estudiantil (VIE), a la Escuela de Computación, y al Instituto Tecnológico de Costa Rica (TEC). Sus recursos, orientación y apoyo proporcionaron la base para mi investigación y educación. Un agradecimiento especial a mi tutor de tesis, Herson Esquivel, cuya guía perspicaz y retroalimentación constructiva dieron forma a mi trabajo. Su paciencia y compromiso con la excelencia fueron cruciales en momentos clave de mi investigación. Finalmente, extiendo mi sincero agradecimiento a quienes financiaron y apoyaron este trabajo. Sus contribuciones no solo hicieron posible esta tesis, sino que también ayudaron a avanzar en el conocimiento de este campo. Gracias a todos los que han sido parte de este camino. Este trabajo no habría sido posible sin su aliento, apoyo y fe en mi potencial.

Abstract. Security vulnerabilities are inherent to software systems. Nevertheless, the software industry is continuously growing and so is the amount of security vulnerabilities discovered every year. For instance, during the year 2023, an average of 79 software vulnerabilities were published every day. In the software security field, the use of vulnerability scanners is common practice. These tools have databases of known vulnerabilities and verify whether a target system is vulnerable or not, by looking for matching records in their database. Although vulnerability scanners automate the tedious process of checking software applications for vulnerabilities, the daily updates to vulnerability scanners remain, predominantly, a manual task. This poses a scalability problem for vulnerability scanners. In this work, we present a novel architecture designed to automate the Vulnerability Identification in software products. This thesis explores the architecture's underlying principles, its implementation, and its performance evaluation. We demonstrate how our system effectively identifies vulnerabilities by using pre-existing AI tools, thereby empowering organizations to proactively secure their software assets, protect sensitive data, and enhance overall cybersecurity resilience. The architecture proposes the use of a database that contains vulnerability signatures which, when compared with the signature of a software product, are used to identify vulnerabilities. To demonstrate the viability of the architecture, two implementations are carried out. The first solution addresses a heuristic model, and the second the use of Artificial Intelligence (AI). More specifically, a Generative Pre-Trained Transformer (GPT) model. The results showed that, for the signature's generation, the GPT model automatically creates the vulnerability database signatures with an accuracy of 100%, whereas its heuristic counterpart achieves a modest 73,2%. In the vulnerability identification process, the recall metric is crucial in because it reflects the ability to detect actual vulnerabilities among all possible cases. Our results show that the GPT-based approach exhibited significantly higher recall 94,6% than the heuristic-based Vulnerability Identification System 23,8%, indicating a more reliable detection of vulnerabilities. This advantage means that using GPT for vulnerability identification reduces the risk of missing critical vulnerabilities, leading to a more secure and resilient system. Based on the results obtained, we conclude that the proposed architecture is able to automate the MITRE CVE-based vulnerability identification, Artificial Inteligence being one of the most promising technologies to automate and improve future vulnerability identification systems.

Keywords: Artificial Intelligence · Vulnerability · Security Tests

Table of Contents

List of Figures

List of Tables

1 Introduction

The proliferation of software systems has changed forever the functioning of modern societies. Besides the traditional software applications like email clients, text editors, and spreadsheets, software now executes the core tasks of utility companies, cars, factories, buildings, and many others. Recent trends like cloud computing, IoT, and Industry 5.0 have strengthened the software industry to achieve a market size value of $583.47 billion in 2022, which is expected to double by 2030 [19].

Unfortunately, software systems are not exempt from security weaknesses that might be exploited by cyber-attackers. To tackle this problem, there have been diverse efforts to improve the security of software systems: programming languages with built-in security features, static source code analysis tools, dynamic binary analysis tools, formal methods tools, security related improvements during the software development process, among several others [24,28,47,49,52]. Nonetheless, these efforts have been insufficient to free our pervasive software systems from security weaknesses.

Exploitable software weaknesses are called *vulnerabilities* [46]. The need to create a central repository of publicly disclosed vulnerabilities led to the creation of MITRE's Common Vulnerabilities and Exposures (CVE) list [29]. In this list, each vulnerability gets assigned a unique identifier along with other details such as the software versions affected, the consequences of the attack, and external references about it. Industry reports show an increasing trend in the amount of publicly disclosed software vulnerabilities, i.e., CVE records [5,40].

As the number of vulnerabilities increases, so does the need to test and identify them. There are various techniques to identify vulnerabilities in software. Some of them are penetration testing, static application security testing, dynamic application security testing, interactive application security testing, among others. What these techniques have in common is that to support the analysis of new scenarios, they must

be updated manually. This manual process indicates that the issue of daily CVE increments persists within these tools. This is because work teams are required to update the identification systems. The process is labeled as manual because the teams consume resources and must spend time to develop the ability to identify specific vulnerabilities.

This investigation proposes to automate the Vulnerability Identification, without the need for human interaction when new CVEs are added.

The proposed solution automatically interprets and extracts relevant information from CVE descriptions. Concretely, it extracts the software name and its vulnerable versions. This extraction is challenging because there are several ways to write this information in natural language. This first step is called *signature generation module*. Afterward, our solution checks if the target software systems match any of our automatically extracted database of CVE signatures. This 2-step process represents the most essential function of any vulnerability scanner. In this way, the solution depends solely on updated information from MITRE's in CVEs.

1.1 Problem Definition

The Common Vulnerabilities and Exposures (CVE) system, managed by MITRE, serves as a central repository for publicly disclosed software vulnerabilities. These vulnerabilities are written in natural language (i.e., English) without a standardized format, making it difficult to automate their processing.

This lack of uniformity forces security professionals to manually review CVE records to update vulnerability scanners and other security tools. This manual approach is time-consuming, prone to error, and can lead to delays in addressing critical vulnerabilities. As the number of CVE records continues to grow, automation becomes increasingly important to keep pace with evolving security threats. So, the core problem is that the natural language nature of CVE descriptions complicates their automated use in vulnerability identification.

Software vulnerabilities can be identified during the development stage but also in its production environment. In the first case, software developers might use static/dynamic application security testing (SAST/DAST), formal methods, and other tools to identify weaknesses and/or vulnerabilities before the software enters its production stage [11, 25]. Once in its production environment, software might still suffer from security vulnerabilities. These vulnerabilities can be previously unknown (zero-day vulnerabilities) or well-documented CVE-listed vulnerabilities.

In an effort to find a more efficient solution, this research focuses on applying Artificial Inteligence to automate the vulnerability identification of publicly known vulnerabilities, specifically targeting software already in production. Our goal is to investigate whether pre-trained Artificial Inteligence models can accurately process and extract relevant information from CVE records, reducing the need for manual intervention. By using Artificial Inteligence to automate this process, our study aims to improve the speed and reliability of Vulnerability Identification, allowing security teams to respond to threats more quickly and effectively.

1.2 Research Justification

Identifying software vulnerabilities is a critical priority for organizations, as undetected vulnerabilities can lead to severe consequences, including data breaches, financial losses, and safety risks. A cost-effective method to identify software vulnerabilities is through the use of *vulnerability scanners*, which are designed to find known vulnerabilities in various software applications. However, these scanners rely on regular updates from the Common Vulnerabilities and Exposures (CVE) database to remain effective. Given the rapid growth of publicly disclosed vulnerabilities, the task of manually updating *vulnerability scanners* becomes increasingly challenging and prone to error.

Over the years, the realm of software development has experienced a noteworthy surge, prompting the industry to

deploy an augmented array of programs and automated systems in response to this escalation. Driven by trends such as cloud computing, mobile applications, the IoT, and industrial automation, the importance of identifying vulnerabilities early is more critical than ever. The widespread use of software in virtually every sector—business, government, and personal use—has increased the potential impact of software vulnerabilities on safety, privacy, and financial stability. Incidents such as autonomous vehicle accidents due to software hacks or bank data breaches resulting from exploited vulnerabilities highlight the tangible risks posed by software vulnerabilities.

Unfortunately, the growth of software vulnerabilities is so rapid that it is extremely difficult to scan and analyze to determine which systems are vulnerable and which are not. The perceptible expansion is supported since at the time this research is carried out, there are 368 CVE Numbering Authority that are operational in 40 countries, adding thousands of new CVEs per year. Based on MITRE's public information, in 2023, were published **28 961 CVEs** [8]. This rapid expansion poses significant challenges for companies and software development teams in maintaining the necessary pace to ascertain whether each newly identified vulnerability is present within their systems. According to Neil Ford in his data collection for IT Governance, during 2023 there have been more than 953 cases of major attacks that allowed attackers to obtain information from more than 5 billion records [17].

Focused on providing a solution in the area of cybersecurity, this research aims to evaluate the capacity of pre-trained AI models in the vulnerability identification process, allowing vendors and software product owners to improve the security of software and the products they use within their architecture. The aggregation and dissemination of vulnerability information cataloged as CVE allow companies, users, suppliers, and other stakeholders to expedite the process of identifying whether their products are susceptible to newly disclosed vulnerabilities within the community. This shared resource reduces the necessity for extensive manual analysis and

testing of individual CVE records, streamlining the process of vulnerability detection.

1.3 Contributions

This thesis proposes an improvement to the vulnerability identification process. Unlike existing solutions, our proposal allows the automated evaluation of software products against new vulnerabilities added to the CVE list. This system generates signatures of new vulnerabilities through the use of AI, which prevents a developer from having to manually add code to support new vulnerabilities. It also facilitates the demonstration of the potential application of pre-trained deep learning models like GPT-3 in cybersecurity contexts.

A key aspect of our proposed approach is its ability to generate dynamic signatures for newly issued vulnerabilities using a pre-trained deep learning model like GPT-3. This process eliminates the need for developers to manually update code in response to each new CVE, which can be both time-consuming and error-prone. The AI-based method adapts to the varied and complex language used in CVE descriptions, providing a flexible solution that can easily incorporate emerging vulnerabilities.

The automated nature of the system offers significant benefits to developers. By reducing manual intervention, developers can focus on high-level tasks such as feature development, code optimization, and other core responsibilities. This automation allows them to be more proactive in identifying and addressing vulnerabilities, promoting a culture of security awareness throughout the development process. The reduced workload also enhances productivity, as developers can spend less time reacting to security risks and more time on innovation.

In addition to its practical advantages, this research contributes to the broader field of cybersecurity by demonstrating the potential of AI, particularly GPT-3, in security applications. The successful use of pre-trained models to generate vulnerability signatures can inspire further exploration into the role of AI in cybersecurity. This opens new pathways for

the development of robust and efficient security solutions, emphasizing the growing importance of AI in maintaining secure software systems.

1.4 Hypothesis

The proposed identification architecture, utilizing artificial intelligence techniques, is expected to demonstrate the feasibility and effectiveness of automating the identification of vulnerabilities within the MITRE Common Vulnerabilities and Exposures (CVE) database. This automation is anticipated to result in significantly improved accuracy, recall, and scalability compared to heuristic methods.

1.5 Objectives

This section presents the general objective and the specific objectives of this research.

1.5.1 General Objective

Evaluate the feasibility and effectiveness of Generated Pre-Trained Models to automate the identification of vulnerabilities within Common Vulnerabilities and Exposures records.

1.5.2 Specific Objectives

- Evaluate the ability of the GPT model gpt-3.5-turbo-0125 without fine-tuning from the company OpenAI, to extract vulnerable versions of software products from the description of CVE records, based on the accuracy, precision, and recall of the hits with respect to the expected labels.
- Evaluate the ability of the GPT model gpt-3.5-turbo-0125 without fine-tuning from the OpenAI company, to identify vulnerabilities from the comparison of character strings, based on the accuracy, precision, and recall of the hits with respect to the expected labels.
- Show the operation of the vulnerability identification process of the proposed architecture by using CVEs listed in MITRE during 2023, based on the accuracy, precision, and recall of the hits with respect to the expected labels.

1.6 Research Questions

- How effectively can the GPT model gpt-3.5-turbo-0125, without fine-tuning from OpenAI, extract vulnerable versions of software products from Common Vulnerabilities and Exposures (CVE) records?
- To what extent can the gpt-3.5-turbo-0125 GPT model from OpenAI, without fine-tuning, accurately identify vulnerabilities through character string comparisons?
- How does the proposed architecture perform identifying vulnerabilities using MITRE's 2023 CVE dataset?

2 Theoretical Framework

Our proposed solution requires knowledge and research in different areas of computer science. The most significant area is cybersecurity, which according to the United States National Institute of Standards and Technology (NIST) [13], consists of “the prevention of damage, protection, and restoration of computers, electronic communications systems, electronic communications services, communications by cable and electronic communications, including the information contained therein, to guarantee its availability, integrity, authentication, confidentiality and non-repudiation." Cybersecurity is an extremely broad field, however, this research is mainly focused on vulnerabilities. According to CISA [1], vulnerabilities can be described as flaws in software, firmware, or hardware that an attacker can exploit to perform unauthorized actions on a system. They can be introduced due to software programming errors. Attackers take advantage of these errors to infect computers with malware or perform other malicious activities. For this, an extremely important concept is, the vulnerability scanner, which is defined by NIST [14] as “[a] tool (hardware and/or software) used to identify hosts/host attributes and associated vulnerabilities (CVE)". So, it consists of an inspection of potential exploitation points in a computer or network to identify security holes. A vulnerability scanner detects and classifies system vulnerabilities in computers, networks, and communications equipment [50].

This research also involves concepts from the Artificial Inteligence, domain. Artificial Intelligence “[i]s the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to be limited to methods that are biologically observable.” AI, like cybersecurity, is an extremely extensive field that has had great growth thanks to the computational power currently available. This research aims to focus on models using Natural Language Processing (NLP) techniques. NLP is a branch of Artificial Intelligence that focuses on the

ability to understand and take actions based on texts or spoken words. Its theoretical definition, according to the IBM company [15] is "[a] field in which computational linguistics (rule-based modeling of human language) is used with statistical, machine learning and deep learning models. Together, these technologies enable computers to process human language in the form of text or voice data and to 'understand' its full meaning, complete with the speaker or writer's intent and sentiment". NLP allows syntactic analysis processes to be executed through computer systems, which seek relationships between words and represent the dependencies of the writing. Also, when using NLP, semantic analysis processes can be carried out, in which an understanding of language is sought and is known to be one of the most challenging areas in NLP. Among the tasks that an NLP implementation can perform is text classification. To understand this process, we can rely on the definition of Shen, D. [45], which describes text classification, consisting of a process dedicated to automatically assigning textual documents (such as plain text documents and web pages) into some predefined categories based on their content. Formally speaking, text classification works in an instance space X where each instance is a document d and a fixed set of classes $C = C_1, C_2, ..., C_N$ where N is the number of classes. In order to implement text classification methods, there are different emphases.

The implementation of text classification can be done with both machine learning and deep learning techniques. While deep learning makes use of neural networks, machine learning is based on algorithms primarily related to probability and statistics [21].

The datasets for the investigation must allow extracting information from CVEs such as labels, descriptions, among others. For this, it is important to know the different data sources available and collected as part of the initial research process. These different dataset options have changes such as the number of CVEs, the format in which the information comes, the amount of information in each CVE record,

among others. Therefore, it is decided to evaluate during the implementation process which of these sources is most useful for the experiment.

2.1 Related Work

The steady growth of publicly disclosed vulnerabilities has driven the development of diverse automated solutions. For instance, Kanakogi, K et al. [22] consists of a probability analysis that, through the use of Natural Language Processing (NLP) models, classifies which attack patterns, documented in the Common Attack Pattern Enumerations and Classifications (CAPEC), are more likely to be related to CVE records.[1] To do this, the authors analyze CVE and attack pattern descriptions to compute a similarity index. Although their work and ours analyze CVE descriptions, the goal of such analyses is different. Whereas they aim at linking CVEs with CAPEC attack patterns, our goal is to automate the development of software vulnerability tests.

Veneta et al. [51] use machine learning algorithms like Linear Support Vector Classification, Naive Bayes, and Random Forest Classifier to classify CVEs by type. Like our work, CVE descriptions are analyzed, but the final objective of the NLP process is not to determine if a software is vulnerable.

Balasubramanian. et al. [2] focus on a conversational agent framework designed to assist system administrators in cybersecurity operations. This study delves into the fine-tuning of GPT-3 models for tasks such as log summarization, detection of specific events, and providing essential cybersecurity instructions to users. The authors report high BERTscore [53] results, indicating effective summarization of log files. Although this study shows GPT-3's ability to assist with cybersecurity operations, the objective is to support system administrators with log analysis, not to identify vulnerabilities based on CVE listed.

Fu. et al. [18] explore the use of ChatGPT for four key vulnerability-related tasks: function and line-level vulnerability

[1] https://capec.MITRE.org/

prediction, vulnerability classification, severity estimation, and vulnerability repair. Through extensive empirical studies, the paper reveals that despite GPT-3's large model scale, it performs poorly compared to specialized language models for vulnerability-related tasks. The work underscores the need for fine-tuning GPT-3 to generalize better for specific vulnerability tasks. This study provides valuable insights into GPT-3's limitations in this context and suggests the need for domain-specific tuning. Our approach differs in that it examines the capacity of GPT-3 to identify if specific software modules and versions are vulnerable, focusing on a narrower, more specific application in cybersecurity.

Other approaches like Fang. et al. [16] explores the offensive capabilities of large language models (LLMs) in cybersecurity, demonstrating how GPT-4 can autonomously hack websites and extract database schemas without prior knowledge of the vulnerabilities. This contrasts with our approach, which focuses on using GPT model to automate vulnerability identification in software products, emphasizing a defensive approach. Despite these differences, both papers contribute to our understanding of the complex role AI plays in cybersecurity, showcasing its potential to both protect and exploit digital systems.

These studies underscore the diverse approaches taken to address vulnerabilities in software systems. While they each make unique contributions to the field, the work of this study stands apart by utilizing GPT-3 through an API to automate the evaluation of software products against new vulnerabilities.

2.2 Background

2.2.1 CVE Records MITRE's CVE project started in 1999 as a need for software vendors to standardize and list in a public repository the vulnerabilities discovered in their products [29]. When a new vulnerability is found, a request is made and assigned to a group of CVE partners who review, list, and corroborate the information provided.

A CVE record consists of 6 parts [34]. (1) The *CVE ID* is a numeric code that uniquely identifies a CVE record. (2) The

name of the software that is affected by the vulnerability. (3) The *version(s)* of the software that have been confirmed to be vulnerable. (4) *Public references* containing complementary and relevant information to understand the vulnerability. It is typically comprised of external web page links. (5) A *prose description* of the vulnerability. Its goal is to allow the reader to understand the different scenarios in which the vulnerability occurs. And (6) *Background information* comprising the vulnerability type, root cause, and impact. These data give information to the reader about the categorization, origin, and consequences of the vulnerability.

Table 1. Examples of Common Vulnerabilities and Exposures (CVE) record descriptions.

CVE identifier	Description
CVE-2022-45937	A vulnerability has been identified in APOGEE PXC Series (BACnet) (All versions < V3.5.5), APOGEE PXC Series (P2 Ethernet) (All versions < V2.8.20), TALON TC Series (BACnet) (All versions < V3.5.5). A low privilege authenticated attacker with network access to the integrated web server could download sensitive information from the device containing user account credentials.
CVE-2018-8880	Lutron Quantum BACnet Integration 2.0 (firmware 3.2.243) doesn't check for correct user authentication before showing the /deviceIP information, which leads to internal network information disclosure.

CVE prose descriptions are written in natural language (i.e., English). MITRE suggests the following structure to write CVE descriptions [48]: `[PROBLEM TYPE] in [SOFTWARE/VERSION] causes [IMPACT] when [ATTACK]`. Table 1 shows two concrete examples of CVE descriptions following MITRE's suggested structure. Although both CVEs adhere to MITRE's suggested structure, the way in which they communicate the same kind of information varies. CVE-2022-45937 uses boolean comparators to explicitly define a range of affected versions (*"All versions < V3.5.5"*). CVE-2018-8880, on the other hand, expresses the

affected version implicitly. It does not use boolean operators but it is the reader who interprets that the vulnerable version is *equal* to the version mentioned in the CVE record (*"firmware 3.2.243"*). The omission of keywords and comparators makes version identification dependent on the wording and context of the description.

2.2.2 Natural Lenguage Processing Models Natural language processing (NLP) has been classified in literature as a branch of Artificial Inteligence concerned with giving computers the ability to understand text and spoken words in much the same way human beings can [20]. NLP plays an important role in the development of AI since it provides the ability to understand, generate responses, and classify information without the need for explicit human interaction.

One of the main challenges of NLP models is the amount (often tens of thousands) of data required by a model to reach the desired precision. As Sharir et al. mentions, the training stages in large language models carry high processing loads [43]. These high loads can cause extended training times even running in specialized hardware.

2.2.3 Generative Pre-trained Transformer Model Generative Pre-Trained Transformer (GPT) is a large language model originally proposed by Radford et al. [38]. In their work, the authors propose a method that simulates the cognitive learning of human beings. Along with this mechanism, an architecture called transformers is proposed. This is considered the state of the art for large language models due to its improvement in the time required for training. GPT consists of a pre-trained transformer. The training process uses large datasets. These datasets contain large amounts of information written by humans, which feeds the model with extensive context and allows it to generate more accurate answers to arbitrary queries. The generation of text is the main objective of GPT models; mainly text that mimics human writing. This model was introduced with the objective of generating an implementation

that is agnostic to the task. GPT has demonstrated outstanding results on diverse domains, even outperforming task-specific models.

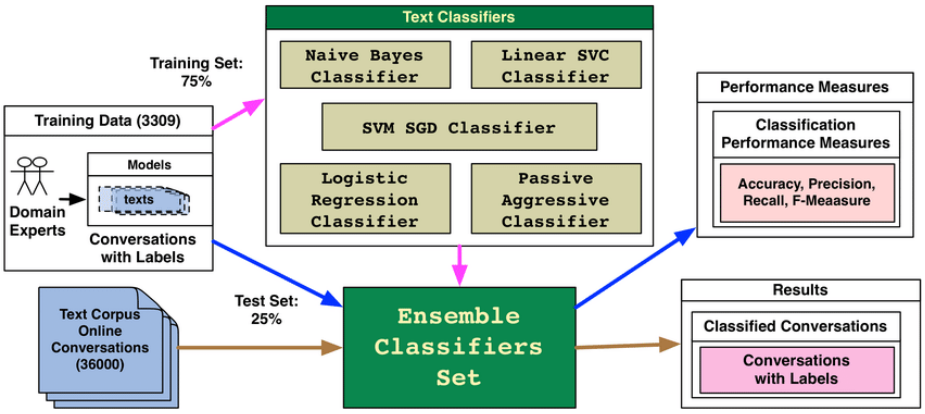

Fig. 1. Text Classification Architecture [39].

2.2.4 TF-IDF In addition to AI, there are also string handling deterministic techniques. These techniques require a bag of words which assigns a weight to the words using specific techniques such as Term Frequency – Inverse Document Frequency (TF-IDF). Inverse document frequency is a statistical measure that evaluates how relevant a word is to a document in a collection of documents. This is done by multiplying two metrics: how many times a word appears in a document and the inverse document frequency of the word in a set of documents. As mentioned in Encyclopedia of Machine Learning by Sammut, C et al. [41], this technique focuses on representing text documents as vectors in order to determine specific weights for each term. Once this process is completed, there are different machine learning algorithms that can be used, such as Voting Classifier, Logistic Regression, Linear SVC. In Fig. 1 can be seen a traditional text classification architecture in which machine learning algorithms, called text classifiers, are implemented.

2.2.5 Vulnerability Identification

Vulnerability Identification is part of a larger process known as vulnerability management. In a vulnerability management process, besides the Vulnerability Identification, it is required to assess, prioritize, and remediate vulnerabilities [31]. Nonetheless, the vulnerability identification stage is crucial as it triggers all the subsequent stages of the vulnerability management process.

There are diverse methods to identify vulnerabilities. Penetration testing is among the preferred cybersecurity activities in industry [30]. However, it is an expensive approach since a professional red team has to be hired to do the job. The use of automated software tools has helped to identify vulnerabilities in a cost-effective way. For instance, fuzzing tools help to identify known and unknown vulnerabilities in software. However, fuzzing tools typically suffer from high false positive rates [26].

On the other hand, the identification of vulnerabilities based on public vulnerability lists, like MITRE's CVE list, is just one function among many in a broader security ecosystem. These lists serve as a ground-truth for known vulnerabilities that various security systems can use to evaluate a target system. While these lists help in discovering known vulnerabilities, they don't identify new ones. However, there's evidence that most vulnerabilities exploited in real-world attacks are already known [35].

A significant challenge with using public vulnerability lists is how to keep up with the increasing volume of CVE records published daily [5, 40]. It is common practice for these lists to be updated manually by security teams, leading to delays in detecting known vulnerabilities. This manual update process is often time-consuming and can create gaps in the detection of vulnerabilities due to the high volume of updates required.[2]

2.2.6 CVE Data Sources

[2] https://forum.greenbone.net/t/cve-2023-6933-seems-to-be-missingmissing/16750

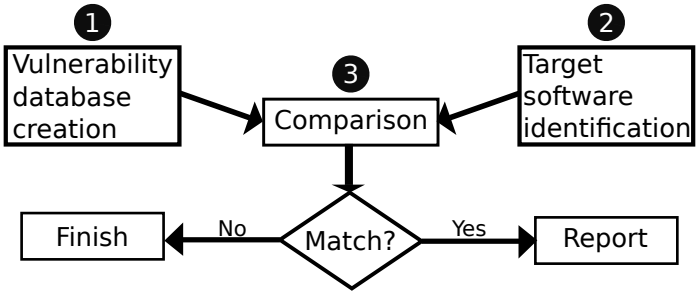

Fig. 2. Basic tasks of our proposed vulnerability identification system: (1) vulnerability database creation; (2) target software identification; and (3) comparison between the target software and the database records.

2.2.6.1 MITRE CVE File: MITRE allows to download of a file in different text formats with all the vulnerabilities found. These formats can be HTML or CSV. The *allitems* file downloaded on May 1, 2024, contains approximately 311 261 vulnerabilities. The files can be accessed at [9].

2.2.6.2 Kaggle CVE: A data set of cybersecurity threats and their importance from NIST. Kaggle is a dataset source where you can find pre-created datasets with a wealth of information on various topics. This particular dataset has information on 89 960 vulnerabilities. This information can be accessed at [23].

2.2.6.3 National Vulnerability Database: This data source is the United States National Vulnerability Database. Among its services is the ability to provide a series of API services which allow obtaining a list of CVEs that are stored in NVD, these give the possibility of automating the data collection process. When executing the data collection through the postman tool, approximately 183,635 vulnerabilities were obtained. The format of the results is JSON, containing extensive details of each of the vulnerabilities. This information can be accessed at [10].

2.2.6.4 CVE and CWE Mapping Dataset (2021): This dataset contains more than 150,000 vulnerabilities reported in NVD during the period 2002-21. It includes the various information of the

vulnerability, including the unique ID, description, severity, severity scores, and CWE category in which the vulnerability falls. Multiple records were used if a vulnerability is assigned to more than one category. The information can be accessed at [3].

3 MITRE CVE-Based Vulnerability Identification Proposal

Software vulnerability identification processes must perform 3 basic tasks: (1) to create a database of vulnerable software in its affected versions as `(software_name, version(s))` pairs; (2) to identify the target system software as a `(software_name, version)` pair; and (3) to compare the software found against the database of vulnerable software. If a match is found, the vulnerability identification reports the target software as *vulnerable*. Otherwise, no vulnerability is reported. In what follows, we refer to the pairs `(software_name, version)` as *software signatures*. The entire process is depicted in Fig. 2.

Our work seeks to automate the overall Vulnerability Identification process by automating its individual steps. In Sect. 3.1 we describe the architecture of the proposed approach including both, the creation of the software vulnerabilities database, and the vulnerability identification system. In Sect. 4 we detail our implementation of the overall system.

3.1 Architecture

Figure 3 shows our proposed high level architecture. This architecture considers two stages. The *Feature Extraction Stage,* consists in the automatic population of the database with signatures of vulnerable software. On the other hand, the *Validation Stage,* aims to identify if a software in a specific version is vulnerable to any of the CVEs whose signatures are stored in the database generated in the previous stage. We now describe both stages in more detail.

3.1.1 Feature Extraction Stage This process is responsible for generating the signatures that are stored in the database. For this, the proposed architecture is made up of several steps, which are: **Obtaining the CVEs in CSV format:** The process starts by obtaining CVE records as comma-separated value (CSV) tuples [33]. The CSV file contains the main elements of a CVE record, such as the CVE-ID and its description. **Signature**

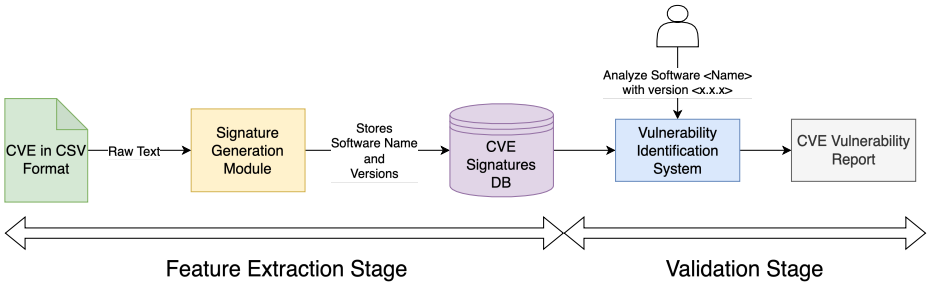

Fig. 3. Architecture of our proposed approach to develop automated vulnerability tests.

Generation Module (SGM): This module must be capable to automatically generate vulnerable software signatures. A vulnerable software signature consists of a `(software_name, version)` pair. Since a single CVE record can comprise multiple versions of a software, it might be needed to create multiple signatures per CVE. The information needed to create vulnerability signatures is available in CVE descriptions (see Table 1). The Signature Generation Module component is one of the main contributions of this work. We propose two possible SGM proposals.

- One implementation is based on an heuristic algorithm. This means that they are a series of predetermined steps that are designed to capture data through evaluations based on common patterns identified in CVE texts.
- The second one leverages an artificial intelligence model. This implementation focuses on the use of deep learning models to determine and extract valuable information from the text in a CVE. Specifically, this proposal aims to use pre-trained models with a large amount of information for text generation.

CVE Signatures Database: Finally, the automatically extracted signatures are stored in a database as *key-value* pairs, where the *key* is the CVE-ID and the *value*, the corresponding signatures extracted from that CVE. Thus, our vulnerability database is composed of `(key: CVE-ID, value: software signature)` pairs.

3.1.2 Validation Stage The main component of the Validation Stage is the Vulnerability Identification System (VIS). The VIS takes as input a software signature and checks whether it matches any of the records stored in the database created by the SGM. The input signature might come from an automated fingerprinting software, a human who wants to know if particular software is linked to CVE records, or any other source. We do not elaborate on this input as it is beyond our scope.

The VIS checks if the input signature is part of the vulnerability records stored in the database. If that is the case, a vulnerability report is created. Otherwise, the system finishes its execution.

For VIS Implementation, we also propose two possible proposals:

- One implementation is based on an heuristic approach. Basically, this means that the signature information must be compared with the user input signature using string comparison algorithms.
- The second one also leverages an artificial intelligence model. This with the objective of evaluating whether a pre-trained model can determine whether a software product is vulnerable or not to a specific CVE, receiving as input the description of the CVE and the input signature.

4 Implementation

This section delves into the implementation of our proposed design by explaining how we built the components shown in Fig 3.

To obtain CVE records in CSV format, it is possible to automate the download from MITRE's website [33]. We implemented this process as a *Cron job* that regularly checks for the latest CVE records available. This data is later processed by the Signature Generation Module (SGM).

The SGM lies at the core of our proposed architecture. We propose two different approaches to implement it. The first one is a heuristic algorithm that, through predefined steps, decomposes the text so that the required data can be extracted from CVE descriptions. The second proposal consists of a Generative Pre-trained Transformer (GPT) model that, by means of a specific prompt, manages to extract vulnerable software signatures from CVE descriptions. We now describe both implementations in more detail.

4.1 Heuristic SGM

This implementation proposal consists of a heuristic algorithm that takes as input CVE descriptions and outputs vulnerable software signatures. The Heuristic SGM performs 3 steps divided in 8 subtasks, as depicted in Fig. 4. For a better understanding of each of them, we use the description of CVE-2022-45937 as an example (see Table 1).

4.1.1 Preprocess Text. In this step, irrelevant information is removed, and specific substitutions are made to standardize the analysis. More specifically, we perform the following changes to the text:

- **Space Comparison Symbols:** In this step, the symbols and specific words shown in Table 2 are identified within the CVE description. A blank character is added before and after each of them, if there is none. A function implemented in

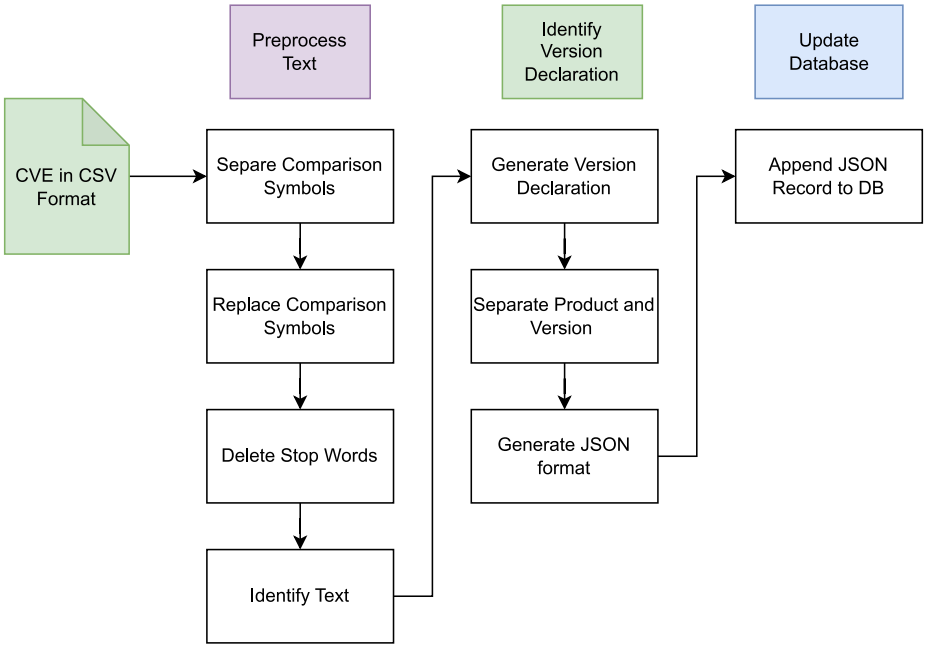

Fig. 4. Heuristic SGM Implementation.

Python that performs this function can be seen in Listing 1. For example, <V3.4 becomes < V3.4.

Listing 1. Split Symbols Python Function

```
def splitSymbols(text):
    words = re.search('(<)[a-zA-Z]|(<=)[a-zA-Z] |
        (=)[a-zA-Z]|(>)[a-zA-Z]|(>=)[a-zA-Z]', text)
    if words:
        match = words.group()
        white_space = " "
        last_two = match[-2:]
        text = text.replace(last_two, white_space.join(last_two))
    return text
```

- **Replace Comparison Symbols:** This step consists of replacing the symbols and specific words as shown in Table 2. This is important because later in the process, all symbols of the text are removed.
- **Delete Stopwords:** The elements of the text that are characterized as stopwords are deleted. As mentioned by Sarica et al. in [42], "the uninformative words, often

referred to as “stopwords”, need to be removed in the pre-processing step, in order to increase signal-to-noise ratio in the unstructured text data”. For this step, the list of stopwords is taken from the Python *nltk.corpus* [37] library.

– **Identify Text:** This is the last stage of preprocessing and consists of labeling the code for later analysis. Each word is going to be tagged with a specific key word. This labeling process will be carried out using regular expressions. The system labels are:
  - *Comparators:* The words *lower*, *equal* or *greater*.
  - *Version:* All the words with the pattern $V < Number > . < Number > \cdots < Number >$. This pattern is recognized with or without the initial V. For example V2.3.4 or 2.4.3. Also, the words $version$ or $versions$ are tagged as *Version*.
  - *Context:* They are all those words that were not tagged as *Version* or *Comparator*. These words contain information about the software product as well as the context of the vulnerability.

A function implemented in Python that performs this function can be seen in Listing 2.

Listing 2. Identify Labels Python Function.

```
def identifyLabelsInText(text):
    v_values_regex = "([vV]{1}?)\d{1,5}\.\d{1,5}|\.\d{1,5}|
    \.\d{1,5}|\d{1,5}\.\d{1,5}\.\d{1,5}|\.\d{1,5}|\.\d{1,5}"
    values_regex = "\d{1,5}\.\d{1,5}"
    version_regex = "(version+)([s]?)"
    compare_regex = "(?:greater|lower)"
    for token in text:
        if re.match(v_values_regex, token.get("text")) or
          re.match(version_regex, token.get("text")):
            token.update({"type": "VERSION"})
        elif re.match(values_regex, token.get("text")):
            token.update({"type": "MODEL"})
        elif re.match(compare_regex, token.get("text")):
            token.update({"type": "COMPARE"})
    return text
```

4.1.1.1 Identify Version Declaration. This phase consists of identifying what we call a version declaration. This operation consists

Table 2. Symbols replacement used in the Heuristic approach.

Symbol	Replacement	Original Word	Replacement
=	equal	before	lower
>	greater	up to	versions lower
<	lower	prior to	versions lower
		and prior	greater versions

of identifying the pattern[n] of labels **context** + **version** + **comparator**. In the first description shown in Table 1, there are 3 declarations. If the first one is analyzed, the **context** corresponds to *"A vulnerability has been identified in APOGEE PXC Series (BACnet)"*. The context contains the name of the software and additional text that is not relevant. This first stage of the statement identification process consists of finding that a sequence of words has the aforementioned labeling pattern. For this, in Listing 3 you can see the code used. This code is based on the labeling process previously discussed and, by using variables that contain information from the token above, can be determined that indeed the sequence of tokens corresponds to a declaration. To remove the irrelevant text, the system extracts the last n words and assigns them as the software product name. Our experiments showed that $n = 4$ provides good results to identify the software product name. Following up with the example, the extracted name would be *"APOGEE PXC Series (BACnet)"*.

Once all the possible version declarations in the text have been identified, we proceed to separate the software products from the versions, separating the software name from the comparison blocks (**version** + **comparator**). This process generates a result like the following {module: APOGEE PXC Series (BACnet), vulnerable_versions: versions $<$ V3.5.5}. The code used to carry out this separation of the elements of a declaration can be seen in Listing 4. It iterates over the declaration list, and using the labels it determines which part of the declaration corresponds to the name of the module and which corresponds to the affected versions.

Listing 3. Search Version Declarations Python Function.

```
def searchForVersionDeclarations(text):
    versionDeclaration = []
    prev_token_index = ""
    prev_token_type = ""
    for index, token in enumerate(text):
        token_type = token.get("type")
        if (token_type != "COMPARE") and prev_token_type == "VERSION":
            versionDeclaration.append(index)
        elif (token_type == "MODEL") and prev_token_type == "COMPARE":
            versionDeclaration.append(index + 1)
        prev_token_type = token.get("type")
        prev_token_index = index
    return versionDeclaration
```

Listing 4. Separe Declarations Python Function.

```
def separateDeclarations(versionDeclaration, text):
    last_declaration = 0
    declarations = []
    for declaration in versionDeclaration:
        current_declaration = {"module":[],"versions":[]}
        key_name = "module"
        is_version = False
        for index in range(last_declaration, declaration):
            is_version = text[index].get("type") == "VERSION" or
            (text[index].get("type") == "MODEL"
            and text[index-1].get("type") == "COMPARE")
            if is_version and key_name == "module":
                key_name = "versions"
                if text[index-1].get("type") == "COMPARE":
                    current_declaration[key_name].append(text[index-1])
            current_declaration[key_name].append(text[index])
        declarations.append(current_declaration)
        last_declaration = declaration + 1
    return declarations
```

4.2 GPT SGM

This implementation proposes the use of the OpenAI's GPT-3 pre-trained models [27]. These models can be used through the official OpenAI API. This API is accessed through HTTP requests. The request requirements include selecting a model to run and a text query. That query is called prompt. The

implementation of this section is programmed in Python. For this, the *OpenAI* library is used. An example of connection to the OpenAI API can be seen in the function 5 The specific steps of this process can be seen in the Fig. 5

This solution is simpler from an architectural point of view, as it is delegated to a third-party component. Previous experiments have demonstrated the ability of the gpt-3.5-turbo-0125 (GPT3) model to perform actions based on text analysis as well as text generation [6]. Consequently, the GPT3 model could be utilized to handle basic queries in cybersecurity due to its proven capacity for text analysis and generation.

Listing 5. Call OpenAI Model Python Function.

```
def call_open_api(cve_prompt):
    client = OpenAI()
    fail = False
    generated_text = ""
    try:
        response = client.chat.completions.create(
        model="gpt-3.5-turbo-0125",
        response_format={ "type": "json_object" },
        max_tokens=3000,
        messages=[
            {"role": "system", "content": "You are
            a helpful assistant designed to output JSON."},
            {"role": "user", "content": cve_prompt}
            ]
        )
        generated_text = response.choices[0].message.content
    except Exception as e:
        retry_time = 2
        print(f"Error {e}. Retrying in {retry_time} seconds...")
        time.sleep(retry_time)
        return call_open_api(cve_prompt)

    return generated_text, fail
```

4.2.0.1 Prompt Execution. At this stage, a request is made to a GPT text processing model. It consists explicitly of indicating that it is required to extract the relevant data of the respective software products and versions found in CVE descriptions. This prompt request is build using the description and the CVE-ID.

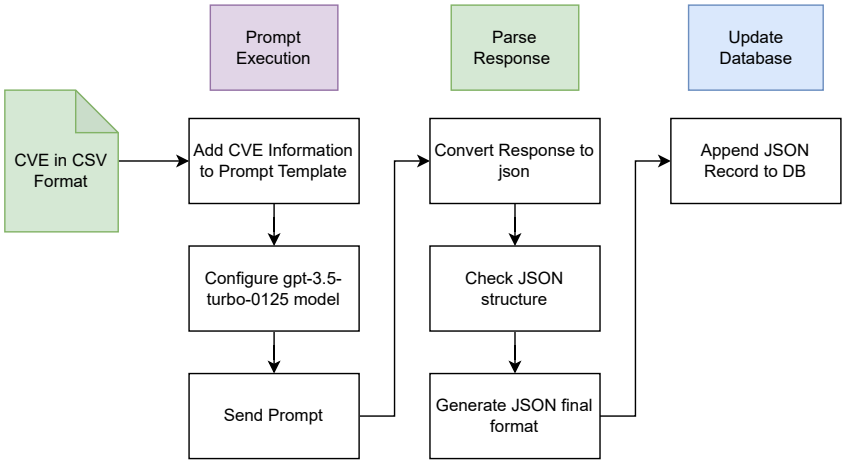

Fig. 5. GTP SGM Architecture.

At this stage, the response from the model is expected to be a list of signatures in JSON format containing the name of the software and the vulnerable versions. The Appendix A.1 contains the prompt used in this section. The expected output of the execution of the prompt for the CVE-2022-45937 can be seen in Listing 6.

4.2.0.2 Parse Response. In this step, the response of the model is validated, to check its format and standardize the information so that it is ready to be uploaded to the database. If the response is in an unsupported format, our system retries until the response complies with what is expected, it performs a configurable number of retries, in case the maximum number is reached, the CVE analysis will be discarded. Among the validations is determining whether the CVE has not been reserved as a candidate by an organization, and it is also validated that the JSON format is properly written. These validations are implemented using Python, as can be seen in 7.

Listing 6. JSON Ouput Example.

```
{
"CVE_ID": "CVE-2022-45937",
"vulnerable_versions": [
    {"module": "APOGEE PXC Series (BACnet)", "versions":
    [{"version": "versions>=3.5.5"}]},
    {"module": "APOGEE PXC Series (P2 Ethernet)", "versions":
    [{"version": "versions>=2.8.20"}]},
    {"module": "TALON TC Series (BACnet)", "versions":
    [{"version": "versions>=3.5.5"}]}
    ]
}
```

Listing 7. Analyze Response Python Function.

```
def analyze_cve(token, cve_prompt, cve, database, database_name, attemp):
    if cve["DESCRIPTION"] == CANDIDATE:
        print("CANDIDATE "+ cve["CVE-ID"], flush=True)
        clean_response, success =
        {"CVE_ID": cve["CVE-ID"], "vulnerable_versions": []}

    elif cve["DESCRIPTION"] != CANDIDATE:
        response, fail = call_open_api(token, cve_prompt)
        clean_response, success = clean_result(response, cve["CVE-ID"])

    if success:
        database["cves"].append(clean_response)
        print("Analyzed "+ cve["CVE-ID"], flush=True)
        save_result(database, database_name)
    else:
        if attemp < 2:
            print("Retrying cve " + cve["CVE-ID"], flush=True)
            analyze_cve(token, cve_prompt, cve, database,
                database_name, attemp + 1)
        else:
            print("Abort cve " + cve["CVE-ID"], flush=True)
```

4.2.0.3 Update Database. This step consists solely of information storage. The main condition for the implementation of this module is that the information is saved in JSON format. For the experiment, the decision was made to use the MongoDB database in its free version [4]. The adapter with the database is developed in Python. This is based on the use of the *pymongo* library. Which simplifies the connection to the database,

allowing to only have to use a few lines of code. Among the data required to use this library are the server as an object, *pymongo.MongoClient* and the name of the database.

4.3 Vulnerability Identification System (VIS)

The purpose of this stage is to determine if there is a signature in a CVE that is identified as vulnerable when compared with the database. The proposed workflow for this stage can be seen in Fig. 6.

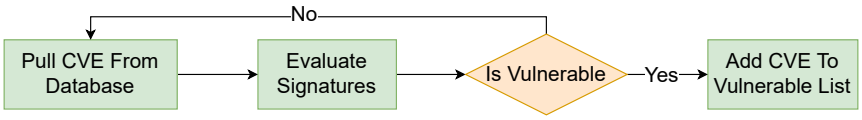

Fig. 6. Vulnerability Identification System (VIS) Workflow.

4.3.1 Pull CVE. In order to use the data that were previously generated, they must be retrieved from the database. We propose to obtain the CVEs that are to be analyzed using the range of dates in which they were uploaded as a filter. This in order to reduce the amount of reprocessing.

4.3.2 Evaluate Signatures. Two possible implementations for this design are provided. The first consists of a fully heuristic process which evaluates the signatures from predefined processes. The second proposal consists of making a request to the GPT model. The request contains the CVE signatures and the software product information.

4.3.3 Heuristic VIS. This proposal consists of using heuristic algorithms that, through knowledge of the format and structure of the information, can carry out the validation. This process can be seen in Figure 7. For this step to work, it requires the signatures for each preprocessed CVE and also the input from the user. This input refers to the software product that is going

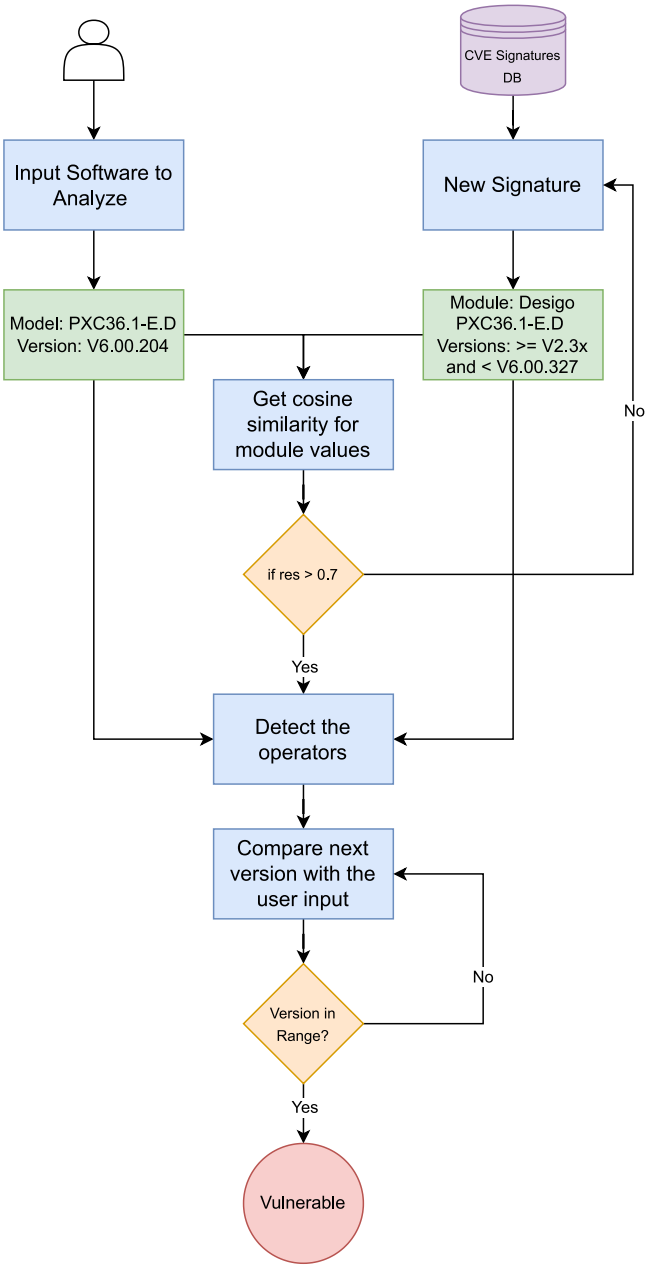

Fig. 7. Example of Heuristic VIS string comparison Workflow. Based on CVE-2019-13927.

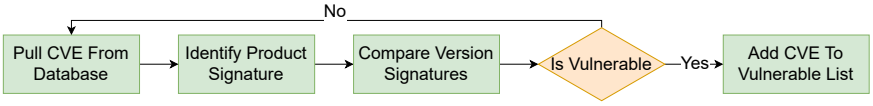

Fig. 8. Heuristic VIS Implementation.

to be evaluated. The implementation flowchart can be seen in Figure 8. In the flowchart, there are two functions:

- **Identify Software Product Signature:** Consists of evaluating if any of the CVE software module name matches the software being analyzed entered by the user. For this, a text comparison is performed, an example of this comparison is the implementation that uses cosine similarity, it is a metric, helpful in determining, how similar the data objects are irrespective of their size and the equation to calculate it is $S_C(x, y) = x \cdot y/||x|| \times ||y||$. The values of this coefficient can vary between the range of -1 and +1. Our previous experimental evaluations determined that the optimal value to use in this step is 0.7. This section can be seen in Figure 7 as the first operation between the user input and each of the database signatures.
- **Compare Version Signature:** Once a software product signature matches, we proceed to compare the version signature. For this, a series of predefined steps are carried out, which analyze the string value of the signature. These steps consist of determining the type of the signature. There are 3 types, the first is *all versions*, in this case all software versions of the product are vulnerable, the second is a *specific version* which is vulnerable, so the type of comparator would be the same. Finally, the third type includes a *range of vulnerable versions*, such as signatures in Listing 6. Once the type has been identified, the operator in question is used to compare the software version with the version signatures identified in the CVE. For this comparison, Python boolean operators are used which, thanks to the fact that the text is normalized, allows comparing the ASCII and Unicode representation

of the strings so that operators such as $\leq$ and $\geq$ are perfectly usable. From an implementation point of view, this is one of the most complicated points. This is because it must be correctly determined whether the declaration uses operators or not, and from there use those same operators to compare the text. So to make it clearer for the reader, the diagram in Figure 7 has the specific flow of each step.

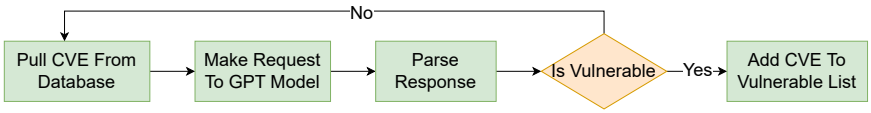

Fig. 9. GPT VIS Implementation.

4.3.4 GPT VIS. This approach focuses on using the GPT-3.5-turbo-0125 model to determine if the evaluated software product is vulnerable or not. To do this, it is proposed to create a prompt that contains the data of; product under evaluation as well as the signatures of the CVE that is being evaluated. Figure 9 shows the architecture of this proposal.

- **Make Request To GPT Model:** This architecture process should focus on the elaboration of the prompt. For this, the software product data is taken into account, name + version. The values of the signatures belonging to the CVE that is being evaluated must also be included in the prompt. This is so that the model has enough information to make the comparisons. The output of this process is expected to be a text containing information to determine if the CVE is vulnerable or not. The Appendix A.2 contains the prompt used in this section.
- **Parse Response:** This task aims to receive the model's response as input, to parse it and analyze the text to determine if there is a pattern that indicates whether the software product is vulnerable. In addition, if it is, it must

output a flag that tells the next task to add the CVE to the list.

4.3.5 CVE Vulnerability Report. It consists of the generation of a report that shows the CVEs which affect the software and version that were given as input to the system.

5 Experimental Results

We evaluate the proposed approach by means of two experiments. The first experiment aims to evaluate and compare the Heuristic and GPT Signature Generation Modules (SGMs) to create the vulnerability signature database. The purpose of the second experiment is to determine the capability of our system to identify vulnerabilities in a subset of target systems. As mentioned in Sect. 4, we implement Heuristic and GPT Vulnerability Identification Systems (VISs), both of which are evaluated here.

The heuristic experiments run on a computer with Ubuntu 20.04 as operating system, using an Intel Core i7 10th Gen CPU and 16GB of DDR4 RAM memory. For the cloud-based GPT service, Azure is the provider used to run the models. This provider allows scaling and adjusting processing requirements on demand, which makes it difficult to provide detailed system specifications.

5.1 Database Generation Experiment

This experiment consists of generating a database of vulnerability signatures. Vulnerability signatures are `(software_name, version)` pairs of vulnerable software. We analyzed 39 534 CVEs published in 2 023 [33]. This represents the 100% of the total amount of CVEs published during that year. We deem this amount of CVEs sufficient to obtain representative results of the performance of our proposed approach.

In this experiment, we assume that all CVEs must clearly state the vulnerable software and its versions. The only exception to this assumption are CVEs labeled as ***REJECTED*** or ***RESERVED***. These kind of CVEs are the only cases in which the modules should not identify any vulnerability signatures.

We started by using a CSV file of the 39 534 CVEs to be analyzed. Then we processed the CSV file using both, the Heuristic and GPT, SGMs implemented.

There are 4 possible outcomes for both SGM implementations. We call it a *true positive* (TP) when at least one correct

vulnerability signature is created from a valid CVE record; a *true negative* (TN) when no vulnerability signature is created from a reserved CVE record; a *false positive* (FP) when a model creates a mistaken vulnerability signature from a valid CVE record; and a *false negative* (FN) when no vulnerability signature is created from a valid CVE record. The result for each approach can be seen in Table 3. From these metrics it is possible to compute the precision ($\frac{TP}{TP+FP}$), recall ($\frac{TP}{TP+FN}$), and accuracy ($\frac{TP+TN}{TP+FP+TN+FN}$) for both implementations. Table 4 shows the results obtained during the experiments.

Table 3. Vulnerability SGM results for the heuristic and GPT implementations.

SGM	TP	TN	FP	FN
Heuristic	14 098	14 842	0	10 594
GPT	24 692	14 842	0	0

Table 4. Vulnerability SGM metrics for the heuristic and GPT implementations.

SGM	Precision	Recall	Accuracy	F1 Score
Heuristic	1,0	0,571	0,732	0,727
GPT	1,0	1,0	1,0	1,0

During the execution of the experiments, we also analyzed metrics related to the execution time for both. These results are detailed in Table 5. It is worth noting the significant difference in the total time required by the GPT SGM (46 982,632 seconds $\approx$ 13 hours) and the Heuristic SGM (240,610 seconds $\approx$ 4 minutes).

We also performed a Multifactor Analysis of Variance (ANOVA) to better understand the factors that affect the execution time of the implemented SGMs. In the experiment, two variables are analyzed. X_1 being the SGM used and X_2 the CVE description length [32]. When carrying out the analysis, the results reaffirm what was seen in Table 5. That

the implementation used has a direct impact on the duration of the process. Furthermore, it also reflects that the length of the description has an impact on the execution time, mainly in the implementation that uses GPT.

Table 5. Summary of the execution time required by the Heuristic SGM and GPT SGM to populate the vulnerability signature database.

		Time per CVE (s)	
SGM	**Total Time (s)**	**Average**	**Standard Deviation**
Heuristic	240,610	0,009	0,008
GPT	46 982,632	1,188	3,462

5.2 CVE Identification Experiment

In this experiment, we evaluate two distinct models for vulnerability identification: a heuristic-based system, referred to as Heuristic VIS, and a system based on the GPT-3 API, referred to as GPT-based VIS (VISs). The goal is to determine which system can better identify software vulnerabilities using a given dataset of Common Vulnerabilities and Exposures (CVEs).

Listing 8. Experiment Ground Truth Input Example.

```
{
    "cves": [
        {
            "CVE-ID": "CVE-2023-40558",
            "isVulnerable": {
                "Module": "eMarket Design YouTube Video Gallery",
                "Version": "3.3.5",
                "Vulnerable": true
            }
        },
        {
            "CVE-ID": "CVE-2023-40558",
            "isVulnerable": {
                "Module": "eMarket Design YouTube Video Gallery",
                "Version": "4.0.0",
                "Vulnerable": false
            }
        }
    ]
}
```

5.2.1 Input Data The input for the experiment is a JSON array containing multiple CVE entries. Each CVE includes details about the software module, its version, and a boolean indicating whether this module-version combination is vulnerable or not. An example of the input JSON structure can be seen in Listing 8:

The dataset consists of 600 test cases, 344 representing vulnerable scenarios and 256 representing non-vulnerable scenarios, derived from 383 unique CVEs.

5.2.2 Generation of Test Data To create the test data for this experiment, the following steps were taken:

1. **Random Selection of CVEs:** We randomly selected 383 CVEs from a total population of 35,954 CVEs published in 2023. This sample size was determined using the following formula for sample size calculation:

$$n = \frac{N \cdot Z^2 \cdot p \cdot (1-p)}{(N-1) \cdot E^2 + Z^2 \cdot p \cdot (1-p)},$$

where:
  - **N** is the total population size.
  - **p** is the estimated proportion of the characteristic of interest in the population. We use $p = 0.5$ to maximize the sample size, as it represents the highest variability and thus yields a conservative estimate.
  - **E** is the margin of error (0.05 for a 5% margin of error).
  - **Z** is the Z-score for a 95% confidence level (approximately 1.96).
  - **n** is the desired sample size.

  This formula is derived from the standard sample size calculation in statistics [7], which balances population size, error margin, confidence level, and variability to determine the optimal sample size.
2. **Manual Verification of CVEs:** Each of the selected 383 CVEs was manually reviewed by the authors to determine the relevant modules and versions. This process involved assigning the ground truth for whether a specific module-version combination is vulnerable to the corresponding CVE.
3. **Creation of JSON Data:** After manual verification, a JSON file was created containing all the relevant information for each CVE, including the module, version, and whether it was deemed vulnerable or not. This JSON file serves as the input data for the experiment described in the previous section.

The random selection of CVEs and the subsequent manual verification ensured a diverse set of test cases for evaluating the heuristic-based and GPT-based vulnerability identification systems. By using a rigorous method to determine the sample size and a thorough review process for the ground truth, the data generated provides a robust basis for conducting the experiment.

5.2.3 Experimental Procedure

The experiment follows these steps:

1. **Retrieve CVEs:** The specific CVE to be evaluated is retrieved from a pre-generated database.

2. **Validation by VIS Models:** The input data is processed by both the Heuristic VIS and GPT-based VIS models. Each model evaluates the vulnerability of a given module and version based on the corresponding CVE. The expected outcome is a boolean indicating vulnerability.
3. **Validation of Results:** The output from each model is compared to the expected ground truth to determine if the result was correct. Based on this validation, the following metrics are calculated:
   - **True Positive (TP):** Cases where the model correctly identified a known vulnerability.
   - **False Positive (FP):** Cases where the model incorrectly identified a vulnerability in a non-vulnerable module-version.
   - **True Negative (TN):** Cases where the model correctly identified a non-vulnerable module-version.
   - **False Negative (FN):** Cases where the model failed to identify a known vulnerability.
4. **Document Results:** The results are recorded, and the next CVE is processed.

5.2.4 Metrics and Evaluation

To evaluate the performance of the Heuristic VIS and GPT-based VIS models, we use the following metrics:

- **Accuracy:** This metric measures the proportion of correctly identified cases out of the total cases, providing a general indication of the model's correctness. It is calculated as $\frac{TP+TN}{TP+TN+FP+FN}$.
- **Recall:** This metric assesses the model's ability to detect true positive cases. It is calculated as $\frac{TP}{TP+FN}$. A high recall indicates effective detection of vulnerabilities.
- **F1 Score:** This metric combines precision and recall, providing a balanced view of the model's performance. It is calculated as $2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$, where Precision is $\frac{TP}{TP+FP}$.

By using these metrics, the experiment aims to determine which model is more effective at identifying vulnerabilities in

a variety of software modules and versions. The results from this experiment will guide future improvements in vulnerability identification systems.

5.2.5 Results Table 6 shows results for the heuristic methodology, consistently revealing low rates of accuracy, recall, and F1 Score. Heuristic VIS has all the same results because it likely uses a fixed set of rules or patterns to make its determinations. This approach lacks adaptability and might not consider the full range of possible variations in the data. The heuristic methodology tends to be rigid, leading to consistent outputs regardless of the nuances in the test runs.

Table 6. Results for Heuristic VIS.

Run	TP	TN	FP	FN	Accuracy	Recall	F1 Score
1-10	82	220	36	262	0,503 333	0,238 372	0,354 978

Table 7 displays results for the GPT VIS method, highlighting high levels of accuracy, recall, and F1 Score. These results indicate better consistency and reliability compared to the heuristic method.

Table 7. Results for GPT VIS.

Run	TP	TN	FP	FN	Accuracy	Recall	F1 Score
1	325	206	50	19	0,885 000	0,944 767	0,904 033
2	322	213	43	22	0,891 667	0,936 047	0,908 322
3	326	205	51	18	0,885 000	0,947 674	0,904 300
4	326	219	37	18	0,908 333	0,947 674	0,922 207
5	322	212	44	22	0,890 000	0,936 047	0,907 042
6	329	213	43	15	0,903 333	0,956 395	0,918 994
7	325	213	43	19	0,896 667	0,944 767	0,912 921
8	327	208	48	17	0,891 667	0,950 581	0,909 597
9	327	208	48	17	0,891 667	0,950 581	0,909 597
10	326	215	41	18	0,901 667	0,947 674	0,917 018

In the GPT VIS experiment, different results in each iteration can be attributed to variations in the data and the inherent

randomness in the model's initialization and learning process. The use of different batches or data shuffling during training can lead to subtle changes in predictions, impacting metrics like accuracy, recall, and F1 score. Despite these fluctuations, the GPT VIS model generally maintains high performance, demonstrating its flexibility and adaptability to diverse data patterns.

Table 8. Average Results for Heuristic VIS and GPT VIS.

	Accuracy		Recall		F1 Score	
Method	Mean	Std	Mean	Std	Mean	Std
Heuristc	0,503	0,000	0,238	0,000	0,354	0,000
GPT	0,894	0,008	0,946	0,006	0,911	0,006

Table 8 shows the average results for the Heuristic VIS and GPT VIS methods. It can be observed that the GPT VIS implementation outperforms the heuristic approach in all metrics. GPT VIS has significantly higher mean accuracy, recall, and F1 score, indicating a more reliable and consistent performance. The lower standard deviations suggest a more stable output compared to the heuristic approach, which has consistent but lower results.

6 Discussion

Our findings indicate that it is indeed possible to automate the generation of a database based on CVEs. By analyzing the results in Table 4 in detail, we can see that despite the fact that both methods manage to identify signatures, the GPT SGM has a greater capacity to identify software products, which can be seen with the value of false negatives. In the Heuristic SGM, 26,79% of the CVEs did not yield identified products, while in GPT SGM 0%. This is verified by reviewing the recall value in Table 4, which clearly shows a greater capacity for this SGM implementation to obtain relevant information from the descriptions. If the results are analyzed based on the time it takes to analyze a CVE, with the results in Table 5 it is known that the GPT SGM takes an average of 132 times longer than the Heuristic SGM to obtain a result. In contrast, the heuristic-based approach had a significant number of false negatives (10,594), resulting in a recall rate of only 0.571. This finding suggests that the heuristic approach, while achieving high precision due to the absence of false positives, lacks the sensitivity required to capture all valid vulnerability signatures, leading to a lower recall rate. Overall, the GPT-based approach demonstrated superior accuracy and reliability in generating vulnerability signatures, albeit at the cost of significantly increased execution time. But still better than doing it manually. The heuristic approach, while faster, had a higher likelihood of missing valid signatures due to its lower recall rate. Future work could focus on optimizing the GPT-based approach for efficiency or exploring hybrid approaches to balance accuracy and speed.

The second experiment evaluated the performance of two vulnerability identification systems: a heuristic-based system (Heuristic VIS) and a GPT-based system (GPT VIS). The results clearly indicate that the GPT-based approach outperformed the heuristic-based approach across all key metrics, demonstrating higher accuracy, recall, and F1 score. The experiment used a dataset of 600 test cases derived from 383 unique CVEs,

allowing for a thorough evaluation of the models' effectiveness in identifying vulnerable software modules and their versions.

In the heuristic-based system, the results showed consistently low rates of accuracy, recall, and F1 score. This outcome is likely due to the static nature of heuristic rules, which may not be able to adapt to variations in the data or capture complex patterns. The lack of adaptability led to a high number of false negatives (FN = 262 per run), indicating that the heuristic approach frequently missed valid vulnerabilities.

The GPT-based system achieved significantly higher performance, with near-perfect metrics in terms of accuracy, recall, and F1 score. This suggests that GPT's adaptability, deep learning capabilities, and context awareness make it better suited for complex data scenarios. However, the increased accuracy and recall come at the cost of higher computational resources and longer execution times, with the GPT system requiring over 13 hours to process the dataset compared to the heuristic system's 4 minutes.

Recall is a critical metric in vulnerability identification because it measures a system's ability to detect all true positive cases. A high recall rate indicates that the system is effective at identifying vulnerabilities, minimizing the risk of undetected security risks. In this experiment, the GPT VIS system's perfect recall demonstrates its effectiveness in capturing known vulnerabilities, making it a more reliable choice for vulnerability detection.

Overall, the results suggest that while the heuristic approach offers faster execution times, it lacks the sensitivity required to detect all vulnerabilities. The GPT-based approach, on the other hand, provides higher accuracy and recall but at the cost of longer execution times. Future work could explore ways to optimize the GPT system for efficiency or investigate hybrid approaches that combine the strengths of both methodologies.

7 Conclusion

In this work, we investigated to what extent future vulnerability scanners would be able to automatically update themselves from publicly disclosed vulnerability reports. Currently, such task is typically done by their developers, who read vulnerability reports written in natural language and then update the scanner with the corresponding vulnerability signatures. Given the growth of publicly disclosed vulnerabilities, the requirement to quickly update vulnerability scanners is becoming a crucial cybersecurity problem.

The core of the proposed approach is the automatic extraction of *vulnerability signatures* from CVE descriptions. We presented two ways to extract such signatures. The first one, was based on a heuristic method developed by our (human) experience in reading thousands of CVE descriptions from different software systems. The second one, leveraged state-of-the-art large language models. Specifically, the GPT-3 system developed and published by OpenIA.

Our results show that it is possible to automate the identification of vulnerabilities using pre-trained AI models. To evaluate the ability of the pretrained model to extract important information from a CVE description. A heuristic model is created that is used for comparison. What the results show is that whereas the heuristic approach is fast (0,009 seconds per CVE on average) but inaccurate (accuracy of 0,732), the GPT-based approach is slower (1,188 seconds per CVE on average) but significantly more accurate (accuracy of 1,0) than the heuristic approach. Despite the larger duration per CVE of the GPT-based approach, it is still faster than analyzing CVEs manually. These findings suggest that while using GPT for cybersecurity tasks is feasible and can yield high accuracy, the long execution times may limit its practicality in some contexts. To address this, further work could explore hybrid approaches that combine the accuracy of GPT with the efficiency of heuristic methods, or seek to optimize GPT-based models to reduce processing times without sacrificing performance.

The ability of the pre-trained GPT model to identify vulnerabilities using CVE signatures database was evaluated in the second experiment. Which yields results that show that the GPT model has a considerably higher average accuracy 0,894 than the heuristic implementation 0,503. Therefore, we consider that a heuristic model is not the best alternative for this process. Obtaining 1,0 accuracy in the first experiment with the GPT implementation and 0.918 in the heuristic implementation in the second experiment. We conclude that the proposed architecture is a viable model to automate the process of identifying vulnerabilities listed in MITRE CVE. Also, that the GPT-3.5-turbo-0125 model can effectively identify vulnerable software versions from CVE descriptions, making it a viable tool for cybersecurity applications.

However, the GPT-based VIS (Vulnerability Identification System) also demonstrated a significantly longer execution time compared to the Heuristic VIS, raising concerns about scalability and real-time application. The Heuristic VIS, while not as accurate or reliable, processed data much more quickly, indicating its potential suitability for situations requiring rapid results.

These findings suggest that GPT-based approaches can play a valuable role in cybersecurity, providing high accuracy and adaptability. To overcome the limitations of longer execution times, future research could explore hybrid approaches that combine the efficiency of heuristic.

Beyond the automated extraction of vulnerability signatures described in this work, we envision GPT-based methods that generate source code to assess whether a vulnerability is present in a target system or not. This would greatly improve the scalability problem faced by vulnerability scanners in the light of the unprecedented amount of vulnerability reports that we see today and its growing trend. We plan to explore this research path in future work.

References

1. Agency, C..I.S.: Security tip (st04-001), https://www.cisa.gov/uscert/ncas/tips/ST04-001
2. Balasubramanian, P., Seby, J., Kostakos, P.: Cygent: A cybersecurity conversational agent with log summarization powered by gpt-3 (2024)
3. Bengaluru, K.: CVE and CWE mapping Dataset(2021) — kaggle.com. https://www.kaggle.com/datasets/krooz0/cve-and-cwe-mapping-dataset, [Accessed 01-Sep-2022]
4. Chauhan, A.: A review on various aspects of mongodb databases. Int. J. Eng. Res. Sci. Technol **8**(5), 90–92 (2019)
5. Check Point Research Team: Check point research: Cyber attacks increased 50% year over year. https://blog.checkpoint.com/security/check-point-research-cyber-attacks-increased-50-year-over-year/ (2022), [Online; accessed 22-Jun-2023]
6. Chen, Y., Wang, R., Jiang, H., Shi, S., Xu, R.: Exploring the use of large language models for reference-free text quality evaluation: An empirical study (2023)
7. Cochran, W.G.: Sampling techniques. John Wiley & Sons (1977)
8. Corporation, M.: (Oct 2023), https://www.cve.org/About/Metrics
9. Corporation, T.M.: cve-website — cve.org. https://www.cve.org/Downloads, [Accessed 28-Ago-2022]
10. Database, N.V.: API Vulnerabilities — nvd.nist.gov. https://nvd.nist.gov/developers/vulnerabilities, [Accessed 01-Sep-2022]
11. Dencheva, L.: Comparative analysis of Static application security testing (SAST) and Dynamic application security testing (DAST) by using open-source web application penetration testing tools. Master's thesis, Dublin, National College of Ireland (August 2022), https://norma.ncirl.ie/5956/, submitted
12. Dictionary, M.W.: Definition of HEURISTIC. https://www.merriam-webster.com/dictionary/heuristic, accessed: 2023-11-11
13. Editor, C.C.: Cybersecurity - glossary: Csrc, https://csrc.nist.gov/glossary/term/cybersecurity
14. Editor, C.C.: Vulnerability scanner - glossary: Csrc, https://csrc.nist.gov/glossary/term/vulnerability_scanner
15. Education, B.I.C.: What is natural language processing?, https://www.ibm.com/cloud/learn/natural-language-processing
16. Fang, R., Bindu, R., Gupta, A., Zhan, Q., Kang, D.: Llm agents can autonomously hack websites (2024)
17. Ford, N.: List of data breaches and cyber attacks in 2023 (Nov 2023), https://www.itgovernance.co.uk/blog/list-of-data-breaches-and-cyber-attacks-in-2023#top-data-breach-stats
18. Fu, M., Tantithamthavorn, C., Nguyen, V., Le, T.: Chatgpt for vulnerability detection, classification, and repair: How far are we? (2023)
19. Grand View Research: Software market size, share, growth & trends (2023)
20. IBM: What is natural language processing (nlp)? https://www.ibm.com/topics/natural-language-processing (2021), [Online; accessed 13-Jan-2023]

21. Janiesch, C., Zschech, P., Heinrich, K.: Machine learning and deep learning. Electronic Markets **31**(3), 685–695 (September 2021). https://doi.org/10.1007/s12525-021-00475-, https://ideas.repec.org/a/spr/elmark/v31y2021i3d10.1007_s12525-021-00475-2.html
22. Kanakogi, K., Washizaki, H., Fukazawa, Y., Ogata, S., Okubo, T., Kato, T., Kanuka, H., Hazeyama, A., Yoshioka, N.: Tracing cve vulnerability information to capec attack patterns using natural language processing techniques. Information **12**(8), 298 (2021). https://doi.org/10.3390/info12080298, https://doi.org/10.3390/info12080298
23. KRONSER, A.: CVE (Common Vulnerabilities and Exposures) — kaggle.com. https://www.kaggle.com/datasets/andrewkronser/cve-common-vulnerabilities-and-exposures, [Accessed 28-Ago-2022]
24. Leino, K.R.M.: Developing verified programs with dafny. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. pp. 1488–1490. IEEE Computer Society (2013). https://doi.org/10.1109/ICSE.2013.6606754, https://doi.org/10.1109/ICSE.2013.6606754
25. Li, J.: Vulnerabilities mapping based on OWASP-SANS: A survey for static application security testing (SAST). Annals of Emerging Technologies in Computing **4**(3), 1–8 (jul 2020). https://doi.org/10.33166/aetic.2020.03.001, https://doi.org/10.33166%2Faetic.2020.03.001
26. Li, J., Zhao, B., Zhang, C.: Fuzzing: a survey. Cybersecur. **1**(1), 6 (2018). https://doi.org/10.1186/s42400-018-0002-y, https://doi.org/10.1186/s42400-018-0002-y
27. Lim, R., Wu, M.: https://openai.com/blog/customizing-gpt-3
28. Lindner, A., Guanciale, R., Dam, M.: Proof-producing symbolic execution for binary code verification. CoRR **abs/2304.08848** (2023). https://doi.org/10.48550/arXiv.2304.08848, https://doi.org/10.48550/arXiv.2304.08848
29. Mann, D.E., Christey, S.M.: Towards a common enumeration of vulnerabilities. In: 2nd Workshop on Research with Security Vulnerability Databases, Purdue University, West Lafayette, Indiana (1999)
30. McGraw, G.: Software security: Building security in. Addison-Wesley (2006)
31. Mell, P., Bergeron, T., Henning, D., et al.: Creating a patch and vulnerability management program. NIST Special Publication **800**, 40 (2005)
32. Michael, K., Neter, H.C.N.J., Li, W.: Applied Linear Statistical Models. McGraw-Hill Irwin, Boston (2005)
33. MITRE: Downloads cve, https://www.cve.org/Downloads
34. Mitre: Cve numbering authority (cna) rules. https://www.cve.org/ResourcesSupport/AllResources/CNARules (2020), [Online; accessed 02-Jun-2023]
35. Nayak, K., Marino, D., Efstathopoulos, P., Dumitras, T.: Some vulnerabilities are different than others - studying vulnerabilities and attack surfaces in the wild. In: Stavrou, A., Bos, H., Portokalidis, G. (eds.) Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8688, pp. 426–446. Springer

(2014). https://doi.org/10.1007/978-3-319-11379-1_21, https://doi.org/10.1007/978-3-319-11379-1_21
36. Peters, R.: cron, pp. 81–85. Apress, Berkeley, CA (2009). https://doi.org/10.1007/978-1-4302-1842-5$_1$2, https://doi.org/10.1007/978-1-4302-1842-5_12
37. Project, N.: https://www.nltk.org/api/nltk.corpus.html
38. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I.: Improving language understanding by generative pre-training. Pre-print (2018)
39. Reichert, J.R., Kristensen, K., Mukkamala, R.R., Vatrapu, R.: A supervised machine learning study of online discussion forums about type-2 diabetes. In: A supervised machine learning study of online discussion forums about type-2 diabetes. pp. 1–7 (10 2017). https://doi.org/10.1109/HealthCom.2017.8210815
40. Rudis, B.: Cve 100k: By the numbers. https://www.rapid7.com/blog/post/2018/04/30/cve-100k-by-the-numbers/ (2018), [Online; accessed 01-Jun-2023]
41. Sammut, C., Webb, G.I. (eds.): TF–IDF, pp. 986–987. Springer US, Boston, MA (2010). https://doi.org/10.1007/978-0-387-30164-8$_8$32, https://doi.org/10.1007/978-0-387-30164-8_832
42. Sarica, S., Luo, J.: Stopwords in technical language processing. PLOS ONE **16**(8), e0254937 (aug 2021). https://doi.org/10.1371/journal.pone.0254937, https://doi.org/10.1371/journal.pone.0254937
43. Sharir, O., Peleg, B., Shoham, Y.: The cost of training nlp models: A concise overview (2020)
44. Sheldon, R.: What is ubuntu?: Definition from techtarget (Aug 2023), https://www.techtarget.com/searchdatacenter/definition/Ubuntu
45. Shen, D.: Text Categorization, pp. 3041–3044. Springer US, Boston, MA (2009). https://doi.org/10.1007/978-0-387-39940-9$_4$14, https://doi.org/10.1007/978-0-387-39940-9_414
46. Shirey, R.: Rfc 4949: Internet security glossary, version 2 (2007)
47. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy (2016)
48. The MITRE Corporation: Cve numbering authority (cna) rules. https://www.cve.org/ResourcesSupport/AllResources/CNARules (2020), [Online; accessed 22-Mar-2023]
49. Tøndel, I.A., Cruzes, D.S., Jaatun, M.G., Sindre, G.: Influencing the security prioritisation of an agile software development project. Comput. Secur. **118**, 102744 (2022). https://doi.org/10.1016/j.cose.2022.102744, https://doi.org/10.1016/j.cose.2022.102744
50. Winkler, I., Gomes, A.T.: Chapter 10 - countermeasures. In: Winkler, I., Gomes, A.T. (eds.) Advanced Persistent Security, pp. 105–130. Syngress (2017). https://doi.org/https://doi.org/10.1016/B978-0-12-809316-0.00010-5, https://www.sciencedirect.com/science/article/pii/B9780128093160000105
51. Yosifova, V., Tasheva, A., Trifonov, R.: Predicting vulnerability type in common vulnerabilities and exposures (cve) database with machine learning classifiers. In: 2021 12th National Conference

with International Participation (ELECTRONICA). pp. 1–6 (2021). https://doi.org/10.1109/ELECTRONICA52725.2021.9513723
52. Yu, P., Wu, Y., Peng, J., Zhang, J., Xie, P.: Towards understanding fixes of sonarqube static analysis violations: A large-scale empirical study. In: Zhang, T., Xia, X., Novielli, N. (eds.) IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023. pp. 569–580. IEEE (2023). https://doi.org/10.1109/SANER56733.2023.00059, https://doi.org/10.1109/SANER56733.2023.00059
53. Zhang, T., Kishore, V., Wu, F., Weinberger, K.Q., Artzi, Y.: Bertscore: Evaluating text generation with bert (2020)

A Appendix

A.1 GPT SGM Prompt

"I want you to analyze the following text they correspond to the description of the vulnerability ***<cve-id>****,* ***<cve-description>****, and extract the name of the software or hardware and the versions that are vulnerable to this cve. The idea of this request is to compare this results to a software and check if it is vulnerable or not. Because of that the result of this request must be a json with the following fields,* ***CVE_ID*** *this field is the cve identifier,* ***vulnerable_versions*** *this field is an array of jsons, each json that is part of this array is going to contain the following fields,* ***module*** *this field is the name of the software or hardware,* ***versions*** *this field needs to be a string, and it is going to contain all the vulnerable versions for the software or hardware in question and can only contain the versions numbers and symbols as* $<, >, =<=, >=$ *these symbols are used to represent the versions that are vulnerable and always needs to be a blank space between the comparison symmbol and the version. The result needs to be a json file that contains all the requested information. The answer to this request must be a json format file."*

A.2 GPT VIS Prompt

I want you to act as an evaluator who has the ability to determine if a module (hardware or software product) in a specific version is vulnerable or not to a given CVE.

A CVE is: Common Vulnerabilities and Exposures (CVE) make up a list of computer security flaws that is available to the public. When someone talks about a CVE, they refer to a flaw to which a CVE identification number has been assigned.

To do this, I will provide you with the following information about the module to be evaluated:

- **Module name**: <Module>
- **Module version**: <Version>

You must determine if, with the information received, the module is vulnerable to CVE <CVE-ID>with the description <CVE-DESCRIPTION>.

The response must be in JSON format, following this structure:

Listing 9. JSON Response Format

```
{
    "CVE-ID": "<CVE-ID>",
    "Module": "<Module>",
    "Version": "<Version>",
    "IsVulnerable": <Boolean value, True if vulnerable, False if not>
}
```

*In this format, the values for **"CVE-ID"**, **"Module"**, and **"Version"** must be strings. The **"IsVulnerable"** field should return a boolean value indicating if the module and version are vulnerable to the given CVE.*

Glossary

Artificial Inteligence Artificial intelligence is the simulation of human intelligence processes by machines, especially computer systems.. 4, 11, 16, 21

CISA The Cybersecurity and Infrastructure Security Agency is a component of the United States Department of Homeland Security (DHS) responsible for cybersecurity and infrastructure protection across all levels of government, coordinating cybersecurity programs with U.S. states, and improving the government's cybersecurity protections against private and nation-state hackers.[. 16

Cron The system scheduler on UNIX and Linux systems is called cron. Its purpose is to run commands, series of commands, or scripts on a predetermined schedule. Normally these tasks are performed on systems that run 24 hours a day, 7 days a week. Writing cron scripts to perform system maintenance, backups, monitors, or any other job that you would want to run on a schedule is a very common task [36].. 29

CVE Numbering Authority Vulnerabilities are cataloged and listed as CVE records by the organizations cataloged as CVE Numbering Authority (CNA), which means that it is an entity authorized to enter vulnerabilities into the list.. 12

heuristic According to the Merriam-Webster dictionary, heuristic as an adjective means, "[i]nvolving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods." [12].. 27, 28

IoT The Internet of Things (IoT) describes the network of physical objects—"things"—that are embedded with sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and systems over the internet.. 9, 12

MITRE The MITRE Corporation (stylized as The MITRE Corporation and MITRE) is an American not-for-profit organization. It manages federally funded research and development centers (FFRDCs) supporting various U.S. government agencies in the aviation, defense, healthcare, homeland security, and cybersecurity fields, among others... 9, 10, 12, 19, 20, 23, 24, 29

OpenAI OpenAI is an American artificial intelligence (AI) organization consisting of the non-profit OpenAI, Inc.. 33, 34

Ubuntu Ubuntu is a free, open source operating system (OS) based on Debian Linux. It was first released in 2004 when Mark Shuttleworth and a small team of Debian developers founded Canonical and then launched the Ubuntu project [44].. 42

Vulnerability Identification The vulnerability identification process enables you to identify and understand weaknesses in your system, underlying infrastructure, support systems, and major applications. . 4, 10, 11, 23, 26