



Degree Project in Computer Science

Second cycle, 30 credits

Retrieval-Augmented Generation for Vulnerability Classification

Mapping GitHub Issues to CWE

EDVIN NORDQVIST

Retrieval-Augmented Generation for Vulnerability Classification

Mapping GitHub Issues to CWE

EDVIN NORDQVIST

Degree Programme in Computer Science and Engineering

Date: Tuesday 9th December, 2025

Supervisors: Changjie Wang, Simone Ferlin

Examiner: Marco Chiesa

School of Electrical Engineering and Computer Science

Host company: Red Hat

Swedish title: Retrieval-Augmented Generation för Sårbarhetsklassificering

Swedish subtitle: Koppling av GitHub Issues till CWE

Abstract

As technology advances, pivotal services and infrastructure become increasingly dependent on various software, as well as network and computer systems. With the apparent benefits of these modern solutions come the looming threat of exploitation, requiring timely and accurate vulnerability analysis and correction. To this end, Cipollone leverages transformer-based models to "automate the identification of software vulnerabilities through the analysis of GitHub issues". Though Cipollone's pipeline was successful in providing accurate classification alongside detailed vulnerability descriptions, it lacked any form of formal vulnerability categorization, which is often present in a proper vulnerability analysis. This thesis aims to extend Cipollone's pipeline to include automated vulnerability categorization using the widely recognized list of common software and hardware weaknesses, the Common Weakness Enumeration (CWE). There have been various prior attempts at general automation of CWE labeling. However, they all suffer from the abundance of available labels, contrasted with the scarcity of real-world examples demonstrating their use. This leads to insufficient training or fine-tuning data, limiting achievable precision or class coverage in the final labeling. To circumvent this, we leverage the already available, comprehensive CWE documentation using Retrieval-Augmented Generation without relying on training or fine-tuning. The precision and computational efficiency of this approach demonstrate the viability of knowledge retrieval in supporting and automating vulnerability categorization. In addition to completing the loop of Cipollone's pipeline, this serves to significantly narrow the window of exploitation between vulnerability discovery and correction. Thus, this research provides an efficient, scalable, and powerful framework for ensuring the security of software and computer systems. Finally, the reduced reliance on training or fine-tuning in automated CWE labeling broadens avenues for future work and solutions in the field.

Keywords

Vulnerability classification, Vulnerability detection, Machine learning, Large language models (LLM), Retrieval-augmented generation (RAG), Information retrieval, GitHub issues, Common weakness enumeration (CWE)

Sammanfattning

I takt med teknologisk utveckling blir centrala tjänster och infrastruktur alltmer beroende av olika programvaror samt nätverks- och datorsystem. I samband med de uppenbara fördelarna av dessa moderna lösningar följer det överhängande hotet av exploatering vilket kräver snabb och noggrann sårbarhetsanalys och korrigerande åtgärder. För detta ändamål utnyttjar Cipollone transformatorbaserade modeller för att ”automatisera identifieringen av programvarusårbarheter genom analys av GitHub-problem” (min översättning). Medan Cipollones lösning lyckades ge träffsäker klassificering tillsammans med detaljerade sårbarhetsbeskrivningar saknade den formel sårbarhetskategorisering som ofta finns i en ordentlig sårbarhetsanalys. Denna avhandling syftar till att utvidga Cipollones lösning för att inkludera automatiserad sårbarhetskategorisering med användning av den allmänt erkända Common Weakness Enumeration (CWE). Diverse tidigare försök av generell automatisering av CWE-klassificering har genomförts. Dock, har samtliga en mångfald av tillgängliga klasser men lider brist på verkliga exempel av deras användning. Detta leder till otillräcklig tränings- eller finjusteringsdata vilket begränsar den uppnåeliga precisionen eller klasstäckning av den slutliga klassificeringen. För att undvika detta utnyttjar vi den tillgängliga, omfattande CWE-dokumentationen med hjälp av Retrieval-Augmented Generation utan att förlita oss på träning eller finjustering av modellen. Precisionen och beräkningseffektiviteten av våra resultat visar på användbarheten av informationssökning i att stötta och automatisera sårbarhetsklassificering. Utöver att fullständiggöra Cipollones lösning leder detta till att i betydande omfattning minska avståndet mellan upptäckten av sårbarheter och deras korrigerande åtgärder. följaktligen, tillhandahåller ett effektivt, skalbart och kraftfullt ramverk för att säkerställa säkerheten av programvara och datorsystem. Slutligen vidgar det minskade beroendet av träning och finjustering möjligheterna för framtida arbete och lösningar inom automatiserad CWE-klassificering.

Nyckelord

Sårbarhetsklassificering, Sårbarhetsdetektering, Maskininlärning, Stora språkmodeller (LLM), Retrieval-augmented generation (RAG), Informationssökning, GitHub Issues, Common Weakness Enumeration (CWE)

Acknowledgments

I would like to express my heartfelt gratitude to Simone Ferlin, my supervisor at Red Hat, for providing me with the opportunity to work on this thesis and supporting me through to its completion. Similarly, I am also deeply thankful to Marco Chiesa and my supervisor at KTH, Changjie Wang, from whom I've learned a great deal and who supported me throughout this journey. I wish to extend my thanks to Daniele Cipollone, not only for his invaluable research that made this work possible, but also for personally helping me in brainstorming the goals and ideas for this thesis and always being available for support. Furthermore, I would like to thank my great friend Georgios Vidalakis and my mom Annika Nordqvist for putting up with my constant ranting about this project and lending me their invaluable perspectives, insights, and suggestions—helping me stay sane along the way.

Finally, I am thankful to Red Hat for hosting me, making this thesis possible, and providing me with the chance to embark on this remarkable journey.

Stockholm, December 2025
Edvin Nordqvist

Contents

1	Introduction	1
1.1	Background	2
1.2	Problem	3
1.3	Purpose	4
1.3.1	Ethics and Sustainability	4
1.4	Goals	4
1.5	Research Methodology	5
1.6	Delimitations	6
1.7	Structure of the thesis	6
2	Background	7
2.1	Vulnerabilities	7
2.1.1	Vulnerability lifecycle	8
2.1.2	Common Vulnerabilities and Exposures (CVE)	9
2.1.2.1	The CVE Record	10
2.1.2.2	The CVE Lifecycle	10
2.1.3	The National Vulnerability Database (NVD)	10
2.1.3.1	Enrichment Process	11
2.1.3.2	CNAs and Acceptance Levels	11
2.2	The Common Weakness Enumeration (CWE)	12
2.2.1	CWE Relationships	12
2.2.1.1	Special Relationships	13
2.2.2	CWE Vulnerability Mappings and Content	14
2.2.3	CWE Views	14

2.3	Natural Language Processing (NLP)	14
2.3.1	Language Modeling (LM)	15
2.3.2	Recurrent Neural Networks (RNNs)	16
2.3.3	Attention	16
2.3.4	Transformers	18
2.3.5	Pre-trained Language Models (PLMs)	19
2.3.5.1	BERT	20
2.3.6	Large Language Models (LLMs)	20
2.3.7	Retrieval Augmented Generation (RAG)	21
2.3.7.1	Retrieval	21
2.3.7.2	Generation	22
2.3.7.3	Augmentation	22
2.3.7.4	Training	23
2.4	Related work	23
2.4.1	Cipollone and Transformer Based Vulnerability Detection	23
2.4.1.1	Approach	23
2.4.1.2	Data collection	24
2.4.1.3	Why GitHub?	24
2.4.1.4	Results and Conclusion	25
2.4.2	Automated CWE Categorization	25
3	Methodology	29
3.1	Research Process	29
3.2	Data Collection	30
3.2.1	CVE and GitHub Pipeline	30
3.2.2	CWE collection and filtering	30
3.3	Experimental design	31
3.3.1	RAG Methodologies	32
3.3.1.1	Knowledge Base Creation and Optimization	32
3.3.1.2	Prompting	34
3.3.1.3	RAG pipeline	38

3.3.1.4	Fusion Retrieval	38
3.3.1.5	RAPTOR	39
3.3.2	Baseline Experiment	40
3.4	Evaluation framework	41
3.4.1	Evaluation Data	42
3.4.2	Performance Metrics	44
4	Results and Analysis	47
4.1	Baseline	47
4.2	Retrieval	47
4.2.1	RAPTOR and Fusion	51
4.3	Major results	51
5	Discussion	55
5.1	Limitations	55
5.2	Reflections	56
5.3	Future Work	57
5.3.1	RAG Optimization	57
5.3.2	Prompt Engineering	57
5.3.3	Training and Fine-tuning	58
5.3.4	Evaluation	58
6	Conclusions	59
	References	61
A	Supporting materials	67

List of Figures

2.1	Abstraction tree example visualizing Common Weakness Enumeration (CWE) parent and child relationships [28].	13
2.2	Basic RNN architecture visualized (reproduced from Figure 1 in [47]). . .	17
2.3	Transformer architecture (reused from Figure 1 in [49]).	19
2.4	Cipollone’s proposed pipeline (reused from Figure 3.4 in [4]).	24
3.1	Research Process.	30
3.2	Final automated vulnerability categorization pipeline.	32
3.3	Automated vulnerability categorization pipeline used for Retrieval-Augmented Generation (RAG) experimentation.	33
3.4	Distribution of the top CWEs for each true positive sample.	43
A.1	Forest-like graphs of CWE views 1003 and 1003.	68

List of Tables

2.1	Top frequent domains used in references of the CVEs scraped by Cipollone (reused from Table 3.2 in [4])	25
3.1	Breakdown of the number of CWEs in different abstraction levels and statistics on the children of those CWEs for CWE view 1000 and 1003. . .	40
3.2	Breakdown of the structure and important aspects of our test data samples.	43
4.1	Baseline results on view 1003 and view 1000 on the "small" true positive data sample.	48
4.4	Results of the first, basic retrieval experiment.	49
4.7	Results of the second retrieval experiment.	50
4.10	Results of the third retrieval experiment.	51
4.13	Results of the RAG pipeline experiment.	52
4.16	Results of the complete combined pipeline experiment.	53

List of acronyms and abbreviations

AGI	Artificial General Intelligence
AI	Artificial Intelligence
BERT	Bidirectional Encoder Representations from Transformers
CNA	CVE Numbering Authority
CPE	Common Platform Enumeration
CTI	Cyber Threat Intelligence
CVE	Common Vulnerabilities and Exposures
CVMAP	Collaborative Vulnerability Metadata Acceptance Process
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DISA	U.S. Defense Information Systems Agency
DL	Deep Learning
ICL	In Context Learning
ITU-T	International Telecommunication Union's
LLM	Large Language Model
LM	Language Modeling
ML	Machine Learning
MLM	Masked Language Model
NIST	National Institute of Standards and Technology
NLM	Neural Language Model
NLP	Natural Language Processing
NVD	National Vulnerability Database
OS	Operating System
OWASP	Open Worldwide Application Security Project
PLM	Pre-trained language model
RA-LLM	Retrieval-Augmented Large Language Models
RAG	Retrieval-Augmented Generation
RAPTOR	Recursive Abstraction Processing For Tree-Organized Retrieval
RNN	Recurrent Neural Network
SLM	Statistical Language Model
SOTA	State Of The Art

Chapter 1

Introduction

The proliferation of the computer has resulted in a societal dependency on digital systems. Everything from power grids to banking systems, the stock exchange to the public square, is dependent on servers, software, and various computer infrastructures. As such, the exploitation of these systems and the software that maintains them poses an ever-greater risk of causing devastating damage on a global scale. With each update of a particular software or application, the risk of introducing vulnerabilities enabling this exploitation increases. This results in a continuous cycle where software development introduces vulnerabilities that, upon discovery and disclosure, require correction before further development [1]. Of particular concern in this vulnerability life cycle are unknown and/or unaddressed vulnerabilities, so-called "zero-day exploits" [2]. These have the potential of exploitation before patches are made publicly available, and with the gradual increase in zero-day exploitation, [3], narrowing the gap between the birth, discovery, and disclosure of vulnerabilities is vital.

To that end, Daniele Cipollone explored the potential of transformer-based language models for vulnerability detection within GitHub repositories through analysis of their GitHub issues [4]. GitHub is a hugely popular cloud-based version control platform that simplifies collaborative software development for tracking changes, issues, and features [5]. GitHub issues track bug reports, new features, ideas, and general discussions [6]. As such, GitHub issues provide relatively concise documentation, facilitating vulnerability auditing. Cipollone developed a pipeline consisting of an embedding model, XGBoost classifier, and OpenAI-based **Large Language Model (LLM)** classifier to classify GitHub issues indicative of a vulnerability in the associated repository. Although suffering from a high false positive rate, the model achieved an accuracy of 90%, demonstrating its potential in vulnerability detection [4].

Although Cipollone's model provides a brief description of the vulnerability at the end of his pipeline, he only employs true-false labeling, opting not to categorize the detected vulnerabilities [4]. However, the **Common Vulnerabilities and Exposures (CVE)** in the **National Vulnerability Database (NVD)** that Cipollone pulls vulnerability data from [4] contains **Common Weakness Enumeration (CWE)** information, which constitutes proper vulnerability labeling [7]. The **CWE** as of version 4.16 consists of 940 Weaknesses

[8]. These are common software and hardware weakness types that could have security ramifications [9]. The documentation for each weakness is extensive and can contain everything from extended descriptions to demonstrative and observed examples [10]. This naturally begs the question: can we somehow leverage the extensive **CWE** documentation to categorize the detected vulnerabilities at the end of Cipollone's pipeline?

Retrieval-Augmented Generation (RAG) models are **LLMs** that leverage dense vector indexes of embedded documents in combination with a pre-trained neural retriever to enhance specialized knowledge. Upon a query, the retriever will fetch the top-K most relevant documents (where K is the number of retrieved documents), which, in turn, provide the **LLM** additional context when generating a response [11]. This work explores the potential of **RAG** on Cipollone's pipeline to close the loop and provide proper vulnerability categorization. This will consecutively facilitate understanding, not only narrowing the gap between birth, discovery, and disclosure in the vulnerability life cycle but also "correction". Leveraging retrieval to enhance the contextual and specialized knowledge of **LLMs**, this study aims to determine whether it's feasible to categorize vulnerabilities in a cost-effective and scalable manner. The findings of this study demonstrate not only the viability of the approaches presented but also highlight the potential of **RAG** in facilitating efficient, low-cost vulnerability identification. Thus, expediting the vulnerability life cycle and reducing the risk of exploitation.

There have been various studies investigating the automation of labeling **CVEs** in the **NVD** with missing **CWE** labels using various **Machine Learning (ML)** and **LLM** strategies [12, 13, 14, 15]. As well as automated **CWE** labeling given vulnerability descriptions [16]. However, none have researched the potential of **RAG** for computationally efficient vulnerability labeling or focused on labeling in GitHub issues.

1.1 Background

LLMs are advanced language models [17] that leverage large model and/or data sizes [18] to enable generalization on unseen tasks [19], exhibiting potential for **Artificial General Intelligence (AGI)** [20]. **Language Modeling (LM)** aims to model the likelihood of generated word sequences given previous context to predict future tokens. In other words, **LM** predicts the next output token based on the previous generation. Scaling **LM** enables generative [21] deep learning models (**LLMs**) [19] that excel at a wide variety of **Natural Language Processing (NLP)** tasks. These tasks include, but are not limited to, text generation; language understanding; arithmetic, logical, and temporal reasoning; machine translation and question answering [17]. Indeed, **LLMs** have previously been successfully adopted in research on automated **CWE** classification [22, 14, 13].

Despite the impressive capabilities of **LLMs**, they have historically struggled with factual errors on knowledge-intensive tasks due to their sole reliance on parametric memory, which **RAG** can help alleviate. **RAG** augments the **LLM** query before generation to include pertinent retrieved information [23]. To achieve this, **RAG** utilizes a vector index of specialized vectorized documents alongside some pre-trained retriever [11]. The retriever encodes the query, applies some search algorithm such as vector embedding similarity

or BM25 [24], and fetches the top-K most relevant documents [11]. This adds important context, thus preventing hallucinations and factual errors [23].

The **NVD**, product of **National Institute of Standards and Technology (NIST)**, is a "U.S. government repository of standards-based vulnerability management data" [7]. The **NVD** enriches **CVEs** from the **CVE** list by adding association impact metrics **Common Vulnerability Scoring System (CVSS)**, applicability statements **Common Platform Enumeration (CPE)**, and, most pertinent to this study, Vulnerability types **CWE** [7]. The **CVE** Record, product of the MITRE corporation, is a catalog of publicly disclosed vulnerabilities. The vulnerabilities are discovered and disclosed to the **CVE** via various organizations around the world in an effort to coordinate, prioritize, address, and communicate consistent descriptions of vulnerabilities [25]. The **CVE** list enriched by the **NVD** contains useful data such as **CVE-ID** descriptions, references of various possible sources including GitHub and GitHub issues, related **CWEs**, etc. [26]. Finally, the **CWE**, as explained in the introduction, consists of a list of common weaknesses that could have security ramifications along with extensive documentation. The primary difference between the **CVE** and **CWE** is that the **CVE** documents individual instances of vulnerabilities detected in various software, whilst the **CWE** is a list of more general vulnerability classes/types that can be used for classification.

We can divide the vulnerability life cycle into 6 phases: discovery, disclosure, correction, publicity, scripting, death [1]. This life cycle can then be categorized into 3 different risk exposure periods: black, grey, and white risk. Black risk encompasses the time between the discovery and disclosure of a vulnerability, when it is most sensitive to exploitation. Grey risk refers to the period between disclosure and correction, when appropriate stakeholders are notified. The period after correction is referred to as "white risk", where the vulnerability poses a minimal threat. The entire time between discovery and patch time (correction [1]), in other words, the period of black and gray risk exposure, a vulnerable software is at risk [27].

1.2 Problem

The issue of categorizing a vulnerability is a complicated one. Mapping **CWE** identifiers to newly discovered vulnerabilities requires meticulous analysis and identification of root causes. Searching the **CWE** for appropriate weaknesses, understanding the **CWE** structure, terminology, and relationships. Correlating **CVE** Records, bug or vulnerability tickets with **CWE** entries, etc. [28]. The overwhelming number of sources of vulnerability information, such as various mailing lists, blogs, and GitHub issues, makes such analysis a time-consuming and difficult problem for cybersecurity experts.

In the **NVD**, the task of analyzing and labeling vulnerabilities is afforded to vendors, third-party security researchers, and vulnerability coordinators [7]. Leaving such analysis completely up to humans and various expert sources introduces high operational time and cost in addition to the possibility of human error and contradiction, as evident when analyzing the **NVD**.

Given these various challenges and **LLMs** proficiency at language understanding,

reasoning, and contextual comprehension [17], one might pose the question: Is it possible to leverage the extensive external knowledge available to us with **LLMs** to automate and/or substantially alleviate this analysis process?

1.3 Purpose

With the increase in digitization and the potential impact and risks vulnerabilities pose, limiting the time of Black and Gray risk exposure periods is critical. Crucial steps in the process of discovery, disclosure, and eventual correction of a vulnerability are detection and categorization. Categorization, in particular, facilitates further analysis and vulnerability patching.

While Cipollone also investigated the possibility of utilizing **LLMs** for automating the analysis of cybersecurity-related information, his work primarily focused on detection and the black risk phase. Specifically, he focused on true-false vulnerability labelling through GitHub issue analysis accompanied by an informal vulnerability description [4]. This work aims to extend Cipollone's pipeline to include formal categorization, in an effort to further reduce both the black risk and gray risk exposure periods. As part of this research, the potential of **LLMs** for general vulnerability labeling is also investigated.

1.3.1 Ethics and Sustainability

Early detection and categorization of vulnerabilities before traditional disclosure dates significantly mitigates the risk of cybersecurity-related attacks and exploitation. Optimized vulnerability information analysis through automated vulnerability categorization enhances overall cybersecurity by mitigating risk exposure periods. Furthermore, this work aims to minimize the cost of such automation utilizing **RAG** alongside smaller, more cost-effective models. **RAG** specifically helps minimize computational cost through efficient retrieval, unlike traditional, more costly methods such as fine-tuning or training. Through this approach, this thesis aims to promote sustainable and environmentally conscientious research practices and technological advancements.

1.4 Goals

The overall goal of this thesis is to investigate the potential of using **LLMs** alongside external knowledge retrieval for vulnerability classification. Specifically, by extending Cipollone's pipeline, which analyzes GitHub issues and is detailed more thoroughly in chapter 2. From this, a series of sub-goals can be established:

- Adjust Cipollone's model and prompts as well as data collection to work well with **RAG** and different retrieval strategies.
- Develop an extension of Cipollone's pipeline that makes use of the external knowledge from the **CWE** and the descriptions the pipeline provides to enable accurate formal categorization.

- Compare various retrieval strategies as well as a brute force baseline approach, detailed in chapter 3, using standard performance metrics.

To address these goals, a series of iterative experiments was conducted and compared. The results were promising, demonstrating the potential of using **LLMs** with **RAG** to support vulnerability categorization and analysis. Furthermore, the findings of this thesis highlight significant opportunities for future research and practical application.

1.5 Research Methodology

This thesis employs an experimental methodology, utilizing a quantitative approach due to the inherent empirical nature of the problem.

The study begins with a thorough evaluation of the available literature on relevant topics. These include, related work, **LLMs**, **RAG**, vulnerability research, the **NVD**, **CVE** and **CWE**. This grounds the approach and methodology in established knowledge, providing a clear picture of the **State Of The Art (SOTA)**. In this way, gaps in available research are identified to ensure the novelty of the vulnerability labeling problems addressed. Particular attention is allocated to research on the **CWE** and **RAG**, enabling the proper usage of necessary tools and technologies.

Following the literature study, the **CWE** gets scraped and thoroughly analyzed to not only establish a vector store for retrieval but also to gain familiarity with the database. Additionally, Cipollone's data collection and vulnerability detection pipeline is re-implemented. To enable correlation between particular GitHub issues and established vulnerabilities, Cipollone utilizes information from the **NVD**. Similarly, **CWE** data correlated to each issue is aggregated via a slight refactoring of Cipollone's data collection pipeline.

The **NVD** data is then thoroughly filtered to ensure an accurate training and testing dataset. The filtering is based on the different aspects of the **CWE** and GitHub repositories that are identified.

Finally, Cipollone's detection pipeline is expanded upon and adjusted to incorporate classification. **LLMs** are chosen for their advanced Language understanding, reasoning capabilities, and contextual comprehension. Furthermore, **RAG** is incorporated to mitigate the tendency of **LLMs** to hallucinate in relation to highly specialized knowledge. **RAG** is also fairly computationally cheap compared to fine-tuning and dynamically adjusts to new information by simply expanding the underlying knowledge base. This flexibility is especially useful when considering the **CWE**, which receives regular updates.

In summary, the methodology incorporates a thorough literature study, careful knowledge base creation, meticulous data collection and filtering, and advanced refactoring and expansion of Cipollone's pipeline to incorporate vulnerability categorization. The detailed outlining of this methodology is described in chapter 3.

1.6 Delimitations

Following Cipollone's work, this thesis limited itself explicitly to GitHub issues as a source of data for vulnerability categorization. Additionally, although there are many vulnerability-related databases, this work limited itself to the **NVD** and **CWE**.

Only **RAG**-like strategies were explored so as not to get overwhelmed by the numerous ways of achieving automated categorization. Additionally, due to time and resource limitations, it was not feasible to explore all possible **RAG** strategies. There are numerous optimizations available for different use cases, all of which couldn't possibly be included within the boundaries of this thesis. Furthermore, this thesis is not concerned with an exhaustive exploration of various **RAG** strategies. Instead, this thesis aims to research the potential of employing **RAG** for vulnerability categorization, leaving the task of optimization for future work. Finally, following Cipollone's work, this thesis exclusively utilizes OpenAI models and established tools such as LangChain.

As this was an extension of Cipollone's thesis. Limitations attributed to Cipollone's pipeline that are not addressable by the proposed extension in this thesis have to be considered, as this work is not interested in reinventing Cipollone's pipeline.

1.7 Structure of the thesis

Chapter 2 presents detailed relevant background information about the vulnerability lifecycle, **LM** and **LLMs**, **RAG**, the **NVD** and **CWE**. Additionally, the background contains a summary of Cipollone's research along with a comprehensive survey of previous research and solutions in automated vulnerability categorization. Chapter 3 presents the methodology and methodology used to solve the problem, including experimental design, data collection, and filtering, in addition to evaluation frameworks and metrics. Chapter 4 covers the results and their implications and interpretations. Chapter 5 more deeply discusses and reflects on the results from a more analytical perspective and in terms of their practical implications. Limitations in our research and opportunities for future work are also thoroughly discussed. Finally, we conclude the paper in chapter 6.

Chapter 2

Background

This chapter aims to provide a thorough background on vulnerabilities; various vulnerability databases like the **CWE**; **NLP** techniques, including **LM**, **LLMs**, and **RAG**; and finally, Cipollone's previous work that this thesis builds on, along with various related works in automated **CWE** categorization. This is intended to clarify the focus of this work and provide sufficient background in relevant and necessary fields.

2.1 Vulnerabilities

Vulnerabilities in computer software and security can be defined as "any weakness or flaw existing in a system. The susceptibility of a system to a specific threat attack or harmful event, or the opportunity available to a threat agent to mount that attack." [29] Or more specifically, "Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source" according to **NIST** [30]. For a definition of the term "weakness", see subsection 2.2.2.

Vulnerabilities may accidentally be introduced at any point of the software development process via errors, design flaws, and all-around inadequate security procedures. They may affect any component of an automated system, not limited to hardware and software, but also organizations, procedures, personnel, and management and administration. They often involve weaknesses facilitating unauthorized access to information, disruption of critical processing, or denial of service [29]. Potential impacts may range from small inconveniences to the compromising of various critical systems on a global scale, costing billions of dollars. As such, a thorough understanding and analysis of various common vulnerabilities and weaknesses is fundamental for proper modern cybersecurity procedures to alleviate severity and mitigate potential damages. These procedures typically concern various types of stakeholders, such as vulnerability repositories, researchers, developers, and vendors. The procedures may include penetration testing, static and dynamic analysis, proper vulnerability resource/information monitoring, and various automated discovery and investigation tools.

2.1.1 Vulnerability lifecycle

The vulnerability life cycle describes the states a vulnerability can inhabit during its lifetime. This is not always a linear process, as the progression may vary depending on the interactions between the exploit, software, host system, etc. Despite this, general models have been proposed for this lifecycle. Arbaugh *et al.* [1] introduced the concept of a vulnerability lifecycle, capturing all possible vulnerability states in a granular fashion. They defined the general order of appearance of states as follows:

- *Birth*: Denoting the birth of the vulnerability. Often introduced unintentionally in development. Specifically, the vulnerability is considered to exist when there is a nonzero probability of exploitation. In other words, internal testing and development do not count.
- *Discovery*: The original discovery of the vulnerability is often not an event that can be known, as it's not always disclosed. Regardless, when either a malicious or benevolent actor discovers a security flaw in a product, it turns into a discovered vulnerability
- *Disclosure*: The discoverer details the vulnerability and its security implications to a wider audience. The vulnerability may be reported to developers directly, but more generally, disclosure is conducted through mailing lists, forums, or particularly the [CVE-List](#) as described in subsection 2.1.2.1.
- *Correction*: A software module is released that corrects the underlying flaw responsible for the vulnerability
- *Publicity*: The vulnerability is made public to a larger general audience through various forms of media, such as news articles.
- *Scripting*: The exploitation is scripted and becomes standardized, enabling malicious actors of little or no skill to exploit the vulnerability. This may also include cookbooks and anything that significantly facilitates exploitation, dramatically increasing the number of potential exploiters.
- *Death*: The number of systems that are under threat of exploitation shrinks to insignificance. This may occur if old vulnerable patches cease to be available, all systems at risk are patched, attackers completely lose interest, etc.

Frei *et al.* [27], developing on Arbaugh *et al.* model, introducing a more holistic view, separating the lifecycle into the following phases. [27]

- *Discovery*: In line with Arbaugh *et al.*, discovery is the earliest reported date of the discovery of the underlying flaw resulting in the vulnerability. Vulnerabilities may exist before discovery, but do not pose any risk at that stage.
- *Exploit*: The earliest date of the availability of an exploit.
- *Disclosure*: In comparison to Arbaugh *et al.* a more strict definition is used. Strictly,

The first date a vulnerability is described on a channel where the disclosed information on the vulnerability

1. is freely available to the public
 2. is published by a trusted and independent channel
 3. has undergone analysis by experts such that risk rating information is included.
- *Patch*: Equivalent to *correction* in Arbaugh *et al.* model. May include a patch, workaround, or fix.

In association with these updated phases, Frei *et al.* detailed three major *risk exposure periods* defining the severity of the vulnerability relative to its position in the vulnerability lifecycle. [27]

- *Black Risk*: This period encompasses the period between discovery and disclosure, where only a closed group is aware of the vulnerability. This group could be hackers or vulnerability specialists working on a fix, but regardless, there is knowledge of certain security risks. However, the public and important stakeholders may have no access to this knowledge, making this an extremely volatile period in the vulnerabilities lifecycle as there is a high risk of *zero-day exploits* with no known patches or potential safety procedures to mitigate damages.
- *Grey Risk*: This period encompasses the period between disclosure and patch availability. Zero-day exploits are still of great concern, but unlike the black risk exposure period, stakeholders such as vendors and customers have access to important knowledge about the vulnerability that may be used to assess individual risk and mitigate potential damages.
- *White Risk*: The period following the patch availability and installation of said patch by users, where the vulnerability poses little to no threat. This is often followed by the eventual death of the vulnerability previously introduced by Arbaugh *et al.*

Proper monitoring of various **Cyber Threat Intelligence (CTI)** resources, such as the **NVD** and associated **CVE-List**, comprises a critical part of proper proactive vulnerability mitigation. Such methods of knowledge-sharing between cybersecurity stakeholders are crucial in narrowing the gap between discovery, disclosure, and patch availability and, as a result, limiting the associated risk exposure periods [31]. The more efficient the vulnerability analysis, the quicker the publication onto these databases and thus the sooner mitigation of said vulnerability will be possible. With this in mind, automated tools alleviating vulnerability analysis are highly sought after.

2.1.2 Common Vulnerabilities and Exposures (CVE)

The **CVE** List is a publicly disclosed catalog of cybersecurity vulnerabilities in cooperation with various organizations around the world referred to, by the **CVE** as **CVE Numbering Authorities (CNAs)** [25]. Launched in 1999 by the MITRE corporation, the **CVE** List has become widely adopted and included in alerts by numerous **Operating Systems (OSes)** vendors and organizations in an effort to facilitate global transparency and cooperation. Equally, **CVE** Records are used as unique identifiers for vulnerabilities in public watch lists such as the **Open Worldwide Application Security Project (OWASP)**. [32]

The **CVE** underwent bureaucratic adoption as early as 2004 when **U.S. Defense Information Systems Agency (DISA)** issued a task order for information assurance applications that required the use of products that use **CVE** IDs. Ultimately, the intergovernmental 150-year-old treaty-based organization, **International Telecommunication Union's (ITU-T)** adopted the **CVE** as part of their **CVE** Compatibility Program in 2011. [32]

2.1.2.1 The CVE Record

/ The **CVE** List is a list comprised of **CVE** Records. A **CVE** record is a machine-readable entry describing the vulnerability. Each **CVE** Record must at a minimum contain a **CVE** ID of four or more digits; a brief description of the vulnerability; products and versions the vulnerability affects, and at least one relevant reference [33]. Additionally, a **CVE** Record can inhabit one of the following states [33]:

1. **Reserved:** The record **CVE** ID has been reserved by **CNA**.
2. **Published:** The record has been populated with the necessary data and published under its **CVE** ID by a **CNA**
3. **Rejected:** The record associated with the **CVE** ID should no longer be used but remains on the **CVE** List for documentation.

2.1.2.2 The CVE Lifecycle

The process of a vulnerability getting published on the **CVE**-List can be separated into the six following steps [33]:

1. **Discover:** A vulnerability is discovered.
2. **Report:** The discovered vulnerability is reported to a **CNA**.
3. **Request:** The **CNA** assigns a **CVE** identifier (**CVE** ID).
4. **Reserve:** The ID is reserved, meaning the ID is being used for early-stage vulnerability coordination, but the vulnerability is not yet ready to be publicly disclosed.
5. **Submit:** **CNA** submits details including but not limited to affected product(s); affected or fixed product versions; vulnerability type, root cause, or impact; and at least one public reference.
6. **Publish:** The **CNA** publishes the **CVE** Record to the **CVE** List once all the mandatory elements described in section 2.1.2.1 have been included.

2.1.3 The National Vulnerability Database (NVD)

The **NVD** , formally known as the Categorization of Attacks Toolkit or ICAT, began in 1999 as an access database of attack scripts. It's a vulnerability database operated by **NIST** and built on the **CVE** List [7]. renamed to the **NVD** in 2005 [34] the project aims to enrich the **CVE** with vulnerability types (**CWE**), applicability statements (**CPE**), associated impact metrics (**CVSS**) and other relevant metadata [7].

2.1.3.1 Enrichment Process

Once a **CVE** Record has been assigned a **CVE** ID and published, the vulnerability will undergo analysis by the **NVD**. The enrichment process consists of the following two stages [35]:

1. **Analysis:** Based on provided data such as description and provided external references, a **NVD** enrichment team member associates appropriate **CWE**(s) and remaining metadata (**CVSS**, **CPE**, etc.)
2. **Verification:** A second and typically more experienced enrichment team member reviews and verifies the enriched metadata. Once this is completed, the enriched **CVE** metadata is publicly published, ensuring the proper standards and procedures have been applied.

2.1.3.2 CNAs and Acceptance Levels

Organizations that are authorized to assign CVE IDs are known as **CNAs**. Generally, **CNA** consists of seasoned and established vendors and organizations with experience in cybersecurity. They may be sought after for consulting by research vendors or well-seasoned bug bounty hunters. **CNAs** have a well-defined scope for which vulnerability types they're assigned to vet and are subject to rules in an effort to maintain consistency. [36]

If the **CNA** provides enrichment data in their **CVE** submission, **NVD** information obtained through the enrichment process detailed in section 2.1.3.1 and **CNA** information is compared to determine alignment. This is called the **Collaborative Vulnerability Metadata Acceptance Process (CVMAP)**, which is a program that determines each **CNA** acceptance level based on submission alignments with the **NVD** enrichment team [37]. The acceptance level of a **CNA** determines the credibility of the **CNA** and is independent across different enrichment categories (**CVSS**, **CPE**, etc.)

There are four separate possible acceptance levels [38]:

1. *Reference:* The **CNA** is participant in **CVMAP** and data provided is displayed on the **NVD** website.
2. *Contributor:* Everything for acceptance level *Reference* applies and the **CNA** is an active participant in **CVMAP**. Additionally, the **CNA**'s submissions have aligned enough with **CVMAP** practices, granting them the next acceptance level.
3. *Provider:* Everything for acceptance level *Contributor* applies and the **CNA**'s submissions have aligned enough with **CVMAP** practices granting them the highest acceptance level. At this acceptance level, the **CNA**'s data is of equal value to data provided by official **NVD** Analysts. Additionally, submitted data will immediately be added to the **NVD**, and audits will be performed less frequently (at a minimum of one in ten).
4. *NVD:* acceptance level reserved for **NVD** analysts. This data is considered equal to that of the provider acceptance level.

These acceptance levels are particularly useful when filtering data for **CWE** classifications, as sometimes **CNAs** disagree with each other as illustrated in chapter 3.

2.2 The Common Weakness Enumeration (CWE)

The **CWE** is a community-driven list of over 900 weaknesses [28] constituting conditions "in a software, firmware, hardware, or service component that, under certain circumstances, could contribute to the introduction of vulnerabilities." [39] Strictly speaking, a Weakness is not a vulnerability, but the underlying root cause(s) that lead to said vulnerability. Examples of more general root causes include "missing authentication", "improper bounds check", or "stack-based buffer overflow" [28]. This is particularly useful for vulnerability labeling as it illuminates what needs to be addressed to eliminate the vulnerability. Hence, why the **NVD** uses the **CWE** as labels for "vulnerability types" [7].

2.2.1 CWE Relationships

The **CWE** is organized in a forest-like structure, where the further up the trees you go, the more abstract the weaknesses are, as illustrated in Figure 2.1. These levels of abstraction correlate with the amount of specificity present in the **CWEs**.

Following are the four levels of abstraction [39]:

- *Pillar*: Highest level of weakness that cannot be abstracted any further, describing a mistake but implying nothing specific regarding affected resources or where such a mistake was made. It is meant to act as a theme for the class, with base and variant weaknesses below it.
- *Class*: More specific than a pillar weakness but still relatively abstract and typically independent of any specific language or technology. Typically, describes issues in terms of one or two of the following dimensions:
 - Behavior: An action that a product (software, hardware, firmware) takes.
 - Property: "Security-relevant characteristic of an individual resource or behavior that is important to the system's intended security model, which might change over time".
 - Resource: "Entity that is accessed or modified within the operation of the product, such as memory, CPU, files, or sockets. Resources can be system-level (memory or CPU), code-level (function or variable), or application-level (cookie or message)."
- *Base*: Abstract weakness but contains sufficient details to infer specific methods of detection and prevention. Typically, describes an issue in terms of two or three of the dimensions featured in the *Class* abstraction, including technology and language.
- *Variant*: A concrete weakness typically linked to a specific language or technology. Typically, describes issues in terms of three to five of the dimensions mentioned in

the levels of abstraction described above.

There are cases where **CWEs** won't be part of any of these abstraction levels if they're considered composite **CWEs**. These are **CWEs** with composite structure relations explained in subsection 2.2.1.1. These are **CWEs** that consist of two or more other weaknesses that must be included lest the risk of exploitation is either greatly reduced or eliminated, or the nature of the composite as a vulnerability changes. These will be labeled as *Compound* in the *Abstraction* field of the **CWE** entry.

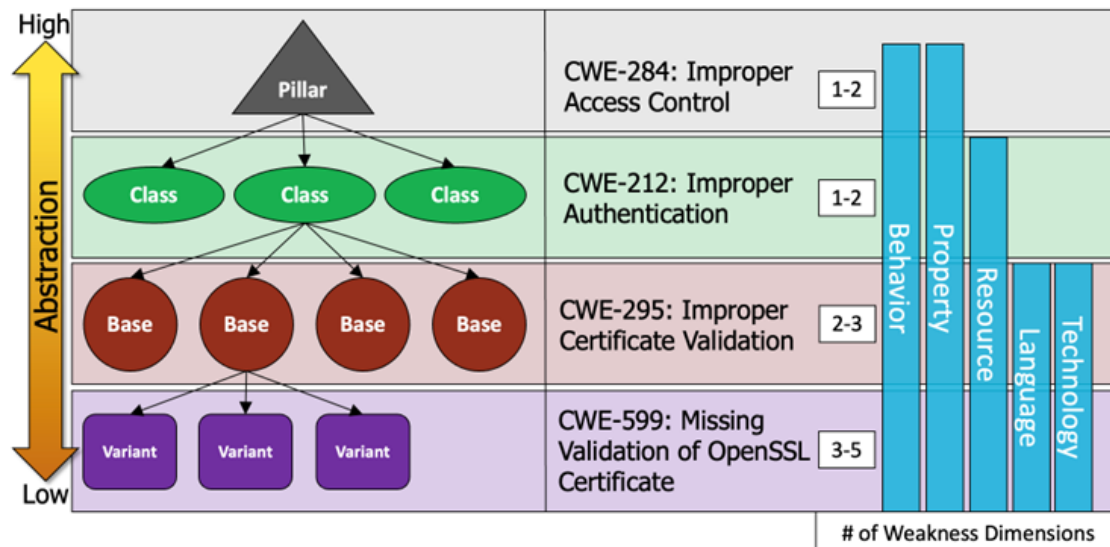


Figure 2.1: Abstraction tree example visualizing **CWE** parent and child relationships [28].

It is worth noting that the **CWE** is organized in a forest **like** structure and not a perfect forest. There are several **CWEs** with multiple accessible paths, meaning they have multiple ancestors. Additionally, there are times where **CWE**'s have children or parents in the same level of the tree, or parent-child relationships skip multiple levels of the tree. Finally, Variants are not necessarily exclusive in being leaf nodes. Regardless, most of the entries follow the rules of a forest structure, although it's not completely consistent. The entire forest structure of **CWE** view 1000 and 1003 is illustrated in Figure A.1 in Appendix A for reference. As illustrated in sub-Figure A.1a, view 1000, containing every **CWE** entry and their relationships, is far from a perfect forest. For an explanation and definition of "CWE views", check subsection 2.2.3.

2.2.1.1 Special Relationships

Besides parent-child relationships the **CWE** contains more specific relationships including:

- Chaining structure relations: These consist of `StartsWith` meaning the first weakness of a chain; `CanPrecede` meaning the weakness is primary to others; and `CanFollow` meaning the weakness is resultant from others[40].
- Composite structure relations: These include `Requires`, which refers to a component-weakness of the weakness in question, and `RequiredBy`, which refers

to a composite-weakness that the weakness in question is a component of [40].

- Similarity relationships: These include `CanAlsoBe`, meaning the weakness in question can be perceived as another weakness in different contexts, and `PeerOf` referring to similarities not covered by any other relationships. [41].

2.2.2 CWE Vulnerability Mappings and Content

Ideally *Base* or *Variant* level weaknesses should be used for classification however *Class* level **CWEs** may be used if there is no accurate *Base* or *Variant* **CWE** available. For further clarity, there are **CWE** vulnerability mappings defining the applicability of weaknesses for vulnerability labeling. These include [28]:

- **ALLOWED**: The **CWE** ID can be used for vulnerability mapping.
- **ALLOWED-WITH-REVIEW**: The mapping notes need to be thoroughly reviewed before labeling.
- **DISCOURAGED**: The **CWE** ID should Ideally not be used.
- **PROHIBITED**: The **CWE** ID must never be used.

Furthermore, **CWE** entries contain extensive documentation following an extremely exhaustive schema. The most relevant fields for this study are the mandatory description field, along with the optional extended description and demonstrative examples fields [9]. The demonstrative example field illustrates how each weakness may appear in code. Intro-text is also included, detailing the context of the example, and finally, an explanation of the code [41].

2.2.3 CWE Views

Besides the four weakness abstractions, there are also specific collections of weaknesses organized for various purposes, so-called **CWE** Views. Usually, these are a subset of the **CWE**, allowing for easier navigation of the **CWE** list according to some specific use case. [28] Notable views in regard to this study are views **CWE-1003** and **CWE-1000**. **CWE-1003** is the **NVD** dedicated view for simplified vulnerability mapping that the **NVD** uses for most of their vulnerability categorization [35]. This view is dedicated to effective vulnerability categorization and actually does follow a perfect forest structure as illustrated in Figure A.1b in Appendix A. **CWE-1000** (Research Concepts) contains all **CWE** entries and is meant to facilitate research into weaknesses and their interdependencies [42]. The **NVD** also uses this view for rare cases where **CNAs** provides **CWEs** that are more specific than the weaknesses available in **CWE-1003** [35].

2.3 Natural Language Processing (NLP)

This section thoroughly introduces the concept of **NLP**, **LM**, and the major milestones and developments that inform the introduction of **LLMs**. We introduce **NLP** by defining **LM** and introducing its most fundamental building blocks, such as **Statistical Language**

Model (SLM) and n -gram models in subsection 2.3.1. RNNs are introduced in subsection 2.3.2, informing the need for the concept of "attention" explored in subsection 2.3.3. The introduction of "attention" leads us to the fundamental building block for larger language models, the Transformer, in subsection 2.3.4. Following the Transformer, the precursor to LLMs, the **Pre-trained language model (PLM)** is shortly explored in subsection 2.3.5 before finally providing a well-informed definition of the LLM in 2.3.6. Finally, we introduce **RAG** and why it's useful in subsection 2.3.7.

2.3.1 Language Modeling (LM)

NLP can be summarized as **Artificial Intelligence (AI)** focused on enabling computers to understand the meaning and intentions of human language [43]. This seemingly straightforward task poses a complicated and longstanding challenge, as machines cannot naturally grasp the semantic complexities of higher-order natural languages. To address this dilemma, **Language Modeling** was introduced. **LM** aims to model the generative likelihood of a string of words given a language in an effort to predict the probabilities of future or missing tokens [18]. This probability can traditionally be modeled as a set of conditional probabilities $\mathbb{P}(x_t | \mathbf{x}_{<t})$, in terms of which the probability of a word sequence \mathbf{x} can be expressed as the following product.

$$\mathbb{P}(\mathbf{x}) = \mathbb{P}(x_1)\mathbb{P}(x_2 | x_1) \dots \mathbb{P}(x_t | \mathbf{x}_{<t}) = \prod_{t=1}^T \mathbb{P}(x_t | \mathbf{x}_{<t}) \quad (2.1)$$

In equation 2.1, $\mathbf{x}_{<t}$, referred to as the history, is the sequence x_1, x_2, \dots, x_{t-1} , x_t is the prediction and T is the sequence length. The computational mechanism for obtaining these conditional probabilities is what is referred to as a *language model*. The model parameters are trained by maximizing the probability of real-world data under the model [44], enabling token prediction and generation in an autoregressive manner. These types of Markovian models are also known as **Statistical Language Model (SLM)** [17]. Specifying further, **SLMs** with fixed context is known as n -gram models, where two histories are treated as equivalent if they end in the same $n - 1$ words. Specifically predicting the next token based on the $n - 1$ previous tokens as such:

$$\prod_{t=1}^T \mathbb{P}(x_t | \mathbf{x}_{<t}) = \prod_{t=1}^T \mathbb{P}(x_t | x_{t-1}, \dots, x_{t-n+1}) = \prod_{t=1}^T \mathbb{P}(x_t | \mathbf{x}_{t-n+1:t-1}) \quad (2.2)$$

Although **SLMs** have been found useful in various **NLP** tasks such as speech recognition, language translation, and information retrieval, they suffer from the curse of dimensionality. As n increases, data sparsity worsens and word sequences present in the training data are less likely to be present in the testing data, vice versa [45]. This is due to the n -grams extremely high dimensionality for larger values of n , determined by $V^n - 1$, given vocabulary size V . In other words, as n increases, the number of independent model parameters grows exponentially [44]. This complicates the estimation of higher-order languages as it requires calculating the joint distribution between an enormous number of discrete random variables.

2.3.2 Recurrent Neural Networks (RNNs)

While various smoothing techniques were developed to deal with the high dimensionality challenges of n -gram models, the introduction of the **Neural Language Model (NLM)** [45] revolutionized language modeling [46] in this respect. **NLMs** addressed the limitations of n -gram models in two ways: using lower-dimensional real-vector embeddings for words rather than one-hot encodings and through the use of neural networks to represent the language model. The real-vector feature embeddings provided more generalization, each feature representing different aspects of a word. The neural network reduced the dimensionality of the model from exponential to linear, enabling wider context windows [46]. This resulted in the neural model f_θ expanded on below:

$$\mathbb{P}(x_t | x_{t-1} \dots x_{t-n+1}) = f_\theta(x_t, \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots, \mathbf{x}_{t-n+1}) \quad (2.3)$$

where x_t is the word prediction at time t , $\mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots, \mathbf{x}_{t-n+1}$ denotes the embeddings of their respective words, f the neural network and θ the learned network parameters. With this approach, dimensionality scales linearly with vocabulary size and the order n , rather than exponentially as before [45].

After the introduction of **NLMs**, various neural modeling methods were developed, most notably, the **Recurrent Neural Network (RNN)**. **RNNs** are specifically designed to process sequential data by utilizing recurrent connections, allowing information to cycle within the network and making them particularly suitable for **LM**. At each time step, an RNN utilizes an input layer, a hidden layer, and an output layer as visualized in Figure 2.2. First, the hidden state gets calculated as follows:

$$\mathbf{h}_t = \sigma_h(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h) \quad (2.4)$$

where σ_h is an activation function; \mathbf{x}_t is the input word embedding; \mathbf{h}_{t-1} is the previous hidden state; \mathbf{W}_{xh} and \mathbf{W}_{hh} are learned weight matrices between the input and hidden layer, and the recurrent connection, respectively; finally, \mathbf{b}_h is a bias vector. Given equation 2.4, the predicted output of each time-step can be computed as follows:

$$\mathbf{y}_t = \sigma_y(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y) \quad (2.5)$$

where \mathbf{W}_{hy} is the learned weight matrix between the hidden layer and output layer. [47]

2.3.3 Attention

As with any **NLM**, the parameters (weights) in the **RNN** are trained via backpropagation. However, as the gradients are propagated backwards in time, they become progressively sensitive to the vanishing and exploding gradient problem, making it difficult for **RNNs** to learn long-term dependencies. Many **RNN** based architectures have been proposed to address this, such as the LSTM or GRU, but one that stands out amongst the rest is the attention mechanism [47]. In principle, attention allows the language model to select for high-value information by allocating more attention to important sections of the input sequence. In this way, it can process more information more efficiently and accurately,

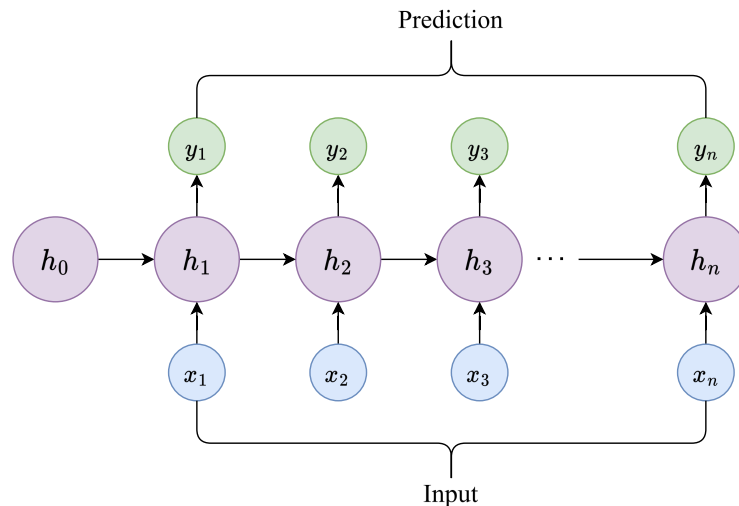


Figure 2.2: Basic RNN architecture visualized (reproduced from Figure 1 in [47]).

alleviating the exploding and vanishing gradient phenomena.

The attention mechanism relies on computing a context vector at each time step c_t that represents the context relationships between the current output symbol and each term of the entire input sequence. We can then compute the output of each time step as $\text{RNN}(c_t)$. This can be achieved in multiple ways, but in general most attention models include: a feature matrix, encoding the source data features in \mathbf{K} , referred to as a key; a task-related representation matrix or vector, \mathbf{q} , referred to as a query; a score function, f ; a distribution function, g ; and a function, ϕ , that, given a set of values and their corresponding weights, returns a vector.

First, we obtain an energy score, $e = f(\mathbf{q}, \mathbf{K})$ describing how keys and queries are matched/combined (most commonly additively or multiplicatively). These energy scores measure the relevance of keys to specific queries, and this relevance influences what the model pays attention to when generating output. The scores are mapped to attention weights through $\alpha = g(e)$, where g usually corresponds to the softmax function. Finally the weights and their corresponding values are fed into ϕ , commonly implemented as a weighted sum of the values \mathbf{V} as follows:

$$\mathbf{c} = \phi(\{\alpha_i\}, \{\mathbf{v}_i\}) = \sum_{i=1}^n \alpha_i \mathbf{z}_i \quad (2.6)$$

where $\mathbf{z}_i = \alpha_i \mathbf{v}_i$. The matrix \mathbf{V} is often necessary when neural networks compute the context vectors and introduces a new feature representation called the *value*, each element of the *value* corresponding to an element in \mathbf{K} . Often, the *key-value pair* \mathbf{V} and \mathbf{K} are treated as the same matrix. However, they can also be explicitly separated, constituting two different representations of the same input data. [48].

2.3.4 Transformers

Despite the introduction of attention and a multitude of other solutions for the vanishing and exploding gradient problem, the inherent sequential nature of **RNNs** makes them difficult to parallelize. However, larger-scale **NLP** involving longer sequence lengths and larger training corpora necessitates parallelization for computational power and efficiency. As such, transformers [49] were introduced to address the fundamental constraint of sequential computation by doing away with the **RNN** architecture entirely and completely relying on the *self-attention* mechanism:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \quad (2.7)$$

where \mathbf{Q} , \mathbf{K} , and \mathbf{V} are the queries, keys, and values, respectively, packed together into their respective matrices, and $\frac{1}{\sqrt{d_k}}$ is a scaling factor. \mathbf{Q} , \mathbf{K} and \mathbf{V} are calculated as $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$, $\mathbf{K} = \mathbf{X}\mathbf{W}_K$, $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ respectively where \mathbf{W}_Q , \mathbf{W}_K and \mathbf{W}_V are learnable *query*-, *key*- and *value* weights and \mathbf{X} is the input matrix containing the input token embeddings. Reusing the same input matrix \mathbf{X} to encode queries, keys, and values in this manner, without relying on recurrence, is what is referred to as *self-attention* [50].

Looking at equation 2.7 from the perspective of **LM**, a query q from \mathbf{Q} corresponds to an individual token/word x_t , the vector representation of which is contained in \mathbf{X} . \mathbf{K} and \mathbf{V} , on the other hand, will contain information and features of $x_{<t}$. Thus, $f \cdot$, represented in equation 2.7 by dot product attention $\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}$, expresses the relevance of a particular key k to the query q . In this way, more *attention* will be put on the value associated with that key, the more relevant the query is to said key [50].

The Transformer architecture illustrated in Figure 2.3 is composed of an *encoder-decoder* architecture with 6 identical encoder and decoder layers. An *encoder-decoder* is a neural network architecture where the encoder attempts to capture features and information from the source sequence by mapping it to a context vector, and the decoder uses said context vector along with the previous output to predict the next symbol. The encoder layers have two sublayers, the first being a multi-head self-attention mechanism and the second, a position-wise fully connected feed-forward network [49]. The multi-head attention refers to separating the attention block into multiple attention layers, so-called heads, that each can focus on separate aspects of the input text. This enables the model to capture more information about the input string by computing multiple contextual representations for each symbol [50]. These attention heads are fully parallelizable, further improving computational efficiency. The decoder is comprised of the same sub-layers as the encoder, in addition to a third sub-layer, which performs multi-head attention over the output of the encoder stack. Additionally, utilizing a mask and offset embeddings to ensure that the predictions only depend on the previous outputs. Finally, since the architecture is no longer inherently sequential, the embeddings utilize positional encodings to keep track of the input sequence order [49].

Together, these components, along with additional details beyond the scope of this section, come together to create a computationally efficient model. Additionally, it exhibits a remarkable understanding of complex, long-distance dependencies and relationships by

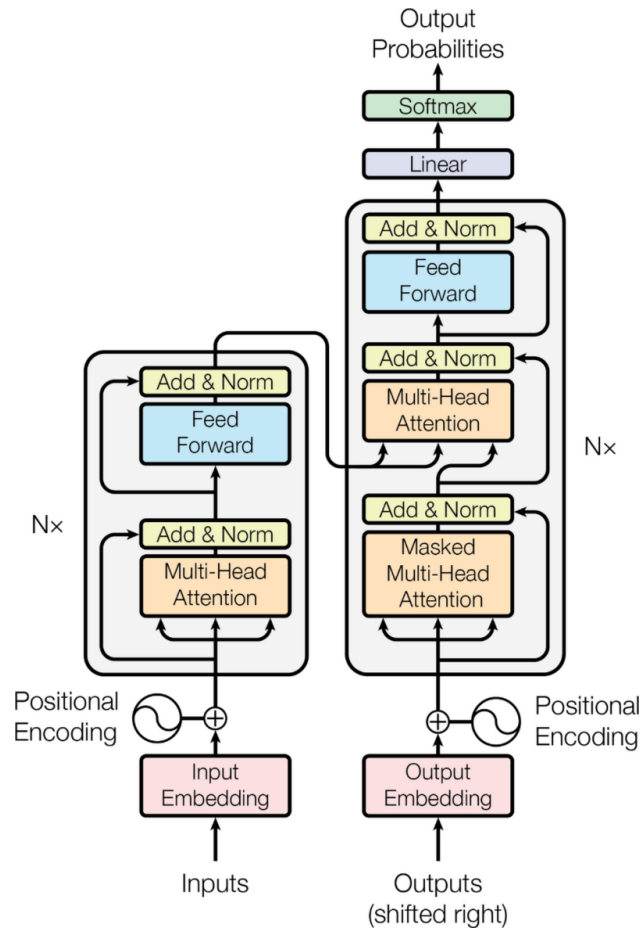


Figure 2.3: Transformer architecture (reused from Figure 1 in [49]).

virtue of the multi-head self-attention mechanism.

2.3.5 Pre-trained Language Models (PLMs)

Building on the great success of the Transformer architecture, **PLMs**, utilizing self-supervised learning, transformer encoders, decoders, or both, began to emerge. Pre-training is built on the idea of transfer learning, where a model learns some abstract and generalizable task that it can then apply to more efficiently and accurately solve some more specific problem [51]. In the case of **LM**, the model may be pre-trained to predict the likelihood of an input sequence by maximizing the following likelihood

$$L_1(\mathcal{U}) = \sum_i \log \mathbb{P}(u_i | u_{i-k}, \dots, u_{i-i}; \Theta) \quad (2.8)$$

also known as the cross-entropy. Here $\mathcal{U} = \{u_1, \dots, u_n\}$ is an unsupervised corpus of tokens, k is the context window size and Θ is the network parameters modeling conditional probability \mathbb{P} [52]. After the pre-training step, a labeled dataset tailored for a specific downstream task can be used to finetune the model. This is known as the "pre-training then fine-tuning" paradigm [51]. Intuitively, one might imagine playing a huge crossword containing a large corpus of data and picking up on various patterns, relationships, and fundamental features in natural language as they go. In this way, **PLMs** acquires a

considerable amount of lexical, syntactic, and semantic knowledge during pre-training, facilitating NLP tasks specialization through fine-tuning on minimal labeled data [46].

2.3.5.1 BERT

PLMs are made possible by the enhanced efficiency and language representation capabilities of transformers. This is due to the large-scale deep learning necessary to adequately capture lexical, syntactic, and semantic language features for generalization [46]. Consequently, PLMs are not only evaluated on accuracy and efficiency but also on how many downstream NLP tasks they can generalize to. One model exhibiting exceptional performance in this respect was the **Bidirectional Encoder Representations from Transformers (BERT)**, advancing the SOTA on 11 NLP tasks [53]. The predecessors to BERT, such as GPT-1 are trained on predicting the next token given the previous tokens (auto-regressively) using cross entropy as in equation 2.8 [52]. In other words, the model is unidirectional, entailing future invisibility, making it particularly suited for language-generation problems [46]. However, this unidirectionality is sub-optimal for sentence-level and token-level tasks such as question answering, where it is crucial to incorporate context from both directions. To address this, BERT incorporates a **Masked Language Model (MLM)** pre-training objective where the model is tasked with predicting randomly masked tokens in the input string. This enables the representation to fuse left and right context, facilitating the pre-training of a deep bidirectional transformer. Furthermore, to improve generalization on downstream tasks involving understanding of sentence relationships, BERT also pre-trains on the next sentence prediction task [53]. BERT was a major achievement in PLM research, establishing itself in NLP systems, outperforming humans in some downstream tasks, and inspiring a vast amount of research into masked LM [51].

2.3.6 Large Language Models (LLMs)

Although PLMs were a significant breakthrough in NLP, moving the field beyond highly specialized models that usually could not be used for more than one or a few tasks [51], they still required adequate fine-tuning and sometimes task-specific pre-training on unseen problems [19]. Since the release of GPT and BERT, PLMs have scaled exponentially in terms of both parameter sizes and training corpora [51]. This upscaling, partly made possible by the computational efficiency of the Transformer architecture, moved LM forward towards LLMs with the arrival of GPT-3, its success inspiring the development of numerous modern models [19]. In this sense, LLMs are simply scaled up, "large", PLMs. However, this seemingly minor adjustment enables them to generalize without any task-specific training and significantly improves task-agnostic, few-shot, and zero-shot capabilities [51]. As a result, a trend of complex task unification has emerged, where LLMs are exhibiting surprising, *emergent abilities*. For one, LLMs are capable of **In Context Learning (ICL)**, enabling prompt tuning, where demonstrative examples are concatenated along with the model query for the model to learn from. These examples are written in a natural language and help teach the model through demonstration without necessitating parameter updates. This significantly reduces computational cost and provides the user with an interpretable, convenient interface for communication and knowledge sharing with

the model [54].

In recent years, with the proliferation of ChatGPT, there has been a growing interest in LLMs as they seemingly excel in a multitude of NLP tasks. Their apparent usefulness in language tasks enables a gradual and seamless incorporation into our professional and everyday lives. Indeed, there has been a sharp increase in arXiv papers related to LLMs following the introduction of ChatGPT, the scaling effect on LLMs resulting in our closest estimation of general-purpose task solvers [18]. These emerging capabilities present in LLMs and their SOTA performance in NLP tasks make them perfect candidates for vulnerability classification via natural language descriptions.

2.3.7 Retrieval Augmented Generation (RAG)

While LLMs exhibit revolutionary generalization capabilities, they are still subject to *hallucinations* which pose a significant challenge for the reliability and trustworthiness of the models [55]. Hallucinations are instances where "the model-generated content does not align with or cannot be verified by existing world knowledge", often due to imperfect training data and limitations in training and generation methods [18]. Furthermore, the lack of domain-specific knowledge in fields requiring it, such as medicine or law, often exacerbates this limitation, with hallucination rates reaching 88 percent [55]. To mitigate this issue, RAG attempts to ground LLMs in real-world factual knowledge by introducing an external knowledge source via knowledge-retrieval [11].

The original model leverages a retriever $p_\eta(z | x)$ that based on input sequence x retrieves text documents z , parametrized by η . The generator $p_\theta(y_i | x, z, y_{1:i-1})$ parametrized by θ , can then utilize z as additional context along with previous tokens, $y_{1:i-1}$ and original input x when generating target sequence y . The retriever and generator are trained jointly by minimizing the negative log likelihood $-\sum_j \log p(y_j | x_j)$ given a "fine-tuning training corpus of input/output pairs" (x_j, y_j) [11].

In general RAG falls under **Retrieval-Augmented Large Language Models (RA-LLM)** and can be separated into three major parts, *Retrieval*, *Generation* and *Augmentation*.

The material in the following subsections 2.3.7.1, 2.3.7.2, 2.3.7.3, and 2.3.7.4 is primarily drawn from Ding *et al.* [55].

2.3.7.1 Retrieval

Given an input query, the model retrieves the top-K most relevant documents from an external knowledge source such as a database or dense vector index. Wikipedia is one of the most common such retrieval sets, but this could also be closed-source organizational documents for internal use. There are two types of retrieval:

- *Sparse*: Sparse retrievers are word-based, such as TF-IDF, where terms are weighted on their inverse term frequency "so that matches on less frequent, more specific, terms are of greater value" [56]. There is also BM25, which builds on TF-IDF but takes into account document length normalization and term frequency saturation [57].

- *Dense*: Dense retrieval utilizes a continuous vector space to embed the query and documents with certain criteria, such as semantic similarity. These retrievers are often trainable, improving adaptability and decreasing the reliance on database construction quality and query generation.

Furthermore, there are different levels at which the external corpus can be indexed, also known as Retrieval granularity. These range all the way from whole documents to individual tokens, with chunk retrieval becoming the most popular for generative tasks, providing a good tradeoff between saving space and search cost.

Finally, there are different pre- and post-retrieval enhancements, these include but are not limited to

- **Query expansion** where pseudo documents are generated via another **LLM** to expand the query with relevant information.
- **Query rewrite** which aims to rewrite the query to be more conducive for retrieval, usually via another **LLM**.
- **Query augmentation**, which utilizes retrieval outputs as a new query to further relevant information retrieval. For example, intermediate results of chain of thought reasoning can be used for further retrieval, after which the results can be integrated into the chain of thought [18].
- **Document re-ranking** documents retrieved via different approaches are assembled and re-ranked to boost robustness. For example, this can be done by prompting **LLMs** to self-check and perform confidence assessment [18].
- **Noise reduction**, which attempts to reduce the irrelevant information in retrieved documents. For example, an **LLM** can be prompted to do relevant information extraction or summarization [18].

2.3.7.2 Generation

There are two types of generation strategies, white box and black box, which differentiate themselves based on whether model parameters are accessible or not, respectively. Thus, while white box generation enables more adaptive generation through parameter optimization, the successes of closed-source large models, infeasible for most smaller stakeholders to train and build from scratch, have made black box generation mainstream in **RA-LLM**. The resulting secrecy surrounding the internal structure of these models, therefore, forces black-box **RA-LLM** to focus more on the retrieval and augmentation process.

2.3.7.3 Augmentation

Augmentation concerns itself with how the retrieved results are incorporated into the generation step. The most common strategies for black-box retrieval involve either concatenating the retrieved results with the input query in some way (Input-Layer Integration) or joining retrieval and generation results post-hoc (Output-Layer Integration). This could involve a refining mechanism where an already generated answer

is adjusted with the retrieved results in mind to enhance accuracy. Methods for assessing the necessity for retrieval in the first place might also be employed.

Another aspect of augmentation is retrieval frequency. While the standard method retrieves once to find all relevant information, some solutions incorporate every-token or every-n-token retrieval, which may increase accuracy but worsen computational cost.

2.3.7.4 Training

One of the key strengths of **RAG** is its ability to acquire knowledge from external sources without additional training. Even so, pre-training or fine-tuning of the retriever and/or generator to better adapt the retrieved context for domain-specific tasks can substantially improve performance. These training strategies can be separated into three major categories.

- **Independent training**, where the retriever and generator are trained separately. For example, **LLMs** can be trained to effectively leverage the retrieved context through negative log likelihood loss.
- **Sequential Training**, where the components are trained sequentially. First, either the retriever or the generator is independently trained before being frozen and training the other component. Often, existing models such as **BERT** can be employed to avoid the need for pre-training of the generator.
- **Joint Training** is what was used in the original paper [11] and involves an end-to-end process for updating the retriever and generator simultaneously.

However, in most black-box scenarios, neither the parameters nor the inner workings of the model are available, forcing a training-free approach. In such a scenario, there is a myriad of different approaches that may be taken in terms of refining the original prompt.

2.4 Related work

2.4.1 Cipollone and Transformer Based Vulnerability Detection

As explained in previous sections, zero-day vulnerabilities pose a unique and serious threat to software security, the mitigation of which requires a laborious process of analyzing vast amounts of text data from various communication channels. In an effort to alleviate this and expedite vulnerability detection, Daniele Cipollone explores the potential of utilizing **LLMs** and transformer-based models for automated cybersecurity analysis of communication channels, specifically GitHub. In short, detecting vulnerabilities through the analysis of GitHub issues.

2.4.1.1 Approach

To this end, Cipollone proposed a pipeline illustrated in Figure 2.4, consisting of an embedding step to enable true/false classification via an XGBoost classifier, followed by

the utilization of an LLM for further scrutiny of the classification results and to provide a concise description of the vulnerabilities detected. The idea here is to combine the power of embedding models to capture semantic similarities and LLMs to encode contextual information and facilitate a nuanced understanding of each issue. The XGBoost model is a robust gradient boosting algorithm that uses the embedding as an input feature to predict the relevance of an issue to vulnerabilities. It provides accurate predictions by utilizing structural information derived from the embedding models and is widely renowned for its exceptional performance in classification tasks.

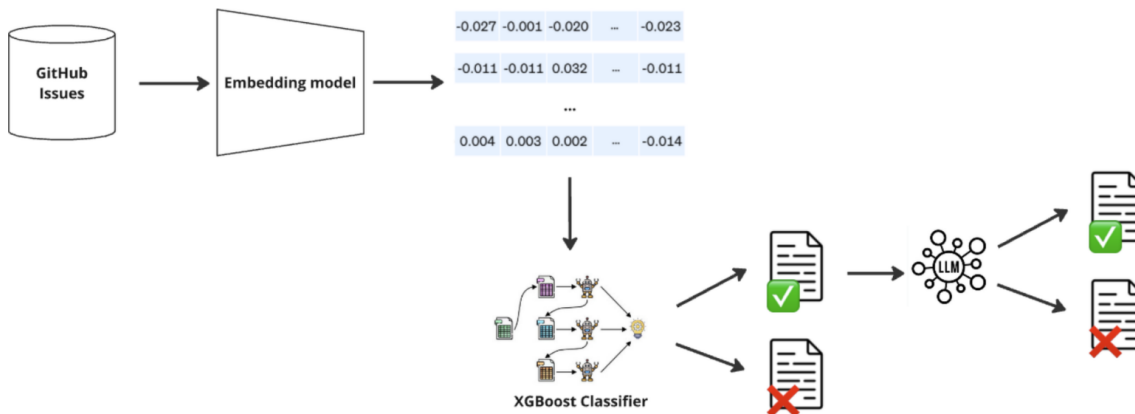


Figure 2.4: Cipollone’s proposed pipeline (reused from Figure 3.4 in [4]).

2.4.1.2 Data collection

Cipollone utilized the NVD, scraping the CVEs published between January 1, 2019, and June 2, 2024, and isolating the CVEs with references to GitHub issues. He further limits the data to only the issues of the top 50 repositories with the highest number of related vulnerabilities from 2023 onward. He does this to keep the data manageable. The non-vulnerability-related issues from these repositories were also proportionally included to serve as negative examples. Finally, he removed repositories that did not contain a sufficient number of issues and issues exceeding 8191 tokens, as this was the maximum context window of the embedding model `text-embedding-3-large` [4] used in combination with the XGBoost classifier.

2.4.1.3 Why GitHub?

After analysis of the CVEs collected for frequency of domains used, Cipollone recognized that an overwhelming majority of the CVE references were directly associated with GitHub, as seen in Table 2.1. Thus, he infers that GitHub and issue tracking are some of the most essential resources in vulnerability auditing and analysis. As such, Cipollone deliberately utilizes GitHub issues as the primary communications channel for analysis. This reasoning also informed our decision to continue focusing on GitHub issues, given the inherent nature of this thesis as an extension of Cipollone’s work.

Domain	Count
github.com	42,165
git.kernel.org	10,461
vuldb.com	8,332
lists.fedoraproject.org	4,790
patchstack.com	4,256
lists.apache.org	3,805
www.zerodayinitiative.com	3,620
packetstormsecurity.com	3,523
portal.msrc.microsoft.com	3,340
plugins.trac.wordpress.org	3,258
wpscan.com	3,238
exchange.xforce.ibmcloud.com	3,048
www.wordfence.com	2,752
www.openwall.com	2,700
bugzilla.redhat.com	2,531

Table 2.1: Top frequent domains used in references of the CVEs scraped by Cipollone (reused from Table 3.2 in [4])

2.4.1.4 Results and Conclusion

Cipollone compared the different approaches present in the proposed pipeline, incorporating zero-shot and few-shot learning in the LLM-step. The final combined approach, incorporating the entire pipeline described above, achieved the greatest accuracy of around 90%. However, Cipollone’s approach suffers from a high false positive rate and therefore a comparatively low precision score of 0.69 when considering the positive (vulnerable) entries in the test sample. Furthermore, Cipollone evaluated the accuracy of the vulnerability descriptions generated at the end of his pipeline by comparing their similarity to the descriptions found in the NVD. Achieving an average similarity score of 7.8%. Cipollone concludes that although there is room for improvement in future work, leveraging embedding models shows strong potential as a tool for vulnerability detection that balances classification accuracy and detailed vulnerability descriptions.

While vulnerability detection is a crucial step in addressing a vulnerability and, as such, narrowing the window of opportunity for exploitation, identification is of equal importance. Cipollone does provide a vulnerability description. However, it does not constitute a formal classification. This work attempts to close the loop by extending Cipollone’s pipeline, utilizing the generated vulnerability description for formal vulnerability classification via RAG.

2.4.2 Automated CWE Categorization

Mapping weaknesses to vulnerabilities is clearly a fundamental step in CTI and thus vulnerability exploitation and mitigation around the world. However, the process of identifying pertinent weaknesses given a vulnerability is a cumbersome and time-consuming process subject to human error. As such, there have been various studies

researching potential methods of automating this process.

As early as 2017 Na *et al.* [58] Explored the use of vulnerability overview text and **CWE** IDs gathered from the **NVD** to generate a Naive Bayes classification model. Na *et al.* focused on the top 10 most frequent **CWEs**. Separately evaluating Naive Bayes classification across the top 2, 3, 5, and 10 **CWEs**. The results showed a significant decrease in accuracy as the number of classification categories increased, dropping from a 99.8% accuracy on the top two **CWEs** to 75% on the top 10. Later, Aota *et al.* [59] proposed a scheme utilizing the Boruta feature selection algorithm and random Forest classifier in an effort to automatically classify vulnerability descriptions. Limiting themselves to 10 **CWE** classes each corresponding to a minimum of 100 **CVE** entries, Aota *et al.* achieved a classification accuracy of 96.92%.

The proliferation of **Deep Learning (DL)** enabled methods for tackling the sheer number of applicable **CWEs** and challenges inherent in effective text representation. Thus, **DL**'s advantages in long-term dependencies and automatic feature extraction inspired further research into automated vulnerability Categorization.

Following in the footsteps of Na *et al.*, Aghaei *et al.* [12] explored automating the classification of **CVEs** by introducing ThreatZoom. ThreatZoom utilized a tree-like hierarchical method drawing upon the structure of the **CWE**. Also, referred to as "Hierarchical Decision-Making", it followed a top-down approach by training a neural network classifier for each node in the **CWE** tree. At each decision-making step, either another classification node or a leaf node was chosen until a leaf node was reached. The resulting prediction then became the sum of the list of candidates, using a multi-hot representation at each decision step. This enabled multiple **CWE** predictions where a correct classification was defined as the **CVE** label being included in at least one of the **CWE** candidates. Additionally, the weights of the networks were initialized by the TF-IDF score of each entry neuron and their corresponding classes in the hidden layer. This further extracted statistical, as opposed to purely semantic, features. Aghaei *et al.* achieved an accuracy of 75% on the MITRE dataset and 92% on the **NVD**, which at the time contained 364 and 116 **CWE** classes respectively.

Pan *et al.* [16] leveraged BiGRU and TextCNN to address the challenge of the ever-increasing, large number of classification classes. While ThreatZoom supported relatively many **CWEs**, its hierarchical tree-like, multi-label structure sacrificed precision due to the large number of redundant candidate weaknesses. Pan *et al.* proposed the BiGRU-TextCNN classification framework, where BiGRU captures long-term dependencies by considering past and future context while TextCNN captures local features across text via convolution. Through this framework, Pan *et al.* achieved an overall accuracy of 90% using 158 Base level **CWEs** across 59918 vulnerabilities, although struggling with weaknesses that have fewer observed vulnerabilities.

Congruent with the increasing popularity of deep learning, transformer-based models were establishing themselves. Their convenient adaptability to downstream tasks, utilization of the GPU [19], and effectiveness at general-purpose **NLP** [18] lent themselves well to automated **CWE** labeling.

Wang *et al.* [13] starts experimenting with BERT as early as 2021. Wang *et al.* focused on the top 10 **CWEs** after excluding the **CWE-NVD-Other** and **CWE-NVD-noinfo** classes. Transformer-based method, BERT was then compared with common previous methods of automated **CWE** labeling. This includes Naive Bayes, Decision Trees, Logistic Regression, CNN, and LSTM. Wang *et al.* achieved an accuracy of 90.7%, significantly outperforming the other methods examined.

Das *et al.* [60] explored pre-training BERT using **CVE** and **CWE** description by introducing V2W-BERT. V2W-BERT combines link prediction to map **CWEs** and a reconstruction decoder to consider both link classification loss and reconstruction loss simultaneously. This preserved the pre-trained description context of BERT when updating the link prediction model. Finally, this enables further classification of **CWEs** with few to no **CVE** training instances through description similarity. Das *et al.* focused on 124 **CWEs** distributed across three abstraction levels and used a hierarchical structure similar to ThreatZoom. Thus, sacrificing precision to enable multi-label predictions by predicting paths rather than individual **CWEs**. The evaluation of the model was separated into different categories depending on the number of **CWE** training examples. Achieving an overall accuracy of top 1, top 6, and top 20 accuracy of 83.6%, 91.4%, and 94.4%, respectively. Notably, when predicting **CWEs** not present in the training set, accuracy drops drastically. Decreasing to 14.9%, 44.0%, and 67.5%, respectively, across 336 samples between 2018 and 2020.

Returning to ThreatZoom, while showing great potential, it suffered from substantial overhead due to the need to train a separate model for each node of the MITRE tree. Recently, Kota *et al.* [14] proposed a solution to address this by focusing on view **CWE-1003**, which can be separated into a top and bottom layer. This enables the use of a layer classifier along with a top and bottom-layer cross-encoders to automate the classification of **CVEs** across 130 different **CWEs**. The encoders are trained with BERT under the hood using both **CVE** and **CWE** descriptions to capture semantic similarity. This enables understanding of the broader context and dependencies between the **CVE** descriptions and **CWE** information. Although the overall accuracy achieved (72.1%) was lower than that of ThreatZoom (92% when considering view **CWE-1003**), the result was a more practical and precise solution capable of capturing semantic similarity.

Finally, in recent developments, Albanese *et al.* [15] leveraged **CVE** descriptions in combination with simple **NLP** techniques to map previously unseen **CVEs** to their most likely **CWE**, introducing CVE2CWE. CVE2CWE utilizes **CVEs** to create a document for each **CWE** by merging the descriptions of all associated **CVEs**. In the training step, CVE2CWE can then create embeddings for the generated **CWE** documents by calculating the TF-IDF scores of the terms occurring within them. When predicting an unknown **CVE**, CVE2CWE embeds the **CVE** in the same way. This enables the use of cosine similarity over the **CWE** documents to retrieve the top-K predictions, akin to **RAG**. To ensure the prevalence of training data, Albanese *et al.* focused on the top **CWEs** ranging from the top 10 to the top 50, in increments of 5. The results showed a $k = 1$ accuracy ranging from 85.5% to 57.2%, decreasing as the number of categories increased. Furthermore, with 25 categories, CVE2CWE achieved accuracies of 69.9%, 82.5%, and 87.5% for $k = 1$, $k = 2$, and $k = 3$, respectively. Finally, Albanese *et al.* recognized that incorrect **CVE-**

to-CWE classification was often due to close cosine similarities between the embeddings of the correct CWE documents and the ones predicted. A more sophisticated accuracy score that offset the effect of CWE similarities was also measured to adjust for this. This metric showed a $k = 1$ accuracy ranging from 93.14% and 75.90%, capturing the true performance more accurately.

Excluding prohibited CWEs, at the time of this thesis, there are a total of 881 CWEs allowed for Categorization, only 42 of which are discouraged. This means that there are over 800 viable categorization classes available. Many of these labels are rarely ever applied, with the categorization following a logarithmic distribution. As such, collecting enough training data to properly equip a model for universal categorization is very difficult. With this in mind, many previous works limit themselves to the top CWEs or to CWE view 1003. Indeed, the methods that do address the large number of categorization classes either suffer from low precision and high overhead, as with ThreatZoom, or low accuracy, especially where there are few existing examples, as with Kota *et al.* and Pan *et al.* Only ThreatZoom even addresses more than 200 CWEs and subsequently achieves a lower accuracy, dropping from 92 to 75 percent when tested on the larger MITRE dataset over CWE view 1003. On the other hand, the wide applicability and inherent knowledge present in pre-trained LLMs on NLP tasks suit them well in addressing these limitations in the available data. While Fine-Tuning is an option, when considering LLM's tendency to hallucinate on knowledge-intensive tasks, the scattered, sparse, and limited data and extensive CWE documentation available, utilizing some sort of retrieval seems most appropriate. Indeed, Albanese *et al.* experimented with CWE documentation curation in combination with NLP and retrieval and found promising results. However, in common with the other previous works, they limited themselves to a smaller number of categories. Only addressing 25-50 categories, as their model still required training data for accurate document creation. Furthermore, they used a relatively primitive form of embedding using TF-IDF; subsequently, their accuracy suffered as classification categories increased. This work attempts to address all these issues and limitations by leveraging more sophisticated LLM embeddings and pre-trained retrievers along with the promising reasoning capabilities of LLMs through the use of RAG.

Chapter 3

Methodology

This chapter outlines the methods and experiments used in this research. Section 3.1 provides an overview of the steps taken for the project's execution; 3.2 thoroughly outlines the data collection and filtering process; 3.3 details the experiments conducted, and 3.4 explains the evaluation process of the compiled results.

3.1 Research Process

This work grounds itself on a thorough literature study of the technologies involved, current **SOTA**, and previous solutions in similar and relevant research. Although the specific nature of this research makes it particularly novel, the general idea of automated vulnerability categorization is nothing new, as illustrated in the related work subsection 2.4.2. As such, examination of previous work enabled inquiry into best practices, potential challenges, and earlier solutions and results. Additionally, various tutorials and supplementary resources on topics beyond the scope of the literature review were examined. Following the literature study, a thorough analysis of the **CWE** as its utilization in this study required a deep understanding of its inner workings, contents, and potential human errors. Additionally, Cipollone's pipeline was analyzed and expanded to include **CWE** categorization and reinitialized based on Cipollone's original code. This code was directly accessible through the author, Cipollone himself.

Expansion of Cipollone's pipeline included **CWE**-adjusted data collection and incorporation of vulnerability categorization through **RAG** and **LLMs**. Various **RAG** and **LLM** strategies were explored, considered, and designated for final experimentation based on their applicability to the particular use case. This pruning of available options was necessary given the extensive number of available strategies, rendering exhaustive experimentation unfeasible. From the experimentation, results were collected, thoroughly analyzed, and presented. Lastly, a baseline solution was constructed to compare our results against and evaluate their overall practical utility. The research process is visualized in Figure 3.1.

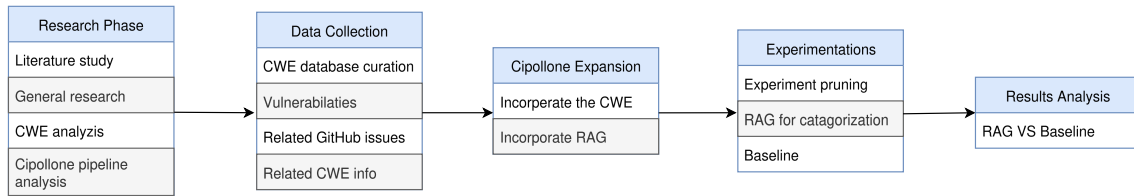


Figure 3.1: Research Process.

3.2 Data Collection

Due to the complexity of the **CWE** and process in which **CVEs** are categorized, meticulous and thoughtful data collection was required to ensure the validity and fairness of said data. Data collection and pruning were therefore considered carefully, based on official **NVD** and **CWE** documentation.

3.2.1 CVE and GitHub Pipeline

For the **CVE** and GitHub data collection, Cipollone’s pipeline, detailed in subsection 2.4.1.2, was utilized for consistency. However, the end date was changed to the time of data collection (2025-03-04) to enable the collection of more and newer data. Furthermore, we collected all related issues from 2019, rather than just the top 50 from 2023, to enable larger experiments along with experiments consistent with Cipollone’s methods.

3.2.2 CWE collection and filtering

The **CWE-API** [61] was utilized for collecting research view **CWE-1000** and **NVD** view **CWE-1003** in JSON format. These would serve as the primary knowledge sources for various types of retrieval specified in subsection 3.3.

When associating **CWEs** with **CVEs**, various peculiarities in the data and processes for how **CWEs** are assigned have to be considered and adjusted for. Foremost, **CVEs** may not be associated with any **CWEs**. In this case, they’re either assigned `NVD-CWE-Other` or `NVD-CWE-noinfo`. Such entries were promptly removed from the **NVD** data collected. Furthermore, **CVEs** may have multiple **CWEs** assigned to them as they may exhibit more than one weakness. Since this work limits itself to a single assignment, these **CVEs** were also removed. Justifying this decision further, an overwhelming majority of the assigned **CVEs** collected (around 97%) required only one **CWE** for categorization.

A large number of **CVEs** were labeled by multiple **CNAs** that sometimes disagreed with each other. In such cases, one could appeal to their acceptance level to resolve such a conflict. Furthermore, **CVE** entries include sources for the **CNAs** that labeled them, which can also be used for credibility evaluation. Thus, the **NVD** data was filtered, and the **CWE** label for each **CVE** entry was determined in the following order of steps.

1. All **CVEs** with no **CWE** data (labeled `NVD-CWE-Other` or `NVD-CWE-noinfo` for all **CNAs**) were removed.
2. For a given **CVE**; if there are labels from **CNAs** with acceptance level *Provider* or

NVD retain those labels and discard the rest.

3. For a given **CVE**; if there are labels from **CNAs** with source `nvd@nist.gov` retain those labels and discard the rest.

For some entries, the acceptance level is below *NVD* even though the source is `nvd@nist.gov`; this implies the **CNA** is an **NVD** analyst. As such, we assume this might be due to human error and prioritize this specific source after filtering based on acceptance level.

4. For a given **CVE**; If all the **CNAs** labeling left after the steps above agree with each other (are equal), assign the **CVE** this labeling; otherwise, discard the **CVE** as a data point.

The remaining **CNAs** will presumably have the same level of expertise, making it unclear who is right if they disagree, resulting in an unresolvable conflict. This case is exceedingly rare when analyzing the data.

5. Finally, out of the remaining **CVEs**, only retain those with a single-assigned **CWE** label.

3.3 Experimental design

The experimental design of this work was based on Cipollone’s classification model, detailed in subsection 2.4.1.1. Most of the experimentation was done without the inclusion of XGBoost, as it is not necessary for **CWE** categorization. However, XGBoost was included towards the end for completeness.

Firstly, Cipollone’s pipeline was extended with basic retrieval, which uses the vulnerability description generated at the end of Cipollone’s pipeline for retrieval. Along with this, the **CWE** documentation was processed and converted into structured markdown-formatted strings with three different levels of descriptiveness defining the amount of information to extract from the original documentation. These three levels of descriptiveness served as different experimental knowledge bases. The second experiment extended Cipollone’s **LLM** prompt to include a more structured and general vulnerability description, examining the potential of prompt engineering. The third experiment attempted to generate missing information in the original documentation for our knowledge base, exploring the potential of automated **LLM** documentation. The fourth and fifth experiments examined the advocacy of fusion retrieval and the potential of a **Recursive Abstraction Processing For Tree-Organized Retrieval (RAPTOR)** inspired retriever on the tree-like structure of the original **CWE** documentation, respectively. After this experimental phase, the original vulnerability description was combined with the retrieved candidate **CWEs** of the best retrieval solution. These were then fed to a **LLM** for final categorization, introducing reasoning and completing the **RAG** pipeline. Afterwards, different ways of providing the retrieved context with the original query were explored. The final pipeline incorporating the XGBoost classifier as in Cipollone’s original model is illustrated in Figure 3.2

Finally, to evaluate the viability and necessity of our **RAG** experiments, a straightforward baseline approach was evaluated. This approach consisted of giving an **LLM** the entire

original **CWE** documentation along with the generated vulnerability description generated at the end of Cipollone’s pipeline.

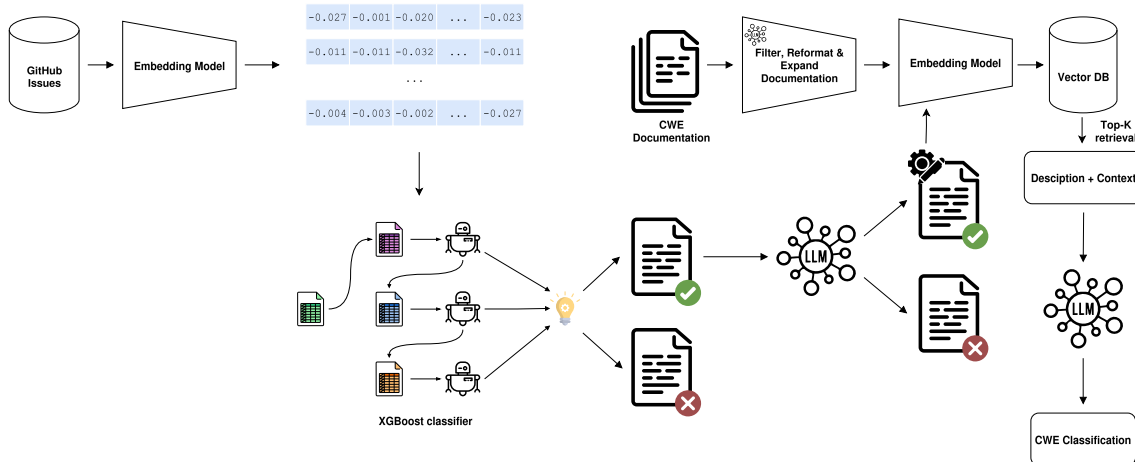


Figure 3.2: Final automated vulnerability categorization pipeline. A direct extension of Cipollone’s pipeline is illustrated in Figure 2.4.

3.3.1 RAG Methodologies

This methodology utilized Cipollone’s **LLM**-based classifier as a foundation to enable automated categorization. Cipollone’s pipeline processes the GitHub issues and labels them as either indicative of a vulnerability or not related. For a GitHub issue, if it was indicative of a vulnerability, a vulnerability description was generated. During experimentation, we isolate the true positives of Cipollone’s **LLM**-based classifier to evaluate the **RAG**-based automated categorization as illustrated in Figure 3.3. The false negatives were not considered, as the **LLM**-based classifier did not provide a vulnerability description for them. The **LLM**-based classifier was the most liberal in assigning vulnerability, meaning it minimized false negatives at the expense of precision. The combined classifier utilizing the XGBoost classifier had higher precision, but at the expense of some true positives. Therefore, to maximize true positives during retrieval experimentation, the combined classifier wasn’t attached until the very end of experimentation when the most accurate **RAG** solution had been identified. This final experiment would then provide the final overall performance of the entire combined pipeline.

3.3.1.1 Knowledge Base Creation and Optimization

While the raw JSON documentation could serve as a **CWE** knowledge base, it is cluttered with unnecessary symbols and repetitive information. On the other hand, when optimizing for retrieval, one wants to maximize the amount of useful identifiable information while minimizing the number of tokens. This minimizes the complexity, volume, and ambiguity of each document, reducing the risk of inaccurate or adverse retrieval. To this end, we converted the JSON documentation to the more readable markdown format and experimented with three separate levels of descriptive volume. [10, 9]

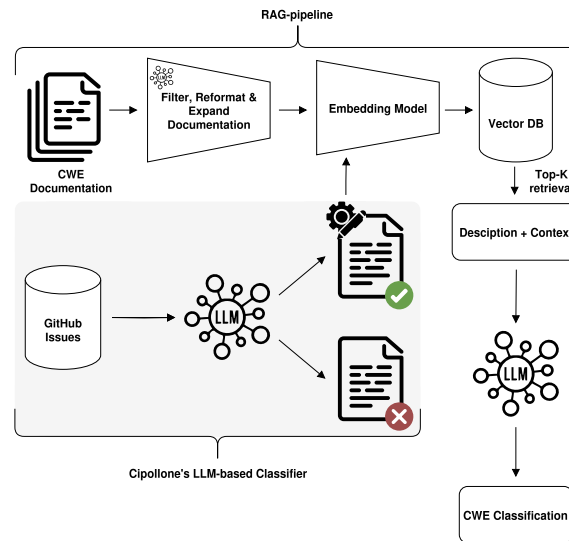


Figure 3.3: Automated vulnerability categorization pipeline used for RAG experimentation. A direct extension of Cipollone’s LLM-based classifier.

- **Light:** Containing only the most identifying fields, including the title (containing the CWE ID), description, extended description, and a demonstrative example illustrating how the weakness may look in actual code with an attached explanation of the context and location of the weakness.
- **Normal:** In addition to the fields present in the **Light** format, the following CWE fields are included.
 - *Alternate Terms* - indicating other names often used for the same weakness.
 - *Notes* - containing miscellaneous information providing additional comments about the weakness that are uncategorized.
 - *Observed Examples* - References to specific observed instances of the weakness containing a brief description of the weakness being cited.
- **Heavy:** In addition to the fields present in the **Normal** format, the following CWE fields are included.
 - *Common Consequences* - Potential negative consequences of the weakness if exploited.
 - *Affected Resources* - Different system resources that may be affected if the weakness gets exploited.
 - *Modes of introduction* - How and when the weakness may be introduced.
 - *Background Details* - Information that is not related to the nature of the weakness itself but is still relevant overall.

Additionally, multiple demonstrative examples may be included rather than just one.

The fields for each format were chosen based on their intuitive and apparent relevance to the information generally contained in the vulnerability descriptions generated at the end

of Cipollone's pipeline. Finally, the documents in these formats were split into smaller documents of at most 800 tokens to reduce volume and embedded into their own respective vector stores for experimentation. Each document was associated with its respective **CWE** via metadata. These formats were evaluated against each other in the initial couple of experiments. Afterwards, the format exhibiting the most accurate performance in retrieval was identified and exclusively utilized in the remaining experiments.

Finally, various embedding models were considered for vector knowledge store creation. We landed on the hugging face embedding model `BAAI/bge-large-en-v1.5` since it was free, runnable locally, relatively fast, and exhibited no worse retrieval performance than any other model tested, such as OpenAI's `text-embedding-3-large`. Additionally, the knowledge base and similarity search Python libraries `FAISS` and `ChromaDB` were used in combination with `LangChain`. `FAISS` was chosen for its speed and simplicity, while `ChromaDB` was required for more complicated filtering during **RAPTOR**.

3.3.1.2 Prompting

Cipollone utilized **LLMs** not only for automated vulnerability detection but also for exploiting its reasoning capabilities for nuanced vulnerability analysis. As such, Cipollone prompted the model to provide a JSON-formatted string containing the following fields.

- `gpt_description`: A detailed description of the vulnerability
- `gpt_confidence`: A confidence score from 1 to 5 intended to help evaluate the accuracy of the classification process
- `gpt_is_relevant`: A boolean indicating whether the GitHub issue is indicative of a Vulnerability or not

The exact prompt used is illustrated below in prompt 3.1 [4].

Prompt 3.1: Cipollone's system prompt

You are a cybersecurity assistant tasked with identifying potential vulnerabilities by analyzing GitHub issues. Your goal is to review each issue and determine whether it indicates a security vulnerability. Provide a detailed description of the issue and a confidence score of how much you are confident about the vulnerability.

In addition to identifying security vulnerabilities, you should also recognize cases where the issue is not a vulnerability. These may include failing tests, minor bugs, or issues related to functionality that do not present security risks.

Please format your response in JSON with the following fields. DO NOT add any additional text to the response:

`gpt_description`: Describe and reason about the issue. Explain if you detect any potential vulnerability or not.

`gpt_is_relevant`: A boolean indicating whether the issue is relevant (true) or not (false). This should be based on the explanation in description.

`gpt_confidence`: An integer from 1 to 5 indicating your level of confidence in the detection (1 = very low, 2 = low, 3 = medium, 4 = high, 5 = very high).

The first experiment simply used the `gpt_description` as the main retrieval query for related **CWE** documentation. However, the description generated by Cipollone for retrieval is vague, unpredictable, and overall not optimized for retrieval, especially not considering the **CWE** database. To adjust for this, another JSON field was proposed named `gpt_vulnerability` that describes the general type of vulnerability in a more broadly applicable sense, "the vulnerability class". This field is intentionally formatted in a similar way to the documents in our knowledge base (the **CWE** documentation). Thus, isolating the three most revealing parts of a regular **CWE** entry: the title, description, extended description, and one demonstrative example. These fields were chosen because they are all included in the light knowledge base documentation format, which was tested to perform the best on the `gpt_description` field. Additionally, limitations on the max number of available output tokens in the GPT-4o mini model (16384) [62] induce an incentive to create a maximally descriptive vulnerability description in as few necessary fields as possible. Multiple different prompt wordings were proposed and evaluated, and ultimately, prompt 3.2 resulted in the overall best performance.

Prompt 3.2: System prompt enhanced for CWE retrieval

You are a cybersecurity assistant tasked with identifying potential vulnerabilities by analyzing GitHub issues. Your goal is to review each issue and determine whether it indicates a security vulnerability. Provide a detailed description of the issue and a confidence score of how much you are confident about the vulnerability.

In addition to identifying security vulnerabilities, you should also recognize cases where the issue is not a vulnerability. These may include failing tests, minor bugs, or issues related to functionality that do not present security risks.

Please format your response in JSON with the following fields. DO NOT add any additional text to the response:

`gpt_description`: Describe and reason about the issue. Explain if you detect any potential vulnerability or not.

`gpt_vulnerability`: If a vulnerability was detected, describe the general type of vulnerability encountered. Please refrain from describing the vulnerability in any way specific to the particular instance of the vulnerability present in the GitHub issue at hand. Rather, describe the widely applicable class of vulnerability. Do this using the following general vulnerability sections in markdown format:

- # <short descriptive name for the general type of vulnerability detected>
- ## Description <general concise description of the type of vulnerability detected>
- ## Extended Description <optional general more thorough description of the type of vulnerability detected>
- ## Demonstrative Scenario <A description of an example scenario where this vulnerability is prevalent with the following subsections
 - ### Scenario <description of the example scenario>
 - #### Vulnerable <language> Code: <if relevant, add vulnerable example code in an appropriate language. This doesn't have to be code but could also be things like shell commands if that's more appropriate for the scenario. Make sure the code is in an appropriate markdown code block>
 - #### Analysis <Explanation for why the scenario is a vulnerability and what parts of the code in 'Vulnerable <language> Code' make it a vulnerable example and why>>

`gpt_is_relevant`: A boolean indicating whether the issue is relevant (true) or not (false). This should be based on the explanation in description.

`gpt_confidence`: An integer from 1 to 5 indicating your level of confidence in the detection (1 = very low, 2 = low, 3 = medium, 4 = high, 5 = very high).

It turns out that a large number of **CWE** entries in the original documentation were missing the field mentioned in both prompt 3.2 and subsection 3.3.1.1. In fact, some entries only contained a title and description, missing even an extended description or demonstrative examples. While this may be due to demonstrative examples not always being applicable, this affected a large portion of the documentation. It was reasonable to hypothesize that our

knowledge base might benefit from more consistency in the documentation. Furthermore, we had complete control over the structure and fields of our retrieval query as it was generated through prompt 3.2 in the `gpt_vulnerability` field. This provided us with a unique opportunity to align our knowledge base documentation with our queries, potentially increasing retrieval accuracy. As such, we introduced prompt 3.3 to complete the **CWE** entries with missing fields and align them with the markdown format of our queries.

Prompt 3.3: System prompt aligning CWE docs with prompt 3.2

You are a cybersecurity assistant responsible for completing vulnerability class descriptions. Each description follows a specific Markdown-based schema that outlines a widely applicable class of vulnerability. The schema is as follows:

1. '# <Short descriptive name for the general type of vulnerability detected>'
 2. '## Description'

General concise description of the type of vulnerability detected
 3. '## Extended Description'

General more thorough description of the type of vulnerability detected
 4. '## Demonstrative Scenario'

A description of an example scenario where this vulnerability is prevalent with the following subsections

 - '### Scenario'

Description of the example scenario
 - '#### Vulnerable <language> Code'

If applicable, vulnerable example code in an appropriate language. This doesn't have to be code but could also be things like shell commands if that's more appropriate for the scenario. Make sure the code is in an appropriate markdown code block
 - '#### Analysis'

Explanation for why the scenario is a vulnerability and what parts of the code in 'Vulnerable <language> Code' make it a vulnerable example and why

You will be provided with incomplete vulnerability descriptions that may be missing either the **Extended Description** and/or the **Demonstrative Scenario** sections. Your task is to complete **only** the missing sections, using the schema above and **Return the full vulnerability description**, with the missing section(s) added in place.

Finally, we combined the original query (vulnerability description generated in the `gpt_vulnerability` field in prompt 3.2) and the retrieved **CWE** documents using prompt 3.4 containing the following fields.

- `gpt_cwe`: JSON output field providing the predicted **CWE**.
- `gpt_cwe_confidence`: JSON output field providing a confidence score from 1 to 5 meant to help evaluate the accuracy of the categorization process

- `cwe_entries`: JSON input field where the retrieved **CWE** documents are returned but converted back to JSON format for structural consistency purposes.

Prompt 3.4: Final RAG prompt

You are a cybersecurity assistant tasked with labeling CWE (Common Weakness Enumeration) to given vulnerability descriptions. You will be provided with a number of complete CWE entries from the CWE database in JSON format. With this context in mind and drawing upon your own knowledge, your job is to decide which of these CWE entries best fits the vulnerability description and is the most appropriate for labeling.

Please format your response in JSON and include the following fields. ****Do not**** add any additional text to the response:

`gpt_cwe`: The CWE-ID (*only the number*) of the CWE entry that best fits the vulnerability description. If none of the CWE entries provided fit the vulnerability description, respond with "None"

`gpt_cwe_confidence`: An integer from 1 to 5 indicating your level of confidence in the labeling (1 = very low, 2 = low, 3 = medium, 4 = high, 5 = very high). The

following will be the provided JSON CWE entries:

```
{cwe_entries}
```

Lastly, for all prompts except the final **RAG** prompt, we used few-shot prompting with up to three examples.

3.3.1.3 RAG pipeline

When combining the retrieval candidates with the generated vulnerability description `gpt_vulnerability`, various methods of injecting documentation of the **CWE** candidates as context were experimented on and evaluated. We concluded that the inclusion of richer documentation for each **CWE** in the final generation steps led to better performance. Thus, for each **CWE** candidate, we provided all the original JSON documentation, including all fields present in the **Heavy** format, detailed in subsection 3.3.1.1.

Finally, the original vulnerability description generated by Cipollone's pipeline by prompt 3.1 in field `gpt_description` outperformed the broad description generated by prompt 3.2 in field `gpt_vulnerability`. We believe this was due to the broader description being optimized for retrieval, unlike the more concise and to-the-point description generated in `gpt_description`. For this reason, we chose to utilize `gpt_description` both in the baseline and final **RAG** generation step.

3.3.1.4 Fusion Retrieval

Fusion retrieval involves combining different retrieval strategies with different strengths and weaknesses to get the best of both worlds. In the case of this study, this would

entail combining the semantic understanding capabilities of vector-based retrieval with the keyword-based matching of BM25. This is simple to implement by computing the similarity scores of our query for both types of retrieval, normalizing them as in equation 3.1, and calculating a weighted combination of these scores as in equation 3.2.

$$\text{norm}(s_i^{\mathcal{R}}) = \frac{s_i^{\mathcal{R}} - \min_j s_j^{\mathcal{R}}}{\max_j s_j^{\mathcal{R}} - \min_j s_j^{\mathcal{R}} + \varepsilon} \quad \mathcal{R} \in \{\text{bm25}, \text{vec}\} \quad (3.1)$$

where $s_i^{\mathcal{R}}$ is the raw BM25 or vector search score (bm25 or vec) for document i and ε is a very small number, such as 10^{-8} , to avoid division by zero.

$$s_i^{\text{fusion}} = \alpha \cdot \text{norm}(s_i^{\text{vec}}) + (1 - \alpha) \cdot \text{norm}(s_i^{\text{bm25}}) \quad \alpha \in [0, 1] \quad (3.2)$$

where α is a hyperparameter describing the weight of the different retrieval strategies.

It is reasonable to assume that combining these forms of retrieval may improve the robustness and accuracy of retrieval. Thus, we included it for experimentation.

3.3.1.5 RAPTOR

So far, we've only discussed retrieval experiments that intuitively make sense given the data, knowledge base, and retrieval setup, which are straightforward to implement. Truthfully, there are countless well-established **RAG** techniques that may or may not lend themselves well to the use case at hand. However, due to limitations in time and resources, this work does not concern itself with exhaustive experimentation of **RAG** strategies. Rather, we leave that up to future research. However, **RAPTOR** seemed particularly suited for the tree-like structure of the **CWE** and was therefore chosen for our final experiment.

RAPTOR builds on the presupposition that detailed long texts often contain subtopics and a hierarchical structure. To address this, **RAPTOR** builds a recursive tree structure of the knowledge base, attempting to balance broader thematic comprehension and granular details within the texts. First **RAPTOR** segments the corpus into chunks of at most 100 tokens, making sure not to cut off mid-sentence and preserve contextual and semantic coherence. After embedding these text chunks using a **BERT**-based encoder, a clustering algorithm is used to group them. An **LLM** can then be used to generate summaries of each of these clusters, which can then again be clustered and summarized using the same process, recursively building a tree with each step. At the end, we get an abstraction tree where each node is a summary of a cluster of summaries, and the leaf nodes consist of the original documentation. [63]

The **RAPTOR** generated hierarchical abstraction tree seemed structurally similar to the relationship trees present in the **CWE**, illustrated in Figure 2.1. As such, through exploiting the inherent properties of the **CWE** documentation, we theorised that we'd be able to bypass the clustering and generative summary process of constructing the abstraction tree and fully focus on retrieval. **RAPTOR** proposes two retrieval strategies *tree traversal* and *collapsed tree*. Collapsed tree considers all nodes in the tree simultaneously by collapsing it and retrieving the top-K documents from there [63]. Under our **RAPTOR** inspired method, this would be equivalent to the regular retrieval we've had been doing so far,

since the tree-like structure has always been inherently present in the **CWE**. Tree traversal, on the other hand, involves traversing the tree from the top downwards by retrieving the top-K nodes at the current level and exclusively using their children for retrieval in the consecutive level until the bottom is reached. The final context would then consist of the union of the retrieval set from each level of the tree [63].

Looking at Table 3.1, we can see that the average number of children for any abstraction level was significantly smaller than the total number of valid **CWEs**. We hypothesized that tree traversal could significantly reduce the number of considered **CWE** for categorization at each level, decreasing the statistical chance of inaccurate retrieval. Since the number of children varied between abstraction levels, our **RAPTOR** inspired experiment could be adjusted to retrieve a different number of documents depending on the abstraction level retrieved from.

Abstraction	Count	Mean child count	Median child count	Max child count	Min child count
Pillar	10	17.90	12.50	41	4
Class	110	5.17	4.00	30	0
Base	521	0.58	0.00	15	0
Variant	292	0.09	0.00	5	0
Compound	7	0.00	0.00	0	0

(a) **CWE** View 1000

Abstraction	Count	Mean child count	Median child count	Max child count	Min child count
Pillar	2	2.50	2.50	5	0
Class	40	2.20	1.50	9	0
Base	81	0.00	0.00	0	0
Variant	5	0.00	0.00	0	0
Compound	2	0.00	0.00	0	0

(b) **CWE** View 1003

Table 3.1: Breakdown of the number of **CWEs** in different abstraction levels and statistics on the children of those **CWEs** for **CWE** view 1000 and 1003.

It is worth acknowledging that the paper introducing **RAPTOR** found that collapsed tree outperformed tree traversal [63]. However, they theorize this is due to the ratio of nodes from each level of the tree being constant, while a collapsed tree could freely retrieve the appropriate granularity for the given query. It is possible that the flexible retrieval strategy, dependent on the abstraction level mentioned above, partially addresses this. Furthermore, considering the highly specific nature of our use case, tree traversal couldn't be entirely ruled out without proper experimentation.

3.3.2 Baseline Experiment

The baseline was meant to represent the most naive **LLM** based solution to automated **CWE** categorization. To this end, it simply provided the entire **CWE** in JSON format to the **LLM**, along with the vulnerability description to be identified and categorized. Because the larger **CWE** View 1000 is around 2.5 million tokens and the maximum context window for **GPT-4o mini** is 128000 tokens, this had to be done in chunks. In other

words, we would split the view into sections of 100000 tokens, making sure to always split in between **CWE** entries so as not to fragment them. Afterwards, we would ask the **LLMs** to identify which, if any, of the **CWEs** in each section aligned with the provided vulnerability description, one prompt at a time. The union of the **CWEs** identified from each section would then once again be given to the **LLM** in a final prompt to isolate the single correct categorization. We used zero-shot prompting consistent with how we did final **RAG** prompting. Additionally, the prompt used was a slightly adjusted version of prompt 3.4, prompt 3.5. We hypothesized that the baseline might exhibit higher accuracy but at a much lower efficiency, resulting in much longer wait times and substantially higher financial costs.

Prompt 3.5: Baseline prompt

You are a cybersecurity assistant tasked with labeling CWE (Common Weakness Enumeration) to given vulnerability descriptions. You will be provided with a number of complete CWE entries from the CWE database in JSON format. With this context in mind and drawing upon your own knowledge, your job is to decide which of these CWE entries best fits the vulnerability description and is the most appropriate for labeling. You should also recognize when none of the CWE entries provided fit the vulnerability description and are all inappropriate for labeling.

Please format your response in JSON ****without code blocks**** so that the response string can be directly converted to a dictionary using python function 'json.loads'. Include the following fields. ****Do not**** add any additional text to the response:

`gpt_cwe`: The CWE-ID (***only the number***) of the CWE entry that best fits the vulnerability description. If none of the CWE entries provided fit the vulnerability description, respond with "None"

`gpt_cwe_confidence`: An integer from 1 to 5 indicating your level of confidence in the labeling (1 = very low, 2 = low, 3 = medium, 4 = high, 5 = very high).

The following will be the provided JSON CWE entries:

```
{cwe_entries}
```

3.4 Evaluation framework

As partially explained in subsection 3.3.1, most of the experiments concerned themselves only with retrieval. Saving the final step of **RAG**, generation for the very end. This is because the performance of the raw retriever would be the primary bottleneck for the final generation step. The final step combined the **CWE** candidates with the original vulnerability description, introducing reasoning through the use of an **LLM** to isolate the correct candidate. Thus, if the correct **CWE** was not present in any of the **CWE** candidates, it was likely that the final generation step would also misidentify the vulnerability description. As such, the first few experiments would only be evaluated on their top-K retrieval performance where $K \in \{1, 5, 10, 15, 20\}$. When measuring top-K performance,

we considered a sample correctly categorized if any of the top-K retrieved categories was the correct one. We did not consider the placement of this guess during evaluation, only that it was present within the top-K retrieved categories. The **RAPTOR** inspired retriever, however, had to be evaluated on the average number of retrieved documents, as the number of documents it retrieved depended on the path it took in the **CWE** graph. Finally, once the best retrieval strategy had been identified, the full **RAG** pipeline was evaluated against the baseline in isolating the single correct **CWE** classification.

We will evaluate our experiments on not only the complete **CWE** view 1000 but also the **NVD**'s **CWE** view 1003. Indeed, around 98% of our positive **CWE** samples were present in **CWE** view 1003. Still, this work aims to evaluate how well **RAG** can generalize on all possible **CWE** despite the large number of categories and sparse sample data. Hence, both views were valuable for experimentation and evaluation.

3.4.1 Evaluation Data

When evaluating our retrieval strategies, baseline and **RAG**-pipeline, the sample test data only consisted of the true positives from Cipollone **LLM**-based classifier. As explained in subsection 3.3.1, only the true positives and false positives would have associated vulnerability descriptions that could be used for retrieval. We already knew, however, that the false positives would be classified incorrectly, as they had already been mislabeled as vulnerabilities by Cipollone's pipeline. As such, they were deemed irrelevant for the separate evaluation of the **CWE** classifiers, including the **RAG** extension to Cipollone's pipeline and the baseline. The same logic was used when considering the false negatives. As such, only when evaluating the entire combined Cipollone and **RAG** pipelines did we include all the original test data.

Due to the large cost of running the baseline, it had to be tested on a smaller subset of the sample data. We settled on a smaller test sample containing 296 true positive samples representing 44 **CWE** classes. This smaller test sample consists of GitHub issues from the top 50 repositories since 2023, following Cipollone's data collection methodology (see subsection 2.4.1.2). For efficiency, we also utilized this sample in the initial experiments to isolate the best retrieval strategy. We then use our largest sample of true positives to get a more representative performance evaluation of said strategy alongside the full **RAG** pipeline. Finally, when evaluating the full combined pipeline, we were forced to use a sample in between (a medium-sized sample), as we needed to allocate 60% of the larger sample for training the XGBoost classifier in accordance with Cipollone's original pipeline. For completeness and comparability, we evaluated the performance of the smaller sample on all experiments. Further details on the samples can be found in Table 3.2 and Figure 3.4.

As indicated in Table 3.2, a small portion of the positive samples and thus **CWEs** were lost to the false negatives of the language model portion of Cipollone's pipeline. Regardless, we reasoned that as long as the number of samples is sufficient, the number of **CWEs** present should not significantly impact the reliability of our evaluation. Our model was not trained or finetuned on these categories; rather, it used pure **RAG** and thus should have minimal bias towards any particular **CWE**, relying solely on retrieval and embedded

Sample Size	# Positive	# Negative	# CWEs	Samples in View 1003 (%)	CWEs in View 1003 (%)
Small	306	1457	45	98.04	86.67
Medium	1364	5055	114	98.31	88.60
Large	3410	12620	152	98.12	78.95

(a) Full data statistics

Sample Size	# Samples	# CWEs	Samples in View 1003 (%)	CWEs in View 1003 (%)
Small	296	44	97.97	86.36
Medium	1264	109	98.42	89.91
Large	3147	148	98.16	79.73

(b) True positive sample statistics

Table 3.2: Breakdown of the structure and important aspects of our test data samples.

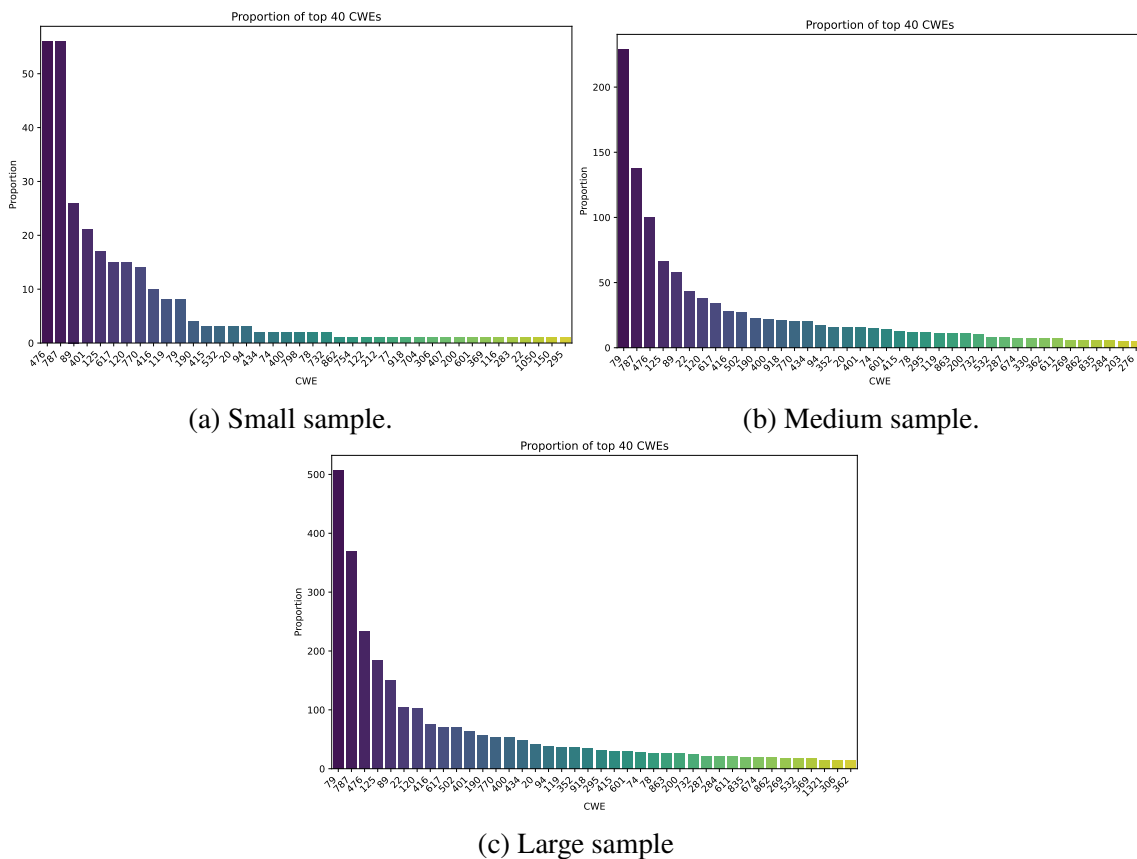


Figure 3.4: Distribution of the top **CWEs** for each true positive sample.

knowledge. Furthermore, the tendency of Cipollone’s pipeline to overestimate the number of positive samples kept the false negatives at a minimum.

Figure 3.4 illustrates the distribution of **CWE** used in categorization and clearly highlights its logarithmic shape and the resulting sparsity of the data.

3.4.2 Performance Metrics

This work utilized standard performance metrics in machine learning tasks for evaluation. First and foremost, we evaluated overall accuracy. However, precision, recall, and F1-score were also of interest, in particular, in relation to their weighted average. The weighted average obtains the average metric value for each class when accounting for the support of each class and thus class imbalances as detailed in equation 3.3. The sheer number of classes and highly imbalanced nature of **CWE** categorization reflected in the available data necessitated the use of the weighted average.

$$\frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i} \quad (3.3)$$

where w_i is the support (number of samples) of the class i and x_i is the metric value of class i .

To explain the different evaluation metrics, we first need to establish the following terms: TP = "True Positives", TN = "True Negatives", FP = "False Positives", FN = "False Negatives". Here, the "positives" in relation to a particular **CWE** would consist of every GitHub issue indicative of a vulnerability categorized under said **CWE**, and the "negatives" would be the opposite. The performance metrics and their utility can then be described as follows.

- *Precision*: The proportion of all positive classifications that were actually positive, as illustrated in equation 3.4. In our case, positive classifications of particular **CWEs**.

$$\text{Precision} = \frac{TP}{TP + FP} \quad (3.4)$$

- *Recall*: The proportion of positive samples that were correctly classified as positive, as illustrated in equation 3.5.

$$\text{Recall} = \frac{TP}{TP + FN} \quad (3.5)$$

- *F1-Score*: Precision and recall come at the cost of one another. More precision entails a harsher critic that risks doubting actual positive samples, while high recall implies a more lax strategy that has a higher risk of missing negative samples. As such, the F1-score aims to combine the two using their harmonic mean, as in equation 3.6. Thus, maximizing F1-score means maximizing both precision and recall simultaneously, combining the strengths of both evaluation metrics.

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (3.6)$$

- *Accuracy*: Finally, overall accuracy is calculated as the proportion of correct

predictions among the total predictions as in equation 3.7.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (3.7)$$

Chapter 4

Results and Analysis

This chapter presents and provides an in-depth analysis of the results obtained from our various retrieval and categorization methodologies. The analysis will focus on comparisons between the different experiments conducted, the reliability of the results, computational and financial costs, and comparisons with previous research and **SOTA** results. Such an examination is meant to provide a comprehensive overview of the efficiency, effectiveness, and practical implications of our solution.

We will use the terms "precision", "recall", and "F1-score" as shorthand for their respective weighted averages for the sake of brevity. For a detailed description of the performance metrics and their respective weighted averages, check subsection **3.4.2**. Finally, for repeatability, it is worth mentioning that results would vary around $\pm 3\%$ due to **LLM** and retrieval randomness, especially during smaller sample evaluation. Nevertheless, the results would never be significantly worse than those presented here.

4.1 Baseline

Our baseline results are illustrated in Table **4.1** and are surprisingly worse than the results of our proposed solution contained in Table **4.13**. It is also much more expensive to run, especially for **CWE** view 1000, at around \$55 as opposed to less than one dollar, and takes at worst 24 hours to run as opposed to around 15 seconds. Interestingly enough, switching to view 1003 only made the results worse for the baseline, as opposed to the **RAG** solution. We believe this may be due to there being some samples not present in view 1003, and the baseline considering too many classes at once for the decrease of around 900 classes to 130 to have an effect. Finally, these results are not impressive enough to justify baseline testing of one of the larger samples.

4.2 Retrieval

This section will present the evaluation results for the various retrieval strategies explored for the final **RAG** pipeline. Unless stated otherwise, these results were generated using the "small" true positive data sample described in subsection **3.4.1**

CWE View	Accuracy	Precision	Recall	F1-Score
1000	0.4628	0.6278	0.4628	0.4885
1003	0.4358	0.5951	0.4358	0.4628

Table 4.1: Baseline results on view 1003 and view 1000 on the "small" true positive data sample.

We can see in Table 4.4 that retrieving on the vulnerability description generated by Cipollone, a relatively unstructured and vague `gpt_description` prompt leads to relatively inaccurate retrieval. Even so, when considering the large number of available classes, lack of fine-tuning and training, we can already see potential, especially when only considering View **CWE-1003**, which achieves up to 85% accuracy. Indeed, previous **SOTA** such as Das *et al.* only achieved 14.9%, 44.0%, and 67.5% top-K accuracy for k equal to 1, 6, and 20, respectively, when considering results on **CWEs** not present in their training data.

Clearly, utilizing view 1003 improves performance significantly. We also notice that the light documentation performs the worst on view 1000, but the best for view 1003, which may be due to the density of information in the **CWEs** specifically chosen for categorization within the **NVD**, and that they may have fewer missing fields.

k	Class	Accuracy	Precision	Recall	F1-score
1	Light	0.1926	0.4640	0.1926	0.2303
	Normal	0.1520	0.5038	0.1520	0.1733
	Heavy	0.1993	0.5566	0.1993	0.2320
5	Light	0.3986	0.8293	0.3986	0.4674
	Normal	0.4527	0.8464	0.4527	0.5077
	Heavy	0.4595	0.8017	0.4595	0.4965
10	Light	0.4730	0.8392	0.4730	0.5289
	Normal	0.5304	0.8824	0.5304	0.6011
	Heavy	0.5405	0.8973	0.5405	0.5875
15	Light	0.5372	0.8423	0.5372	0.5816
	Normal	0.5946	0.9201	0.5946	0.6724
	Heavy	0.5845	0.9022	0.5845	0.6393
20	Light	0.5777	0.8851	0.5777	0.6507
	Normal	0.6419	0.9182	0.6419	0.7089
	Heavy	0.6284	0.9253	0.6284	0.6940

(a) View CWE-1000

k	Class	Accuracy	Precision	Recall	F1-score
1	Light	0.3547	0.5740	0.3547	0.3817
	Normal	0.2601	0.6297	0.2601	0.2889
	Heavy	0.3209	0.6200	0.3209	0.3551
5	Light	0.5676	0.8770	0.5676	0.6308
	Normal	0.5541	0.8376	0.5541	0.6162
	Heavy	0.5709	0.8556	0.5709	0.5994
10	Light	0.7162	0.9150	0.7162	0.7865
	Normal	0.7365	0.9163	0.7365	0.7906
	Heavy	0.7230	0.8827	0.7230	0.7623
15	Light	0.8243	0.9144	0.8243	0.8615
	Normal	0.7872	0.9403	0.7872	0.8337
	Heavy	0.7905	0.9214	0.7905	0.8238
20	Light	0.8581	0.9396	0.8581	0.8933
	Normal	0.8243	0.9480	0.8243	0.8659
	Heavy	0.8243	0.9380	0.8243	0.8580

(a) View CWE-1003

Table 4.4: Results of the first, basic retrieval experiment utilizing the vulnerability description generated by prompt 3.1 in the field `gpt_description` at the end of Cipollone’s pipeline as the query. Three different knowledge bases are evaluated using different levels of descriptiveness (classes) regarding their documentation, detailed in subsection 3.3.1.1. Additionally, we evaluate the top-K performance metrics for five different increasing values of K as described in section 3.4. We notice a maximum accuracy of around 85% for view 1003. The best results for each K are highlighted.

Utilizing a more structured and retrieval-specialized vulnerability description prompt significantly improves overall performance, as evident when comparing Table 4.4 and Table 4.7. We also see that the light documentation is exhibiting much better relative performance. This isn’t surprising considering that our adjusted vulnerability description is formatted according to the light documentation. We thus conclude that aligning the generated vulnerability description with the documentation improves retrieval performance.

k	Class	Accuracy	Precision	Recall	F1-score
1	Light	0.4324	0.6469	0.4324	0.4619
	Normal	0.3007	0.4904	0.3007	0.3111
	Heavy	0.3209	0.6797	0.3209	0.3334
5	Light	0.5709	0.8369	0.5709	0.5854
	Normal	0.5439	0.8627	0.5439	0.5866
	Heavy	0.5439	0.8438	0.5439	0.5766
10	Light	0.6047	0.9049	0.6047	0.6425
	Normal	0.5980	0.9329	0.5980	0.6478
	Heavy	0.6081	0.9386	0.6081	0.6547
15	Light	0.6385	0.9213	0.6385	0.6861
	Normal	0.6318	0.9544	0.6318	0.6846
	Heavy	0.6385	0.9662	0.6385	0.6877
20	Light	0.6858	0.9183	0.6858	0.7440
	Normal	0.6757	0.9544	0.6757	0.7415
	Heavy	0.6622	0.9730	0.6622	0.7105

(a) View CWE-1000

k	Class	Accuracy	Precision	Recall	F1-score
1	Light	0.5000	0.6244	0.5000	0.4884
	Normal	0.5000	0.6868	0.5000	0.5062
	Heavy	0.5101	0.6639	0.5101	0.5018
5	Light	0.6419	0.8963	0.6419	0.6787
	Normal	0.6453	0.8998	0.6453	0.6986
	Heavy	0.6149	0.9048	0.6149	0.6372
10	Light	0.8209	0.9074	0.8209	0.8535
	Normal	0.8277	0.9360	0.8277	0.8704
	Heavy	0.6858	0.9501	0.6858	0.7448
15	Light	0.8581	0.9228	0.8581	0.8835
	Normal	0.8649	0.9521	0.8649	0.9017
	Heavy	0.8142	0.9552	0.8142	0.8691
20	Light	0.8818	0.9184	0.8818	0.8938
	Normal	0.8851	0.9487	0.8851	0.9125
	Heavy	0.8784	0.9490	0.8784	0.9090

(a) View CWE-1003

Table 4.7: Results of the second retrieval experiment, where the prompt at the end of Cipollone’s pipeline is adjusting for retrieval, resulting in prompt 3.2. The enhanced vulnerability description generated in the field `gpt_vulnerability` is then used as the query. Three different knowledge bases are evaluated using different levels of descriptiveness for their documentation, detailed in subsection 3.3.1.1. Additionally, we evaluate the top-K performance metrics for five different increasing values of K as described in section 3.4. The best results for each K are highlighted.

The results exhibited in Table 4.10 illustrate further how aligning the queries with the documentation improves performance. When comparing to Table 4.7, we see significant improvement, especially for lower values of K. Recall and Precision are also beginning to approach each other, though precision is consistently higher, indicating a tendency to misidentify vulnerabilities more often than overestimating or favoring a certain class. We also evaluate the performance on the larger dataset for a more representative evaluation. Here we observe a margin of error of between 4 and 5 percent, confirming the reliability of the previous results. Looking back at Table 4.4, performance has greatly increased without any fine-tuning or training. Indeed, when considering only view 1003, these results are comparable to Das *et al.* even when considering their results on **CWEs** present in their training data.

k	Sample size	Accuracy	Precision	Recall	F1-score
1	Small	0.4122	0.5885	0.4122	0.4266
	Large	0.4480	0.5836	0.4480	0.4624
5	Small	0.6453	0.8431	0.6453	0.7052
	Large	0.6079	0.8431	0.6079	0.6677
10	Small	0.7500	0.9058	0.7500	0.7933
	Large	0.6870	0.8732	0.6870	0.7368
15	Small	0.7872	0.9298	0.7872	0.8359
	Large	0.7318	0.9352	0.7318	0.7855
20	Small	0.8243	0.9422	0.8243	0.8714
	Large	0.7703	0.9357	0.7703	0.8133

(a) View CWE-1000

k	Sample size	Accuracy	Precision	Recall	F1-score
1	Small	0.5507	0.6703	0.5507	0.5523
	Large	0.5558	0.6372	0.5558	0.5556
5	Small	0.8074	0.8867	0.8074	0.8312
	Large	0.7718	0.8472	0.7718	0.7857
10	Small	0.8514	0.8914	0.8514	0.8644
	Large	0.8411	0.9040	0.8411	0.8508
15	Small	0.8953	0.9042	0.8953	0.8948
	Large	0.8700	0.9311	0.8700	0.8827
20	Small	0.9223	0.9411	0.9223	0.9261
	Large	0.8881	0.9528	0.8881	0.9060

(a) View CWE-1003

Table 4.10: Results of the third retrieval experiment, where the prompt at the end of Cipollone’s pipeline is adjusting for retrieval, resulting in prompt 3.2. The enhanced vulnerability description generated in the field `gpt_vulnerability` is then used as the query. Additionally, an expanded version of the light documentation is utilized as the knowledge base. The light documentation is expanded using LLMs and prompt 3.3 to complete missing fields as described in subsection 3.3.1.2. The top-K performance metrics for five different increasing values of K are evaluated as described in section 3.4. Evaluation is conducted for both the small and large sample sizes of true positive data samples described in subsection 3.4.1.

4.2.1 RAPTOR and Fusion

Various values for alpha and setups for K depending on abstraction levels were experimented on for fusion and RAPTOR, respectively, but, for all cases, performance was anywhere from 5 to 20 percent worse than the results exhibited in Table 4.10. Often with up to double the number of retrieved documents in the case of RAPTOR and at least twice as long a runtime. As such, these two strategies were disregarded as not viable for this particular problem. In the case of RAPTOR, this seems to align with the results of Sarthi *et al.* [63], where the collapsed tree structure had overall better performance than tree traversal. As explained in subsection 3.3.1.5, the regular retrieval strategy in our particular application can be seen as a collapsed tree structure.

4.3 Major results

In the following results, ”k” refers to how many categories are retrieved during RAG retrieval before being isolated into a single correct or incorrect classification in the final RAG generation step. In other words, all the following results concern themselves with comparing the top-1 performance metrics when retrieving different numbers of vulnerability candidates on different sample sizes.

As we can see in Table 4.13 we achieve marginally better results than the baseline using are RAG pipeline. Though this isn’t much of an improvement performance-wise, in terms of computational, financial, and environmental cost, it’s a minuscule fraction of the baseline. Evidently, LLMs models that have not been fine-tuned are not great at vulnerability detection, and although our retrieval showed large amounts of potential, the bottleneck

seems to be the generation step. Indeed when looking at the best performance on the large true positive sample and comparing it with the accuracy of the respective retrieval strategy of $K = 20$ in Table 4.10 GPT-4o mini is only able to isolate the correct **CWE** label $100 \cdot \frac{0.4938}{0.7703} \approx 64.10\%$ of the time. Looking at the same $K = 20$ accuracies 1003 we see a similar **CWE** isolation accuracy of $100 \cdot \frac{0.5895}{0.8881} \approx 66.38\%$. Thus, indicating that it is the retrieval step performance specifically that is improved for view 1003, in line with all our experimental retrieval results. This partly explains the contrast between the results on views 1000 and 1003 in Table 4.13. For view 1003, retrieving fewer candidates is better since the fewer candidates, the smaller the bottleneck from the generation (**LLM**) step, and retrieval performance is already high at smaller numbers of K . For view 1000, however, pure retrieval performance is the bottleneck at smaller values of K , and so more candidates have to be retrieved for marginal improvements in accuracy.

k	Sample size	Accuracy	Precision	Recall	F1-score
3	Small	0.4764	0.6805	0.4764	0.5063
	Large	0.4779	0.6339	0.4779	0.5074
5	Small	0.4865	0.6355	0.4865	0.5060
	Large	0.4827	0.6681	0.4827	0.5093
10	Small	0.4831	0.6992	0.4831	0.5079
	Large	0.4881	0.6649	0.4881	0.5187
15	Small	0.4831	0.7074	0.4831	0.5100
	Large	0.4909	0.6567	0.4909	0.5169
20	Small	0.4932	0.6677	0.4932	0.5115
	Large	0.4938	0.6697	0.4938	0.5161

(a) View **CWE**-1000

k	Sample size	Accuracy	Precision	Recall	F1-score
3	Small	0.6115	0.7011	0.6115	0.6208
	Large	0.6225	0.6688	0.6225	0.6255
5	Small	0.6047	0.7387	0.6047	0.6266
	Large	0.6012	0.6791	0.6012	0.6141
10	Small	0.5912	0.6955	0.5912	0.6058
	Large	0.5904	0.6675	0.5904	0.6047
15	Small	0.5709	0.6765	0.5709	0.5834
	Large	0.5917	0.6570	0.5917	0.6004
20	Small	0.5811	0.6517	0.5811	0.5850
	Large	0.5895	0.6583	0.5895	0.5983

(a) View **CWE**-1003

Table 4.13: Results of the **RAG** pipeline experiment where the **CWE** candidates determined by the output of experiment three are combined with the generated vulnerability description using prompt 3.4 as described in subsection 3.3.1.3. Performance metrics are measured for five different values of K . Here, K represents how many **CWE** candidates we retrieve to choose from when isolating the correct category to get the top-1 answer in the final generation step of the **RAG** pipeline. Both the small and large sample sizes of true positives described in subsection 3.4.1 are evaluated. The best results for the large sample size are highlighted.

Finally, for completeness, we include the results of running the full combined Cipollone and **RAG** pipeline together using all the available testing data and counting the fraction of correct categorization in Table 4.16. Here, the results look very promising, but we have to remember that a large portion of the correct classifications are negative vulnerabilities, meaning we can attribute most of the accuracy to Cipollone’s model filtering out negative examples. Even so, the retrieval itself exhibits a lot of potential, and when considering the circumstances of our data mentioned previously, an $K = 1$ accuracy of around 50% in Table 4.13 is unheard of in previous research. That stated, when considering view 1003 previous state of the art, ThreatZoom does exhibit significantly better performance. Though it must be acknowledged that ThreatZoom requires a substantial amount of training and lacks precision as it isn’t based on pinpointing the exact **CWE** as with our model. However, Kota *et al.* solution, which does pinpoint the exact **CWE**, does also outperform us, exhibiting an accuracy of 72.1% as opposed to 62.25%. Regardless, the retrieval itself,

without the generation step, exhibits unheard-of performance on higher values of K and shows significant potential for possible future integration in more sophisticated solutions and pipelines. .

k	Sample size	Accuracy	Precision	Recall	F1-score
3	Small	0.8145	0.8638	0.8145	0.8337
	Medium	0.8450	0.8744	0.8450	0.8528
5	Small	0.8151	0.8626	0.8151	0.8341
	Medium	0.8453	0.8693	0.8453	0.8521
10	Small	0.8174	0.8655	0.8174	0.8369
	Medium	0.8439	0.8663	0.8439	0.8509
15	Small	0.8162	0.8670	0.8162	0.8360
	Medium	0.8445	0.8769	0.8445	0.8520
20	Small	0.8140	0.8587	0.8140	0.8328
	Medium	0.8448	0.8779	0.8448	0.8525

(a) View CWE-1000

k	Sample size	Accuracy	Precision	Recall	F1-score
3	Small	0.8361	0.8694	0.8361	0.8488
	Medium	0.8696	0.8796	0.8696	0.8718
5	Small	0.8395	0.8734	0.8395	0.8527
	Medium	0.8702	0.8813	0.8702	0.8730
10	Small	0.8395	0.8727	0.8395	0.8514
	Medium	0.8727	0.8816	0.8727	0.8734
15	Small	0.8383	0.8673	0.8383	0.8490
	Medium	0.8740	0.8819	0.8740	0.8738
20	Small	0.8372	0.8664	0.8372	0.8478
	Medium	0.8704	0.8805	0.8704	0.8716

(a) View CWE-1003

Table 4.16: Results of the complete combined pipeline experiment where Cipollone’s pipeline for vulnerability identification is combined with the RAG pipeline, where the detected vulnerabilities are categorized according to prompt 3.4. Performance metrics are measured for five different values of K. Here, K represents how many CWE candidates we retrieve to choose from when isolating the correct category to get the top-1 answer in the final generation step of the RAG pipeline. Both the small and medium sample sizes of true positives described in subsection 3.4.1 are evaluated. The complete test data, not just the true positives, are used for evaluation, as we’re testing the complete pipeline, not just categorization. The best results for the medium-sized sample size are highlighted.

Chapter 5

Discussion

In this chapter, we provide a more theoretical discussion of the methods and results, reflecting on the limitations, practical implications, and potential for future work inherent in them.

5.1 Limitations

Due to the costly nature of the baseline solution and its heavy reliance on OpenAI's **LLMs**, this work required the use of a low-cost language model, hence the utilization of `GPT-4o mini`. Furthermore, the use of `GPT-4o mini` ensured consistency with Cipollone's research and aligned with the goal of an economically efficient solution. However, `GPT-4o mini` is limited to a maximum of 16384 output tokens, and the OpenAI API would often seemingly glitch when using this model and return an error, even when the reported number of output tokens stayed below this threshold. This drastically limited the possible length and, therefore, complexity of the viable prompts, limiting prompt engineering. This was one of the reasons why prompt 3.3 focuses on completing the light documentation, as any further complexity would risk resulting in too many output tokens. Furthermore, the use of this model may have generally bottlenecked performance.

Additionally, the costly nature of the baseline model limited the number of samples and variety of **CWE** it could be tested on. However, considering the consistency of results across different sizes of testing samples, we still believe we got a good estimate of its true performance within a small margin of error. Regardless, the baseline is clearly prohibitively expensive and, at the very least, no better than our proposed **RAG** solution in automated **CWE** categorization.

There are a lot of inconsistencies in the **NVD** dataset in regards to **CWE** labeling as explained in subsection 3.2.2. **CNAs** often disagree with each other and some even used prohibited **CWE** labels. This points to the element of human error playing a factor in our testing data and ground truth. Given this, it may be possible that for some samples our solution determines a more appropriate categorization than the ground truth. Unfortunately, we did not have access to the security professionals and resources required to determine instances of this or how often it may occur.

Finally, given the novel and experimental nature of **LLMs** and the enormous array of potentially viable **RAG** strategies, this work is by no means exhaustive. Many potential adjustments and tweaks, from better prompt engineering and more efficient retrieval to fine-tuning and improved **ICL**, could potentially drastically improve performance. Nevertheless, the goal of this study is to explore the potential of employing **RAG** for automated vulnerability categorization. Thus, this study primarily concerns itself with researching the viability of basic **RAG** principles, leaving the objective of optimization to future work.

5.2 Reflections

The results can be interpreted from a variety of perspectives when considering previous research on automated **CWE** categorization. On the one hand, when considering the unmatched sheer number of available **CWE** classes, our model considers a $K = 1$ accuracy of around 50%, which is evidently impressive. On the other hand, a $K = 1$ accuracy of around 60% when only considering **CWE** view 1003 is a far cry from ThreatZoom's 92% accuracy or even Kota *et al.* 72% accuracy on more precise categorization as highlighted in section 4.3. Then again, our solution employs no training or fine-tuning, unlike all previous research.

Additionally, there's the performance of the dense embedding-based retrieval itself to consider. As exemplified in section 4.2, dense retrieval with **LLM** enhanced documentation is competitive with previous research in top-K categorization, not only when considering the unmatched number of **CWE** mentioned earlier but especially when focusing on view 1003. Once again, no training or fine-tuning is required, except for the enhancement of the documentation and vulnerability description generation. Thus, this leads to a very computationally efficient and therefore also environmentally friendly solution as opposed to previous **SOTA** like ThreatZoom.

Furthermore, even though the $K = 1$ performance may not be ideal, narrowing the number of potential **CWE** to 5, 10, 15, or even 20 options in a sea of over 800 possible categories can still serve as a useful tool for facilitating vulnerability categorization even without full automation. In a practical sense, there may also be cases where our solution may not provide the most appropriate **CWE** category but may still point a user in the direction of the true **CWE** when considering the tree-like structure of the documentation. As Albanese *et al.* points out, the similarity of various **CWE** entries may impact accuracy, and an incorrect categorization is likely at the very least adjacent to the true **CWE**.

The impressive results despite the lack of any complicated **RAG** strategies, fine-tuning, and training further demonstrate the potential for future, more sophisticated and optimized **RAG** utilization for automated **CWE** categorization. The bare nature of our solution lends itself well to future expansion.

Regardless, what seems clear when looking at the result is that adept embedding-based dense retrieval on its own made the most significant contribution to the accuracy of the automated **CWE** labeling. On the other hand, the final generation step utilizing **LLM** only resulted in marginal improvements. This ultimately begs the question of whether there are

other **ML** methods that could more effectively utilize the top-K accuracy of our retrieval methods for definitive vulnerability categorization.

5.3 Future Work

As repeatedly stated in earlier sections, the goal of this study was to evaluate the viability of knowledge retrieval for automated **CWE** categorization in an effort to extend Cipollone's vulnerability classification pipeline and complete the loop. As such, multiple potential optimizations were left on the table for future research. There's also potential future research specific to Cipollone's pipeline that won't be mentioned here but is available in his paper.

5.3.1 RAG Optimization

There are an endless number of possibilities for **RAG** potential optimization utilizing various **RAG** strategies. Some notable examples include various GraphRAG strategies, where documentation is processed to create a rich knowledge graph used to enhance retrieval; Dartboard Retrieval, which uses a combined relevance-diversity scoring function to reduce redundancy in retrieval; and hierarchical indices, which use document encoding, summaries, and detailed chunks in a hierarchical indexing system to improve efficiency and relevance. There are also various fusion algorithms, not experimented with here, which may have been more appropriate than a simple weighted combination.

Looking back on our **RAPTOR** solution, perhaps generating our own hierarchical tree through summary generation and clustering rather than relying on the inherent structure of the **CWE** would have improved performance. Even though the **CWE** is structured into a tree-like structure with different levels of abstraction, it is far from a perfect tree. Furthermore, it is possible that the content of a particular **CWE** entry does not serve as an ideal summary for its children. Even though a **CWE** of a higher abstraction is simply a less specific description of the vulnerabilities below it, the goal of its documentation may not exactly be to serve as a perfect summary for the vulnerabilities below it.

5.3.2 Prompt Engineering

As demonstrated throughout this study, our vulnerability categorization pipeline is heavily reliant on **LLMs** for not only vulnerability description but also documentation completion. As such, the quality and clarity of the prompts are gonna have a significant impact on performance. As discussed in section 5.1, we were limited in the viable prompt complexity, but integrating official MITRE instructions for the structure of **CWE** entries or root cause mapping for final **CWE** mapping at the end of the **RAG** pipeline may help align the **LLMs** with security experts. Few-shot prompting at the end of the **RAG** pipeline may also be helpful.

Additionally, in the case of **RAPTOR**, we noticed that the cause for its poor performance was its tendency to misclassify the correct **CWEs** at the higher levels of abstraction. We theorize that this could be solved by generating separate vulnerability descriptions with

increasing levels of generality for each **CWE** abstraction level. Then, during **RAPTOR** tree traversal, at each level, the description congruent with the said abstraction level could be used for retrieval.

5.3.3 Training and Fine-tuning

Since the justification for using **RAG** for a **CWE** categorization was to minimize dependence on the available vulnerability data due to the inherent sparsity of said data, we chose not to experiment with fine-tuning. Regardless, both the vulnerability description task and **RAG** itself can be fine-tuned for the task at hand. Whilst this may increase dependence on the data, it can only improve performance, likely by a significant margin. Indeed, when considering previous research, we hypothesize that fine-tuning the **LLMs** and **RAG** used may be the final step needed to surpass categorization performance with previous **SOTA** such as ThreatZoom. Of course, there is the issue of the limited available data, but considering the impressive performance achieved without any fine-tuning and limited **ICL**, we believe this hurdle can be overcome. With more direct access to the generative model, one could also consider any of the training methodologies mentioned in subsection 2.3.7.4.

5.3.4 Evaluation

Considering the inherent structure of the original **CWE** documentation in our evaluation may also have provided a clearer view into the accuracy and inherent usefulness of our solution. For example, one could weigh the accuracy of a categorization depending on its relation to the correct labeling in the abstraction tree, considering their distance from each other and their type of relationship. Offsetting the similarities between **CWEs** by adjusting for the semantic similarity between them, following the approach of Albanese *et al.*, should also be considered in future work when designing the evaluation framework. Due to limitations in resources and time, and failure to recognize this early on, this thesis unfortunately only employs a relatively basic evaluation framework in comparison.

Chapter 6

Conclusions

This study aimed to assess the feasibility, efficiency, and potential of utilizing **RAG** and more broadly, knowledge retrieval, in supporting or automating vulnerability categorization using the **CWE**. The results confirm the feasibility of this approach and the significant potential of retrieval in facilitating vulnerability categorization. Notably, this is achievable with extremely low computational, environmental, and financial costs and without the need for any training or fine-tuning. Furthermore, the findings clearly demonstrate the potential of optimizing **RAG** as part of a more sophisticated, large-scale categorization automation solution. This is especially apparent in embedding-based knowledge retrieval, which demonstrates competitive performance with previous **SOTA**, regardless of the final generative step of **RAG**.

Our research shows that extending Cipollone's pipeline with **RAG** results in a valuable tool, not only in timely and accurate vulnerability identification but also categorization, essential for understanding the nature of the detected vulnerability. Additionally, leveraging general knowledge retrieval and **LLMs** can help narrow the gap between discovery, disclosure, and correction by facilitating vulnerability analysis. This is critical in reducing the window of opportunity for exploitation and thereby increasing overall software security.

While the results are promising, there remains considerable room for improvement for future work to consider. This includes, but is not limited to, **RAG** optimization, prompt engineering, and most notably, fine-tuning the generative models used, **RAG** or both. There are also several limitations in this research worthy of consideration. These include model costs, model output token capacity, inconsistencies, and human error in the **NVD**, as well as the scarcity and sparsity of training and test data.

In conclusion, this paper has successfully demonstrated the potential of utilizing knowledge retrieval in supporting and automating vulnerability categorization. Additionally, it highlights the viability of **RAG** as an extension of Cipollone's pipeline of advanced transformer-based models: completing the loop by combining vulnerability detection and summarization with categorization. These findings suggest that with further refinement, expansion, and optimization, **RAG** can serve as a powerful tool in the ongoing mission of securing software and computer systems against emerging threats and minimizing the

vulnerability lifecycle.

References

- [1] W. Arbaugh, W. Fithen, and J. McHugh, “Windows of vulnerability: A case study analysis,” *Computer*, vol. 33, no. 12, pp. 52–59, 2000. doi: 10.1109/2.889093. [Online]. Available: <https://ieeexplore.ieee.org/document/889093> [Pages 1, 3, and 8.]
- [2] (2023) What is a Zero-Day Exploit? | IBM. [Online]. Available: <https://www.ibm.com/think/topics/zero-day> [Page 1.]
- [3] C. Charrier, J. Sadowski, C. Lecigne, and V. Stolyarov, “Hello 0-Days, My Old Friend: A 2024 Zero-Day Exploitation Analysis,” 2025. [Online]. Available: <https://cloud.google.com/blog/topics/threat-intelligence/2024-zero-day-trends> [Page 1.]
- [4] D. Cipollone, “Transformer-Based Models for Code Vulnerability Detection Automating Code : Vulnerability Detection in GitHub Issues,” 2024. [Online]. Available: <https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-356372> [Pages xi, xiii, 1, 4, 24, 25, and 34.]
- [5] About GitHub and Git. GitHub Docs. [Online]. Available: <https://docs-internal.github.com/en/get-started/start-your-journey/about-github-and-git> [Page 1.]
- [6] (2025) About issues. GitHub Docs. [Online]. Available: <https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues> [Page 1.]
- [7] “NVD - General,” NIST, 2024. [Online]. Available: <https://nvd.nist.gov/general> [Pages 1, 3, 10, and 12.]
- [8] *CWE Common Weakness Enumeration A Community-Developed Dictionary of Software Weakness Types*, MITRE Corporation Std., 2024. [Online]. Available: https://cwe.mitre.org/data/published/cwe_v4.16.pdf [Page 2.]
- [9] CWE - New to CWE. [Online]. Available: https://cwe.mitre.org/about/new_to_cwe.html [Pages 2, 14, and 32.]
- [10] “CWE - Schema Documentation - Schema Version 7.1,” MITRE, 2024. [Online]. Available: https://cwe.mitre.org/documents/schema/schema_v7.1.html [Pages 2 and 32.]
- [11] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-augmented

- generation for knowledge-intensive NLP tasks,” in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf [Pages 2, 3, 21, and 23.]
- [12] E. Aghaei, W. Shadid, and E. Al-Shaer, “ThreatZoom: Hierarchical Neural Network for CVEs to CWEs Classification,” in *Security and Privacy in Communication Networks*, N. Park, K. Sun, S. Foresti, K. Butler, and N. Saxena, Eds. Springer International Publishing, 2020. doi: 10.1007/978-3-030-63086-7_2. ISBN 978-3-030-63086-7 pp. 23–41. [Pages 2 and 26.]
- [13] T. Wang, S. Qin, and K. P. Chow, “Towards Vulnerability Types Classification Using Pure Self-Attention: A Common Weakness Enumeration Based Approach,” in *2021 IEEE 24th International Conference on Computational Science and Engineering (CSE)*, 2021. doi: 10.1109/CSE53436.2021.00030 pp. 146–153. [Online]. Available: <https://ieeexplore.ieee.org/document/9724608/?arnumber=9724608> [Pages 2 and 27.]
- [14] K. Kota, M. A, and S. V. S, “CWE Prediction Using CVE Description - The Semantic Similarity Approach,” *Procedia Computer Science*, vol. 235, pp. 1167–1178, 2024. doi: 10.1016/j.procs.2024.04.111. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1877050924007877> [Pages 2 and 27.]
- [15] M. Albanese, O. Adebisi, and F. Onovae, “CVE2CWE: Automated Mapping of Software Vulnerabilities to Weaknesses Based on CVE Descriptions:,” in *Proceedings of the 21st International Conference on Security and Cryptography*. SCITEPRESS - Science and Technology Publications, 2024. doi: 10.5220/0012770400003767. ISBN 978-989-758-709-2 pp. 500–507. [Online]. Available: <https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0012770400003767> [Pages 2 and 27.]
- [16] M. Pan, P. Wu, Y. Zou, C. Ruan, and T. Zhang, “An automatic vulnerability classification framework based on BiGRU-TextCNN,” *Procedia Computer Science*, vol. 222, pp. 377–386, 2023. doi: 10.1016/j.procs.2023.08.176. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1877050923009419> [Pages 2 and 26.]
- [17] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie, “A Survey on Evaluation of Large Language Models,” *ACM Trans. Intell. Syst. Technol.*, vol. 15, no. 3, pp. 39:1–39:45, 2023. doi: 10.1145/3641289. [Online]. Available: <https://dl.acm.org/doi/10.1145/3641289> [Pages 2, 4, and 15.]
- [18] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen. (2025) A Survey of Large Language Models. [Online]. Available: <http://arxiv.org/abs/2303.18223> [Pages 2, 15, 21, 22, and 26.]

- [19] K. S. Kalyan, “A survey of GPT-3 family large language models including ChatGPT and GPT-4,” *Natural Language Processing Journal*, vol. 6, p. 100048, 2024. doi: 10.1016/j.nlp.2023.100048. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2949719123000456> [Pages 2, 20, and 26.]
- [20] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang. (2023) Sparks of Artificial General Intelligence: Early experiments with GPT-4. [Online]. Available: <http://arxiv.org/abs/2303.12712> [Page 2.]
- [21] R. Gozalo-Brizuela and E. C. Garrido-Merchán. (2023) A survey of Generative AI Applications. [Online]. Available: <http://arxiv.org/abs/2306.02781> [Page 2.]
- [22] A. Haddad, N. Aaraj, P. Nakov, and S. F. Mare. (2023) Automated Mapping of CVE Vulnerability Records to MITRE CWE Weaknesses. [Online]. Available: <http://arxiv.org/abs/2304.11130> [Page 2.]
- [23] A. Asai, Z. Wu, Y. Wang, A. Sil, and H. Hajishirzi, “Self-RAG: Self-reflective Retrieval Augmented Generation,” 2023. [Online]. Available: <https://openreview.net/forum?id=jbNjgmE0OP> [Pages 2 and 3.]
- [24] H. Soudani, E. Kanoulas, and F. Hasibi, “Fine Tuning vs. Retrieval Augmented Generation for Less Popular Knowledge,” in *Proceedings of the 2024 Annual International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region*. ACM, 2024. doi: 10.1145/3673791.3698415. ISBN 979-8-4007-0724-7 pp. 12–22. [Online]. Available: <https://dl.acm.org/doi/10.1145/3673791.3698415> [Page 3.]
- [25] CVE: Common Vulnerabilities and Exposures. [Online]. Available: <https://www.cve.org/About/Overview> [Pages 3 and 9.]
- [26] Vulnerability APIs. [Online]. Available: <https://nvd.nist.gov/developers/vulnerabilities> [Page 3.]
- [27] S. Frei, M. May, U. Fiedler, and B. Plattner, “Large-scale vulnerability analysis,” in *Proceedings of the 2006 SIGCOMM Workshop on Large-scale Attack Defense*, ser. LSAD ’06. Association for Computing Machinery, 2006. doi: 10.1145/1162666.1162671. ISBN 978-1-59593-571-7 pp. 131–138. [Online]. Available: <https://dl.acm.org/doi/10.1145/1162666.1162671> [Pages 3, 8, and 9.]
- [28] CWE - CVE → CWE Mapping ”Root Cause Mapping” Guidance. [Online]. Available: https://cwe.mitre.org/documents/cwe_usage/guidance.html [Pages xi, 3, 12, 13, and 14.]
- [29] D. Longley and M. Shain, *Data & Computer Security*. Palgrave Macmillan UK, 1989. ISBN 978-0-333-51178-7 978-1-349-11170-1. [Online]. Available: <https://link.springer.com/10.1007/978-1-349-11170-1> [Page 7.]
- [30] C. C. Editor. Vulnerability - Glossary | CSRC. [Online]. Available: <https://csrc.nist.gov/glossary/term/vulnerability> [Page 7.]

- [31] T. D. Wagner, K. Mahbub, E. Palomar, and A. E. Abdallah, “Cyber threat intelligence sharing: Survey and research directions,” *Computers & Security*, vol. 87, p. 101589, 2019. doi: 10.1016/j.cose.2019.101589. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S016740481830467X> [Page 9.]
- [32] CVE: Common Vulnerabilities and Exposures. [Online]. Available: <https://www.cve.org/About/History> [Pages 9 and 10.]
- [33] CVE: Common Vulnerabilities and Exposures. [Online]. Available: <https://www.cve.org/About/Process> [Page 10.]
- [34] NVD - A Brief History of NVD. [Online]. Available: <https://nvd.nist.gov/general/brief-history> [Page 10.]
- [35] NVD - How We Assess Acceptance Levels. [Online]. Available: <https://nvd.nist.gov/vuln/cvmap/How-We-Assess-Acceptance-Levels> [Pages 11 and 14.]
- [36] NVD - CNAs and CVE Counting. [Online]. Available: <https://nvd.nist.gov/general/cna-counting> [Page 11.]
- [37] NVD - CVMAP. [Online]. Available: <https://nvd.nist.gov/vuln/cvmap> [Page 11.]
- [38] NVD - Understanding Acceptance Levels. [Online]. Available: <https://nvd.nist.gov/vuln/cvmap/Understanding-Acceptance-Levels> [Page 11.]
- [39] CWE - CWE Glossary. [Online]. Available: <https://cwe.mitre.org/documents/glossary/index.html#Weakness> [Page 12.]
- [40] CWE - Chains and Composites. [Online]. Available: https://cwe.mitre.org/data/reports/chains_and_composites.html [Pages 13 and 14.]
- [41] CWE - Schema Documentation - Schema Version 7.2. [Online]. Available: <https://cwe.mitre.org/documents/schema/> [Page 14.]
- [42] CWE - CWE-1000: Research Concepts (4.17). [Online]. Available: <https://cwe.mitre.org/data/definitions/1000.html> [Page 14.]
- [43] D. Khurana, A. Koli, K. Khatter, and S. Singh, “Natural language processing: State of the art, current trends and challenges,” *Multimedia Tools and Applications*, vol. 82, no. 3, pp. 3713–3744, 2023. doi: 10.1007/s11042-022-13428-4. [Online]. Available: <https://doi.org/10.1007/s11042-022-13428-4> [Page 15.]
- [44] P. F. Brown, V. J. Della Pietra, P. V. deSouza, J. C. Lai, and R. L. Mercer, “Class-Based n-gram Models of Natural Language,” *Computational Linguistics*, vol. 18, no. 4, pp. 467–480, 1992. [Online]. Available: <https://aclanthology.org/J92-4003/> [Page 15.]
- [45] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, “A neural probabilistic language model,” *J. Mach. Learn. Res.*, vol. 3, pp. 1137–1155, 2003. [Pages 15 and 16.]
- [46] H. Li, “Language models: Past, present, and future,” *Commun. ACM*, vol. 65, no. 7, pp. 56–63, 2022. doi: 10.1145/3490443. [Online]. Available: <https://dl.acm.org/doi/10.1145/3490443> [Pages 16 and 20.]

- [47] I. D. Mienye, T. G. Swart, and G. Obaido, “Recurrent Neural Networks: A Comprehensive Review of Architectures, Variants, and Applications,” *Information*, vol. 15, no. 9, p. 517, 2024. doi: 10.3390/info15090517. [Online]. Available: <https://www.mdpi.com/2078-2489/15/9/517> [Pages xi, 16, and 17.]
- [48] Z. Niu, G. Zhong, and H. Yu, “A review on the attention mechanism of deep learning,” *Neurocomputing*, vol. 452, pp. 48–62, 2021. doi: 10.1016/j.neucom.2021.03.091. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S092523122100477X> [Page 17.]
- [49] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is All you Need,” in *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017. ISBN 978-1-5108-6096-4 pp. 5998–6008. [Online]. Available: <https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html> [Pages xi, 18, and 19.]
- [50] R. Cotterell, A. Svete, C. Meister, T. Liu, and L. Du. (2024) Formal Aspects of Language Modeling. [Online]. Available: <http://arxiv.org/abs/2311.04329> [Page 18.]
- [51] H. Wang, J. Li, H. Wu, E. Hovy, and Y. Sun, “Pre-Trained Language Models and Their Applications,” *Engineering*, vol. 25, pp. 51–65, 2023. doi: 10.1016/j.eng.2022.04.024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2095809922006324> [Pages 19 and 20.]
- [52] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving Language Understanding by Generative Pre-Training,” 2018. [Pages 19 and 20.]
- [53] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019. doi: 10.18653/v1/N19-1423 pp. 4171–4186. [Online]. Available: <https://aclanthology.org/N19-1423/> [Page 20.]
- [54] Q. Dong, L. Li, D. Dai, C. Zheng, J. Ma, R. Li, H. Xia, J. Xu, Z. Wu, B. Chang, X. Sun, L. Li, and Z. Sui, “A Survey on In-context Learning,” in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.64 pp. 1107–1128. [Online]. Available: <https://aclanthology.org/2024.emnlp-main.64/> [Page 21.]
- [55] Y. Ding, W. Fan, L. Ning, S. Wang, H. Li, D. Yin, T.-S. Chua, and Q. Li. (2024) A Survey on RAG Meets LLMs: Towards Retrieval-Augmented Large Language Models. [Online]. Available: <http://arxiv.org/abs/2405.06211> [Page 21.]
- [56] K. Sparck Jones, “A statistical interpretation of term specificity and its application in retrieval,” *Journal of Documentation*, vol. 28, no. 1, pp. 11–21, 1972.

- doi: 10.1108/eb026526. [Online]. Available: <https://doi.org/10.1108/eb026526> [Page 21.]
- [57] S. Robertson and H. Zaragoza, “The Probabilistic Relevance Framework: BM25 and Beyond,” *Found. Trends Inf. Retr.*, vol. 3, no. 4, pp. 333–389, 2009. doi: 10.1561/15000000019. [Online]. Available: <https://doi.org/10.1561/15000000019> [Page 21.]
- [58] S. Na, T. Kim, and H. Kim, “A Study on the Classification of Common Vulnerabilities and Exposures using Naïve Bayes,” in *Advances on Broad-Band Wireless Computing, Communication and Applications*, L. Barolli, F. Xhafa, and K. Yim, Eds. Springer International Publishing, 2017. doi: 10.1007/978-3-319-49106-6_65. ISBN 978-3-319-49106-6 pp. 657–662. [Page 26.]
- [59] M. Aota, H. Kanehara, M. Kubo, N. Murata, B. Sun, and T. Takahashi, “Automation of Vulnerability Classification from its Description using Machine Learning,” in *2020 IEEE Symposium on Computers and Communications (ISCC)*, 2020. doi: 10.1109/ISCC50000.2020.9219568. ISSN 2642-7389 pp. 1–7. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/9219568> [Page 26.]
- [60] S. S. Das, E. Serra, M. Halappanavar, A. Pothan, and E. Al-Shaer, “V2W-BERT: A Framework for Effective Hierarchical Multiclass Classification of Software Vulnerabilities,” in *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*, 2021. doi: 10.1109/DSAA53316.2021.9564227 pp. 1–12. [Online]. Available: <https://ieeexplore.ieee.org/document/9564227> [Page 27.]
- [61] A. Cron, R. Piazza, L. Malinowski, S. A., and C. C. Steve, “CWE-CAPEC/REST-API-wg,” CWE Program. [Online]. Available: <https://github.com/CWE-CAPEC/REST-API-wg> [Page 30.]
- [62] Model - OpenAI API. [Online]. Available: <https://platform.openai.com> [Page 35.]
- [63] P. Sarthi, S. Abdullah, A. Tuli, S. Khanna, A. Goldie, and C. D. Manning, “RAPTOR: Recursive Abstractive Processing for Tree-Organized Retrieval,” in *The Twelfth International Conference on Learning Representations*, 2024. doi: 10.48550/arXiv.2401.18059. [Online]. Available: <http://arxiv.org/abs/2401.18059> [Pages 39, 40, and 51.]

Appendix A

Supporting materials

Source Code Repository

The source code and additional supporting material for this thesis are available at the following GitHub repository:

github.com/Eathus/Thesis-Edvin

CWE Structure Visualized

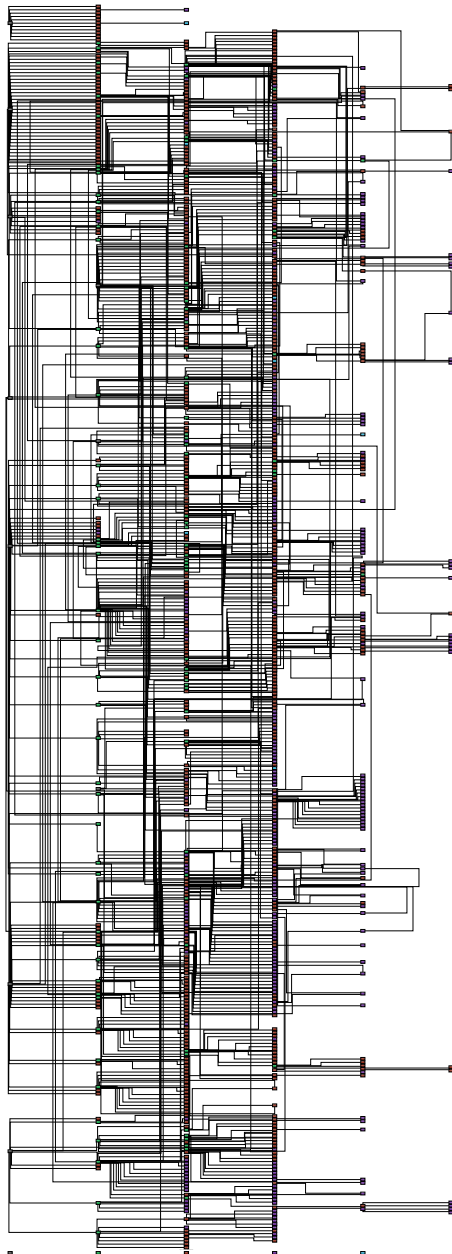
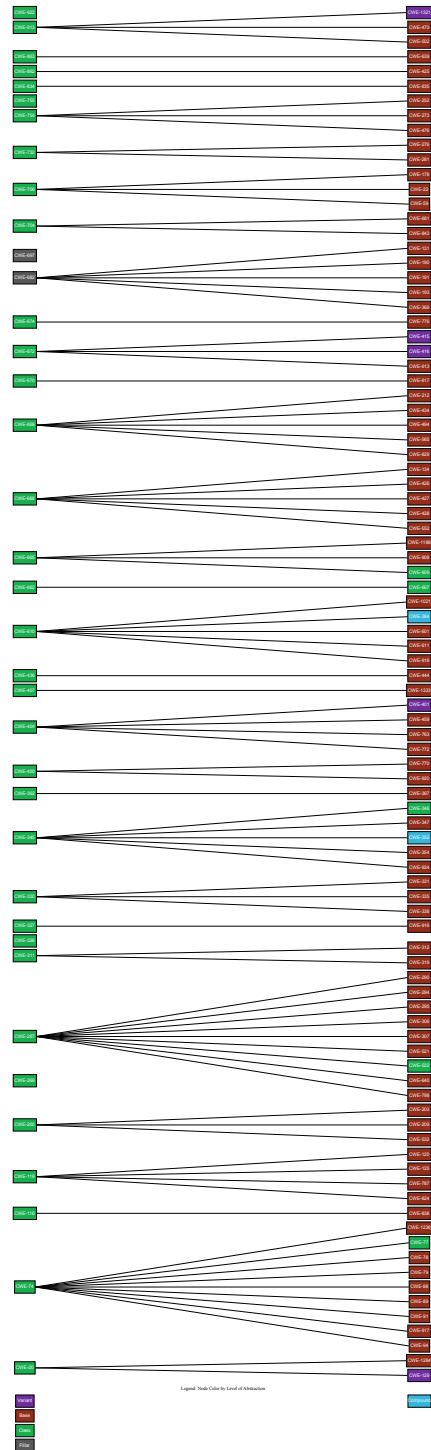
(a) View **CWE-1000**(b) View **CWE-1003**

Figure A.1: Forest-like graphs of **CWE** views 1003 and 1003. The nodes are colored in accordance with the **CWE** entry level of abstraction as specified at the bottom of the graphs. If there is more than one path from an entry, its associated node will be filled with "Path <number of paths>". The pdf file containing sub-figure **A.1a** and **A.1b** respectively are attached below. The ability to zoom in digitally may be required due to the sheer size of the views.

Attached files:  and 

