IMPROVING VULNERABILITY DESCRIPTION USING NATURAL LANGUAGE GENERATION

by

HATTAN ALTHEBEITI
B.S. Umm Al-Qura, 2013
M.S. University of Central Florida, United States, 2020

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida,
Orlando, Florida

Fall Term
2023

Major Professor: David Mohaisen

© 2023 Hattan Althebeiti

ABSTRACT

Software plays an integral role in powering numerous everyday computing gadgets. As our reliance on software continues to grow, so does the prevalence of software vulnerabilities, with significant implications for organizations and users. As such, documenting vulnerabilities and tracking their development becomes crucial. Vulnerability databases have addressed this issue by storing a record with various attributes for each discovered vulnerability. However, their contents suffer several drawbacks, which we address in our work. In this dissertation, we investigate the weaknesses associated with vulnerability descriptions presented in public repositories and alleviate such weaknesses through Natural Language Processing (NLP) approaches. The first contribution examines vulnerability descriptions in those databases and approaches to improve them. We propose a new automated method leveraging external sources to enrich the scope and context of a vulnerability description. Moreover, we exploit fine-tuned pretrained language models for normalizing the resulting description. The second contribution investigates the need for uniform and normalized structure in vulnerability descriptions. We address this need by breaking the description of a vulnerability into multiple constituents and developing a multi-task model to create a new uniform and normalized summary that maintains the necessary attributes of the vulnerability using the extracted features while ensuring a consistent vulnerability description. Our method proved effective in generating new summaries with the same structure across a collection of various vulnerability descriptions and types. Our final contribution investigates the feasibility of assigning the Common Weakness Enumeration (CWE) attribute to a vulnerability based on its description. CWE offers a comprehensive framework that categorizes similar exposures into classes, representing the types of exploitation associated with such vulnerabilities. Our approach utilizing pre-trained language models is shown to outperform Large Language Model (LLM) for this task. Overall, this dissertation provides various technical approaches exploiting advances in NLP to improve publicly available vulnerability databases.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1: INTRODUCTION

Software today is used everywhere, and their security is crucial for fulfilling their purpose. Software vulnerabilities render software systems exposed to the threat of malicious actors, allowing such actors to amplify their impact beyond the boundaries of the software (endpoint) to other associated systems and contexts, including networks, data, and users. Evidently, software vulnerabilities have been demonstrated in many applications deployed across broad industries (e.g., energy, healthcare, and businesses) worldwide, and their ramifications have shown to be severe [17, 57, 82]. Moreover, vulnerabilities in software have had a catastrophic impact on vendors' profit and reputation [21].

Vulnerabilities in modern software systems can put businesses and users at significant risk, making public vulnerability disclosure crucial for security information sharing and risk mitigation [95]. To mitigate this risk through threat information sharing, MITRE's Common Vulnerabilities and Exposures (CVE) [3] was designed to allow the disclosure of software vulnerability information in a centralized repository that can be used for improving the security of the deployed systems. The CVE entry has multiple attributes, for each vulnerability, including a unique CVE identifier, description, affected software, software version, vulnerability types, and other essential and actionable information [67]. The National Vulnerability Database (NVD) [5], managed by NIST [7], is synchronized with MITRE's CVE and seeks to structure CVE data to help inform stakeholders through a unified threat information sharing database. NVD is enhanced with additional attributes, such as Common Weakness Enumeration (CWE) and Common Vulnerability Scoring System (CVSS), which define ways to quantify vulnerability severity.

However, NVD/CVE descriptions have several shortcomings. For example, the description might be incomplete, outdated, or even contain inaccurate information, which could delay the develop-

Table 1.1: Overview of summarization approaches.

§	Task	Target	Technique	Metric
§ Zad	Summarization	Enriched description	Augmentation	ROUGE, Human
§ Mujaz	Summarization	Normalized summary	Multi-task	ROUGE, Compression, Human
§ Tasneef	Classification	Common Weakness Enumeration	Classification	Precision, Recall, F1-Score

ment and deployment of patches. In 2017 Risk Based Security, also known as VulbDB, reported 7,900 more vulnerabilities than what was reported by CVE [47, 53]. Another concern with the existing framework is that the description provided for vulnerabilities is often incomplete, brief, or does not carry sufficient contextual information [19, 22]. The lack of comprehensive and consistent descriptions will make it hard for the developer to develop the appropriate patch.

Given the fast-growing development in the software industries across various platforms and driven by the gap in addressing vulnerabilities' shortcomings, we find it crucial to investigate and fix such deficiencies in public databases systematically through NLP.

Statement of Research

In this dissertation, we propose two multi-staged pipelines that can alleviate the abovementioned issues in public databases and evaluate the results thoroughly. Table 1.1 summarizes the task, target output, technique used, and evaluation metrics for each pipeline. We further elaborate on each pipeline in the following.

***Zad*: Enriching Vulnerability Reports Through Automated and Augmented Description Summarization (Chapter 3).**

Security incidents and data breaches are increasing rapidly, and only a fraction of them is being reported. Public vulnerability databases, e.g., National Vulnerability Database (NVD) and Common

Vulnerability and Exposure (CVE), have been leading the effort in documenting vulnerabilities and sharing them to aid defenses. Both are known for many issues, including brief vulnerability descriptions. Those descriptions play an important role in communicating the vulnerability information to security analysts in order to develop the appropriate countermeasure. Many resources provide additional information about vulnerabilities, however, they are not utilized to boost public repositories. In this work, we devise a pipeline to augment vulnerability description through third party reference (hyperlink) scrapping. To enrich the description, we build a natural language summarization pipeline utilizing a pre-trained language model that is fine-tuned using labeled instances and evaluate its performance against both human evaluation (golden standard) and computational metrics, showing initial promising results in terms of summary fluency, completeness, correctness, and understanding.

***Mujaz*: A Summarization-based Approach for Normalized Vulnerability Description (Chapter 4).**

Public vulnerability databases are an indispensable source of information for tracking vulnerabilities, and ensuring their consistency and readability is crucial for developers and organizations to patch and update their products accordingly. While prior works improve consistency, unifying and standardizing vulnerability descriptions are mostly unexplored. In this work, we present *Mujaz*, a multi-task natural language processing-based system to normalize and summarize vulnerability descriptions. In doing so, we introduce a parallel and manually annotated corpus of vulnerability summaries and annotations that emphasizes several constituent entities representing a particular aspect of the description.

Mujaz exploits pre-trained language models, fine-tuned for our summarization tasks, allowing for joint and independent training for those tasks and producing operational results in terms of ROUGE score and compression ratio. Using human evaluation metrics, we also show *Mujaz* produces complete, correct, understandable, fluent, and uniform summaries.

***Tasneef*: A Classification System to Identify Common Weakness Enumeration (CWE) Using Vulnerability Description (Chapter 5).**

Vulnerabilities pose significant threats to software systems, requiring a thorough understanding of their characteristics and attributes to mitigate their impact and prevent proliferation. Common Weakness Enumeration (CWE) offers a comprehensive framework that categorizes similar exposures into classes, representing the types of exploitation associated with such vulnerabilities. However, the traditional CWE classification process relies on security experts to manually analyze vulnerabilities, which is not only time-consuming but also error-prone. In this paper, we present *Tasneef*, utilizing various methods, including Machine Learning (ML), Pre-trained Language Models (PLM), and Large Language Models (LLM), to automate CWE classification by leveraging textual descriptions of vulnerabilities. Our proposed method employs Natural Language Processing (NLP) techniques to expedite the analysis process and enhance classification accuracy. Our results demonstrate that PLMs still possess powerful capability, achieving the highest scores in terms of classification metrics, including accuracy, precision, recall, and the F1 score. Furthermore, our findings indicate that smaller PLMs can outperform much larger models. Moreover, our study highlights the effectiveness of LLMs as tools for automatic annotations that might be limited for CWE classification.

Orgnization. The dissertation is organized as follows: First, we outline the related works in Chapter 2. In Chapter 3, we examine the content of vulnerability descriptions in public databases and how sufficient they are for understanding. In Chapter 4, we study the variation in style and structure of vulnerability descriptions and the possibility of unifying and normalizing them into a uniform structure. Chapter 5 explores automating vulnerability attribute assignment using its description. Chapter 6 summarizes the main points outlined in this dissertation.

Chapter 2: RELATED WORK

Vulnerabilities are persistently being exploited, driven by the widespread of malware and viruses, as well as the improper implementation of countermeasures. In response, several models have been introduced [75, 76, 16, 74] to address such challenges. Mohaisen *et al.* [75] introduced AMAL, an automated system designed to analyze and categorize malware based on its behavior. AMAL consists of two main components: AutoMal and MaLabel. AutoMal gathers information about malware samples by monitoring and profiling their behavior. In contrast, MaLabel utilizes the artifacts generated by AutoMal to construct feature vector representations for malware samples. Furthermore, MaLabel employs multiple classifiers to classify unlabeled malware samples and cluster them into distinct groups, each containing malware samples with similar behavioral profiles.

Natural Language Processing (NLP) has found numerous applications across diverse domains, including vulnerability management [39, 18] and privacy policy analysis [14, 13]. For example, Alabduljabbar *et al.* [14] used NLP to analyze the content of third-party privacy policies for segment classification, where each segment is assigned a high-level label, signifying a distinctive practice in how users' data are being used. The authors conducted extensive experiments to test different embedding and learning algorithms, choosing the best-performing model to build an ensemble model, achieving over 90% classification accuracy by assigning the correct label for each segment.

In contrast, public repositories serve as valuable resources for comprehensive vulnerability information. However, the quality and consistency have been scrutinized in prior works [39, 20, 80, 18]. Anwar *et al.* [20] conducted an exhaustive analysis, identifying and quantifying several quality issues within the NVD. They examined their implications and proposed solutions for each issue to enhance the overall quality of the NVD. Similarly, Anwar *et al.* [21] conducted an investigation into the influence of vulnerability disclosure on the stock market and its effects across different industries. Their research categorized industries into three distinct groups based on how vulnerabilities affected vendor returns. We divide the research done on vulnerability management using NLP into three broad categories.

Contents Enrichment

The first category of studies focused on improving/enriching the content of NVD [45, 62, 18, 15] or detecting inconsistencies against third-party reports [39]. As the number of vulnerability databases increased, inconsistency and inaccuracy across those databases have been magnified, calling for methods to ensure consistency and accuracy.

Guo *et al.* [45] first investigated the discrepancies between vulnerability description by extracting key features from the X-Force Exchange [4] and SecurityFocus [1], then used the information in both databases to supplement the associated original CVE description. While their work paves the way for improving the description of the CVE entry, it does not address the issue of description normalization. Moreover, absent of ground truth, their approach does not ensure the accuracy of the supplemented description in the presence of inconsistency in the source (X-Force Exchange and SecurityFocus).

Kühn *et al.* [62] introduced OVANA, a system that uses vulnerability attributes to enhance the Information Quality (IQ) and quantify the updated information effectiveness. OVANA deploys Named-Entity Recognition (NER) system, extracting features from a vulnerability description to predict the CVSS score. The extracted features and the predicted score are used to update the NVD. However, the results for both modules were scattered as they performed well on some data and worse on others. Moreover, OVANA does not enforce consistency or ensure the accuracy of a vulnerability description.

To address the inconsistency between the software name and version in NVD and those in third-party reports, Dong *et al.* [39] proposed VIEM, including a Named-Entity Recognition (NER) based module and Relational Extractor (RE). The NER module uses the word and character embedding to locate software names/versions within a description. Because software information could be scattered, the RE module associates the extracted software information with the corresponding entity. Although successful in addressing inconsistencies in the name and version of the software, VIEM is limited by the type of vulnerabilities it addresses—memory corruption-related vulnerabilities. Moreover, VIEM does not produce normalized vulnerability descriptions.

Niakanlahiji *et al.* [81] proposed SECCMiner to analyze Advanced Persistent Threat (APT) reports. SECCMiner uses NLP techniques such as Part-of-Speech (POS) tagging and Context-Free-Grammar (CFG) to extract Noun-Phrases (NP) from the reports and then uses count-based methods to measure the importance of NP in a report across a corpus of reports. NP with the highest score is passed to the information retrieval system to map NP to a specific technique or tactic. However, their dataset is small, covering only 10 years.

Feng *et al.* [41] introduced IoTSheild, deploying NLP techniques to extract Internet of Things (IoT) reports and cluster them into different categories based on their semantics and structure. IoTSheild uses these reports to generate vulnerability-specific signatures that are deployed in Intrusion Detection System (IDS) for matching signatures and detecting exploits associated with them. However, their dataset was built using honeypots, collecting attacks on specific IoT devices, thus restricting the generalization.

Mumtaz *et al.* [78] proposed a method to learn word embedding targeted towards learning vulnerabilities by creating a dataset collected from multiple sources and using Word2Vec [72] to learn word embedding. The learned embedding better-represented terminologies like viruses and malware and associated them with similar concepts. Similarly, Guo *et al.* [46] used a similar approach to extract features from CVEs to curate a dataset. Moreover, the authors trained a Word2Vec model [72]-which is an unsupervised training algorithm to represent words with respect to their context- on a corpus of CVE descriptions to learn targeted embedding for vulnerabilities. Likewise, Yitagesu *et al.* [113] built a targeted embedding using PenTreebank (PTB) to train a BiLSTM network to tag key concepts and technical tokens, creating annotated corpus from a vulnerability description. Although this line of work utilizes some vulnerability information, it does not address the shortcomings in vulnerability databases or aim to fix them.

Vulnerability Classification

The NVD conducts a thorough analysis of any newly discovered vulnerability and assigns various attributes based on its characteristics. Therefore, the last category is to study the possibility of mapping/classifying a vulnerability to a particular attribute. For example, Gonzalez *et al.* [44] used NLP and machine learning approaches to map Vulnerability Description Ontology (VDO) to a vulnerability based on a vulnerability description. Similarly, Kanakogi *et al.* [56, 55] used NLP techniques with the cosine similarity to map CVE description to its corresponding Common Attack Pattern Enumeration and Classification (CAPEC). Three distinct embedding techniques were used to represent the description of all CAPEC and the CVE, CAPEC with the highest similarity to the CVE is assigned to it.

Several studies have addressed the need for automating CWE labeling. Aota *et al.* [23] proposed a machine learning method to classify vulnerability description into its base CWE-ID. The introduced scheme uses Bag of Word (BoW) for vectorizing the description followed by the Boruta algorithm [63], selecting the valuable words to classify CWE. The classifier used is Random Forest (RF) [26], a type of ensemble method that uses several decision trees for classifications. In [11],

Table 2.1: Comparison with similar work

Work	Task	Target	Architecture	Metric	Score	Human Evaluation
Dong et al. [39]	NER	Software names	GRU	Accuracy	0.9764	✘
Kanakogi et al. [56]	Mapping	CAPEC	Embeddings	Recall@10	0.7500	✘
Gonzalez et al. [44]	Classification	VDO Labels	Majority Vote	RBF	0.4200	✘
Wareus et al. [105]	NER	CPE Labels	Bi-LSTM	F1-Score	0.8604	✘
Our works	Summarization	Enriched Summary	BART & T5	F1 Score	0.51	✔
	Summarization	Normalized Summary	Multi-task T5	F1 Score	0.8475	✔

Aghaei et al. introduced ThreatZoom, an adaptive hierarchical classifier that automates CWE labeling by assigning CWE classes based on vulnerability descriptions. It utilizes preprocessing to extract features, employs a neural network (NN) with multiple layers to predict labels hierarchically, and has shown promising results in metrics.

In contrast, Das *et al.* [37] introduced V2W-BERT, a Hierarchical Multiclass Classification model designed to ascertain the relationship between CVE-CWE pairs and return a confidence score for their association. First, V2W-BERT extends BERT's pre-training to incorporate more context relevant to CVEs and CWEs, thereby enhancing BERT's performance in downstream tasks. The second step involves Link Prediction (LP), where the model classifies whether a link should exist between a CVE and CWE. Additionally, a Reconstruction Decoder (RD) is deployed to aid in LP training while preserving BERT's original contextual knowledge. Wåreus *et al.* [105] proposed a method to automate labeling CVE with its appropriate Common Platform Enumeration (CPE), which identifies vulnerable versions in NVD. The model was trained using BiLSTM with Conditionally Random Field (CRF) in the last layer to predict the corresponding CPEs from the text description.

CHAPTER 3: *ZAD*: ENRICHING VULNERABILITY REPORTS THROUGH AUTOMATED AND AUGMENTED DESCRIPTION SUMMARIZATION

Developing the appropriate vulnerability mitigation techniques is a challenging task [77], in practice, as that would entail vulnerability discovery and sharing, as well as patch development by the developers to address the vulnerability. Even when vulnerabilities are discovered, reporting and sharing them is not sufficient, as details pertaining to reproducing such vulnerabilities, including the associated contexts, would be paramount to realize the appropriate mitigation. Documenting vulnerabilities properly and sufficiently will not only help in building the appropriate mitigation but also in understanding the associations between different malicious actors and vulnerabilities, which could facilitate optimizations in the defense landscape [19].

Threat Information Sharing and Shortcomings. The apparent need for vulnerability information sharing has given reasons for the creation of the Common Vulnerabilities and Exposure (CVE), managed by MITRE [3], and the National Vulnerability Database (NVD), managed by NIST [5], which are two critical resources for reporting and sharing vulnerabilities to aid in the process of vulnerability mitigation. CVE allows security analysts and practitioners to report newly discovered vulnerabilities alongside their essential attributes, including a description, and assigns unique identifiers to them for referencing. The content of both NVD and CVE are mostly synchronized, and any update to the CVE should appear eventually in the NVD [39]. Moreover, the CVE description is often used to contextualize vulnerabilities, describing the software they target, associated versions, type of bugs, and their effect if they are exploited. The vulnerability description plays a vital role in communicating the vulnerability details to security analysts to understand the context of the vulnerability (and report associated information) as well as to developers, to help them develop the appropriate security patch. However, NVD and CVE's vulnerability descriptions have several shortcomings, e.g., the description could be brief, lack context, inconsistent, outdated, or incorrect [39, 80, 19]. The vulnerability description is produced by a security analyst who discovers the vulnerability, and its sufficiency may vary, depending on the analyst's perception of the importance of the information they include in this description. Often, the description provided for a vulnerability is brief. Moreover, we noticed that for some vulnerabilities, the description hardly constitutes any information and describes it at a very high level that requires more context. The absence of a comprehensive description will make it hard for the developer to develop the appropriate patch. Moreover, with the diversity of technical resources (e.g., hyperlinks pointing to additional contexts and reports) associated with each vulnerability, evaluating and determining the best one that would aid the original description becomes highly burdensome.

Goals and Challenges. This work aims to systematically enrich the originally provided descriptions in NVD reports by augmenting them with additional resources to achieve two goals: (1) contextualize the original description by containing it within a proper context, and (2) incorporate additional information missing from the original description. In doing so, we address various challenges:

- As the description augmentation should be only based on a novel and relevant contents, there is a need for devising a method for selective discovery and inclusion of potential resources to the summary.
- As the potential description does not follow a normalized form and varies in length and structure, there will be a need to normalize the structure across different vulnerabilities.
- As the objective of the final description is to improve the context and enrich the initial description, there is a need for sound evaluation to assess the realization of such end goals.

Our Approach. To address these challenges, we present *Zad*. The core of *Zad's* design is a Natural Language Processing (NLP) pipeline that normalizes an augmented description over the description entry provided originally in NVD for a given vulnerability. To enrich the original description, we define a similarity metric that discovers the relevant resources for inclusion in the potential eventual description. The normalization task of the vulnerability description is then formulated as a summarization task, where the extended summary is provided as an input alongside a cue for semantic relevance, and a pretrained language model is fine-tuned for this specific task. The fine-tuning process involves using different cues as a reference. Namely, we devise label-guided (i.e., requiring the generation of a few summaries manually) and summary-guided (i.e., requiring providing the original summary as a semantic cue), fine-tuning processes and show their effectiveness. For the effectiveness evaluation, and given the domain-specific nature of our problem, we develop various human metrics and demonstrate that our approaches perform well across them.

Building Blocks

Zad is composed of several components that build up its architecture. However, we first address challenges faced in building *Zad* and how we alleviated those issues in (§3). We then present *Zad's* pipeline, elaborating on each subcomponent in (§3). Next, we introduce a fundamental component of *Zad*, sequence encoder in (§3), explaining its role and the different architectures available for deployment. Finally, we present two pre-trained models commonly used for summarization in (§3).

Challenges

Zad tries to answer a simple question, how can we improve vulnerability description presented in public databases? Leveraging the availability of third-party reports proposes a solution to this problem. However, implementing a methodology that systematically achieves this is challenging. In the following, we address some of these challenges.

Challenge 1: Augmentation technique. *Zad* deploys augmentation as a technique to collect textual content to supplement the original description. However, this process requires defining a metric or criteria for augmentation, ensuring the content is relevant. We approach this by viewing the task as a semantic search problem; given a text (vulnerability description), find the relevant paragraphs presented on a third-party web page content. Therefore, we deploy the cosine similarity as a measure of relevancy to ensure both texts are similar in embedding space.

Challenge 2: Text Encoding . Encoders generally aim to represent a type of data into a unique expression that captures its characteristics in a high-dimensional space. Discrete data, such as text, could be encoded at different levels of abstraction(e.g., word or sentence). Word encoding builds a representation, capturing a word's meaning within its context. On the other hand, sentence encoding creates a single representation for the entire sentence or sequence, capturing various features and semantics in its content. We choose to use a sentence encoder for the following reasons: (1) static word embeddings [72, 84] retain the same embedding for a word regardless of its context. (2) Dynamic word embeddings [85] use a word's context from both directions to build its embedding; however, given the domain-specific nature of our text and the fact that we are scraping blindly, third-party websites make this approach susceptible to various issues. (3) Finally, using word embedding necessitates aggregating those embeddings into a single embedding, representing the sentence. Moreover, this approach mandates ensuring the embedding space describes sentences.

Challenge 3: Evaluation. *Zad* is expected to produce a summary derived from the augmented text yet similar to the original description. However, vulnerability descriptions/reports contain crucial information, making their evaluation critical. Summarization tasks typically rely on word overlaps

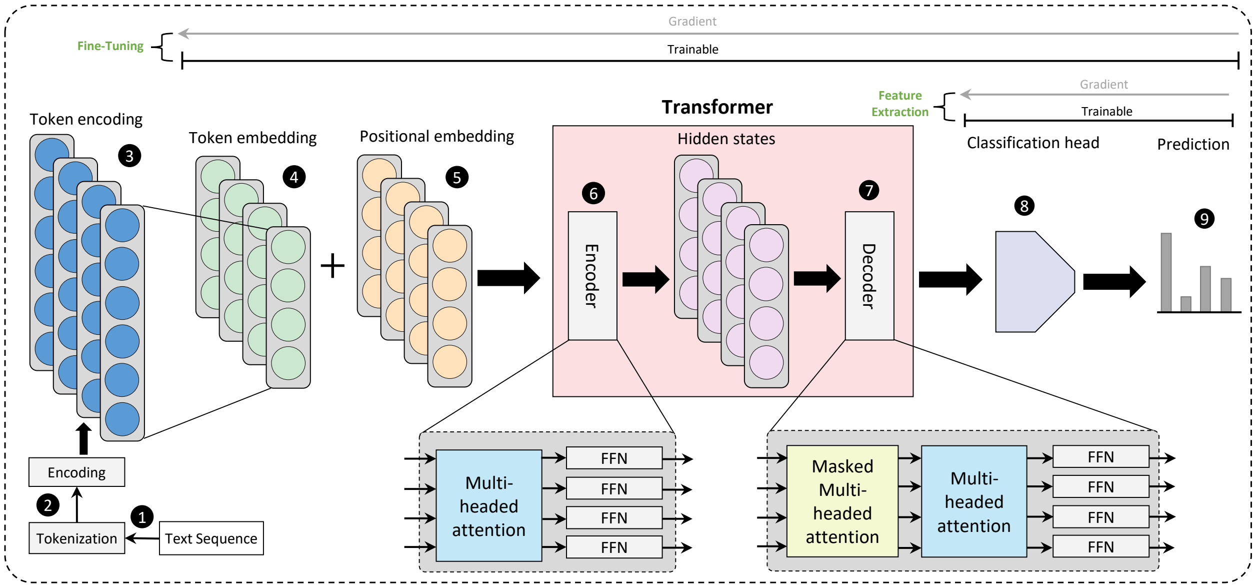

Figure 3.1: *Zad* pipeline. The pipeline consists of multiple steps to transform the text $X_{1:n}$ from a series of tokens into numerical representations. The encoder component then encodes them to incorporate their context, improving their embeddings through self-attention. The decoder's role is to deploy the encoded sequences to predict the target sequence $Y_{1:m}$.

between the generated summary and the label summary to measure its quality. These metrics are insufficient and limited in evaluating content accuracy (e.g., software name or bug). Moreover, computational metrics do not implicate other linguistic aspects of the generated summaries (e.g., fluency or comprehension). Therefore, we develop human metrics providing a set of comprehensive measures to evaluate the content quality in terms of fluency, completeness, correctness, and understanding.

Zad's *Pipeline*

Zad's pipeline is depicted in Figure 3.1. *Zad's* goal is to produce an enriched summary for a vulnerability, providing adequate context with meaningful content while minimizing the semantic gaps between the produced and target summaries. The objective is to find a function $\mathscr{F}$ parametrized by θ to map input text x under length constraint δ to produce output y. More formally, the loss function can be framed as:

$$\min_{|x|<\delta} \left(\mathscr{L} \left(\mathscr{F}_{\theta}(x, y) \right) \right) \tag{3.1}$$

Parameters θ for the pre-trained model are adjusted according to the objective function. In addition, the number of parameters θ is dependent on the underlying model and its size.

The dataset is created using the augmented text assembled from third-party reports as input and the vulnerability's description as the target. Moreover, we devise a second method, a label-guided summary, by selecting 100 samples from the dataset and manually creating their summaries based on the information provided in the augmented text and the description. The dataset creation and annotation details are explained in (§3). Therefore, the model has two options for fine-tuning: the description-guided or the label-guided dataset. Seq2Seq models [97] depend on two text sequences, an input of length n and a target of length m, and the model is expected to find a function F that maps $X_{1:n} \rightarrow Y_{1:m}$.

Next, we discuss the pipeline depicted in Figure 3.1 in more detail. The major steps of our pipeline are tokenization of the input text sequence (description), encoding, token embedding, positional embedding, encoding-decoding (utilizing a fine-tuned pretrained language model), and prediction. Those steps are elaborated in the following.

Annotation. The first step is to prepare another dataset for the label-guided summary approach by annotating 100 samples from the dataset. This step aims to derive a new dataset with ground-truth summaries. Labeling the dataset follows some specific guidelines, explained in section 3. After annotation, two datasets are prepared for training: the description-guided and the label-guided datasets. Only one dataset is chosen and passed through to the next step, which is tokenization.

Tokenization. The first step in the pipeline is tokenization, breaking a text into separate and independent entities called tokens. The tokenization could be based on a unigram, where each word is represented as a single token. In essence, a tokenization can be formulated as $\mathbb{F} : \mathbb{S} \rightarrow \mathbb{S}^{'}$ in which $\mathbb{F}$ is a function that takes a string of text $\mathbb{S} = \{a_1\ a_2\ a_3\ a_4 .... \ a_n\}$ as an input and outputs a set of tokens $\mathbb{S}^{'} = \{a_1,\ a_2,\ a_3,\ a_4, ...,\ a_n\}$.

Tokenization could be applied at a word, sub-word, or character level. Word tokenization will split words based on white spaces or punctuation, and every word, including rare words, will be added to the vocabulary set commonly known as vocab. One drawback of this approach is that the vocabulary size might grow significantly large. Consequently, the dimensionality of the embedding matrix storing all words will be enormous, making training and inferencing costly. To address this problem, it is common to limit the size of the vocabulary set to the most common $100,000$ words in the training corpus and encode all unknown words as <UNK>. Other issues can arise with word tokenization, such as word morphemes, standard abbreviations, and acronyms, but all can be solved using rule-based methods during preprocessing.

On the other hand, character-level tokenization breaks a text into individual characters, which makes the vocab size dependent on the language. Still, it will be smaller compared to words vocab. The intuition behind this is that any text is composed of these characters, and training a model to use these building blocks yields a model that can represent any unknown word. Another advantage of this approach is that it reduces computation costs concerning time and memory. In contrast, the major limitation of character-based tokenization is losing the linguistic structure and considering a text as a stream of characters. This method is standard in certain applications or languages where individual characters play diverse roles and possess more meaning.

A third type is sub-word tokenization, which alleviates the drawbacks of the two aforementioned methods granting a fine-grained method for tokenization. Sub-word tokenization is more sophisticated and requires training a tokenizer to build up the vocab from the provided corpus and tokenize new text accordingly. Sub-word tokenization splits words into simple units learned during training, allowing the tokenizer to handle complex words and associate their embedding with words that have the same constituents. This allows the model to associate singular with plural and relate different morphemes to their root. BART uses Byte Pair Encoding (BPE) [93], and T5 uses SentencePiece [61], which are both sub-word tokenizers, discussed next.

Byte Pair Encoding (BPE). was initially developed as a text compression algorithm [43]. However, the idea was extended to allow for the representation of open vocabulary with size-limit through variable length characters sequence. BPE was adopted and deployed as a tokenizer by many pre-trained models such as GPT [86], GPT-2 [87], and RoBERTa [68]. The intuition behind BPE is that sub-word segmentation could allow a representation of words (including rare and unknown) based on their sub-units which might be shared with other words with similar meanings. As a result, the vocabulary size is minimized, and multiple words could share a unit or more when they are tokenized. We consider the English language in this context for the rest of our explanation.

BPE starts by building a base vocabulary set consisting of ASCII characters and some Unicode characters. This set will represent the initial state of the vocab that will be expanded by merging some of them to create a new token. However, it is essential to set a maximum limit for the vocab size to limit the number of appended tokens. Otherwise, our vocabulary will grow uncontrollably to include various substrings. Moreover, it is essential to note that BPE operates on normalized and pre-tokenized text. Normalization refers to standardizing the text, which includes removing accent or redundant white spaces and lowercasing letters. Pre-tokenization refers to a preprocessing step in which transformer models handle raw texts and perform any necessary adjustments for the model to handle them, such as breaking words on white space or adding some symbols to indicate a unique aspect of the text.

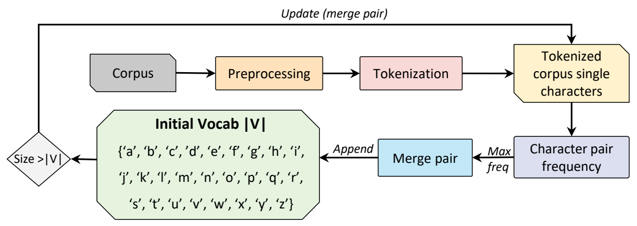
Update (merge pair)
Corpus
Preprocessing
Tokenization
Tokenized corpus single characters
Character pair frequency
Max freq
Merge pair
Append
Initial Vocab |V|
{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'}
Size >|V|

Figure 3.2: A schematic structure for training BPE tokenizer

After pre-tokenization, the corpus is represented as a set of words. BPE starts by breaking every word into individual characters, adding them to the base vocabulary, and maintaining this corpus with single characters. BPE then looks at every two consecutive characters (pair) in this set and their frequency in the corpus; the pair that appears the most in the corpus will be merged to create a new token that will be added to the base vocabulary. The pre-tokenized corpus with single characters will be updated to merge those two characters. This process is repeated by tracking the frequency of two consecutive tokens, merging the most frequent two consecutive (tokens) to create a new one based on the frequency of the consecutive tokens, and adding it to the base vocabulary until it reaches the size limit. The tokenizer is trained on an enormous raw text that should represent the data it will be applied to. The tokenizer also maintains a unique token, such as <UNK>, that represents unknown tokens not learned during training or not included in the vocabulary due to the size limit. Applying the tokenizer to a text will break the text into tokens contained in the base vocabulary, and every character/token that is not contained will be tokenized into <UNK>. Figure 3.2 depicts a simple pipeline for the BPE tokenizer and its procedural steps.

SentencePiece.is another sub-word segmentation algorithm based on the Unigram language model. The Unigram model is a statistical model with two assumptions: (1) It assumes the occurrence of each word is independent of its previous words. (2) A word's occurrences depend on its frequency on the dataset. A sequence $\mathbf{x}$ represented as $\mathbf{x} = (x_1, \ldots, x_m)$ can be modeled as:

$$P(\mathrm{x}) = \prod_{i=1}^{m} p\left(x_i\right), \forall i \; x_i \in \mathcal{V} \tag{3.2}$$

such that $\mathcal{V}$ is the vocabulary set and m is the index of the last token in the sequence, e.g., $P(x_1, x_2, ..., x_m) = P(x_1) \times P(x_2) \times \ldots \times P(x_m)$. SentencePiece starts with an extensive vocabulary set obtained from a corpus and works by removing tokens from that set until it reaches the desired size. The author suggests using all the words presented in the corpus and their most frequent sub-strings, along with all individual characters, as the base vocabulary for the model.

SentencePiece uses the Unigram model to determine the least effective token in the base vocabulary and remove it from the set. First, given a word in the corpus, the model looks to tokenize it by considering multiple candidate combinations that make up this word from the base vocabulary. For example, a word can be formed by its single characters; the probability for this combination will be the frequency of each token (character) divided by the sum of all tokens' frequencies in the vocabulary set, which produces a probability for each token. Following the Unigram model, each token's probability is multiplied to yield the probability for that combination, which can be viewed as a score. This procedure is repeated for every combination in the vocabulary set that makes up the word. Moreover, every word in the corpus goes through the same process to find the best tokenization for each word. The best tokenization of a word is the one that produces the highest probability or score. After obtaining all the scores for a word, the model computes the loss as the sum of the frequency of a word multiplied by the probability of that word. The loss used here is the negative-log likelihood, and the loss function becomes:

$$loss = \sum freq \times (-\log(P(\text{ word }))) \tag{3.3}$$

The model then computes the effectiveness of removing a token from the base vocabulary by removing a token from a vocabulary through Expectation Maximization (EM). The model optimizes the vocabulary by removing a token and recomputing the loss. However, the loss might change because by removing a token, the combinations used to construct the word may change. In contrast, removing a token may not affect the loss and remain the same. This process is repeated for every token in our vocab to determine the token with the minimum impact on the loss which will be removed. The model sorts the loss and keeps top η tokens in the vocab. It is essential to point out that the model does not remove any single character to overcome (OOV) or out-of-vocabulary problems. One final thought on the model, the model does not consider all combinations that make up a word; instead, it obtains the most probable segmentation for a word using Viterbi algorithm [103]. Viterbi algorithm is used mainly in the Hidden Markov Model (HMM), in which the observed data is used to predict a sequence of states. The algorithm builds a graph that detects the possible segmentation of a word using the base vocabulary and attributes each branch between two characters to a probability depending on whether those two characters are presented in the base vocabulary. To find the best path is to find the best segmentation represented by a sequence of branches that produce the highest probability for a particular token.

Token Encoding. The tokenized text is transformed into numerical representation using one-hot encoding with a size equal to the vocabulary size; e.g., 20k-200k tokens.

Token Embedding. The embedding layer projects a token into a vector space of a certain dimension that can be fed into a neural network. Each token is represented with a vector of dimension d such that a token $x \in \mathbb{R}^d$. Token embedding could be initialized with random values or using precomputed embedding like Word2Vec [72, 73] or Glove [84]. In both cases, the embedding will improve and gets updated according to the training dataset. The vector's dimensionality is a hyperparameter, typically set to 512.

Positional Embedding. After tokenization, positional embeddings are created to preserve tokens' positions. Self-attention is permutation equivariance, meaning that shuffling tokens within the sequence will not change the final output representation for each token. Therefore, it is important to break this property, devising a new method to maintain tokens orders. There are a few conditions that would render a positional embedding to be ideal. For example, the embedding should be unique at each time step, and the distance between two-time steps (two consecutive tokens) should be constant. Moreover, the method should generalize to longer sequences. The positional embedding could be either learned during training or fixed using a predefined function. Next, we describe the methods used by our pre-trained models, BART and T5.

Positional Embedding (PE). assigns an embedding vector for each position, defining a unique embedding for each token in the sequence. Therefore, the same word will have different embedding at different positions. One advantage of this method is its simplicity and effectiveness. In contrast, PE cannot encode sequences longer than the largest sequence observed during training, which is an issue if the maximum length is unknown. BART uses PE to represent positional information.

Relative Positional Encoding (RPE). deploys a dynamic method to learn word's position using the attention layer within the encoder. We keep it simple here by focusing on the role of the attention layer to integrate positional information. The attention layer computes how much attention a word should pay or "attend" to other words within a sequence. RPE incorporates positional embedding by analyzing the pairwise relationship between words within a sequence. The input representation is modeled as a fully connected directed graph. For tokens x_i and x_j within a sequence, two edges are represented as a_{ij}^V and a_{ij}^K. The representation of those edges can be incorporated into the attention layer to realize the position of a token with respect to other tokens. The key idea is that a token's relative position with its neighbors matters the most, rather than its absolute position. Moreover, propagating edge information into the attention layer could encode additional information about the relationship between tokens.

Transformer. The Transformer consists of an encoder an a decoder. The encoder uses a multi-headed attention to build a representation that captures the contextual interdependence relationship between tokens. The encoder uses several layers of self-attention to compute how much attention should be paid by every token with respect to other tokens to build the final numerical representation. Modern transformers use the scaled dot product attention which utilizes a query, key, and value computed for each token to produce the attention score for every token with respect to other tokens in the sequence. A simple intuition behind applying several attention layers (heads) is that each head may focus on one aspect of attention, while others may capture a different similarity. By concatenating the output of all heads, however, we obtain a more powerful representation that resembles that sequence. The feed forward network receives every token embedding from the multi-headed attention and processes it independently to produce its final embedding which is referred to as the hidden states.

As the encoder outputs a representation of the input sequence, the decoder's objective is to leverage the hidden states to generate the target words. We note that summarization requires text generation to generate the next token in an autoregressive fashion. As such, the generation procedure's objective is to predict the next token given the previous tokens. This can be achieved using the chain rule to factorize the conditional probabilities as

$$P(x^{(t+1)}|x^{(t)},...,x^{(1)}) = \prod_{t=1}^{T} P(x^{(t+1)}|x^{(t)},...,x^{(1)}) \tag{3.4}$$

A numerical instability results from the product of the multiple probabilities as they become smaller. Thus, it is common to use the log of the conditional probability to obtain a sum, as

$$\log(P(x^{(t+1)}|x^{(t)},...,x^{(1)})) = \sum_{t=1}^{T} \log(P(x^{(t+1)}|x^{(t)},...,x^{(1)})) \tag{3.5}$$

From this objective, there are various methods to select the next token through decoding with two aspects to consider. (1) The decoding method is done iteratively, where the next token is chosen based on the sequence at each time step. (2) It is important to emphasize certain characteristics of the selected word; e.g., in summarization we care about the quality of the decoded sequence, compared to storytelling or open domain conversation where care more about the diversity when generating the next token.

Decoding. In this work, the beam search is used as decoder, since summarization emphasizes factual or real information in the text. This method is parameterized by the number of beams, which defines the number of the most probable next tokens to be considered in the generated sequence and keep track of the associated sequences by extending a partial hypothesis to include the next set of probable tokens to be appended to the sequence until it reaches the end of sequence. The sequences are then ranked based on their log probabilities, and the sequence with the highest probability is chosen. It is important to ensure that at each time step, the decoder is conditioned on the current token and the past output only. This step is crucial to assure the model does not cheat by accessing future tokens. While the transformer architecture is task-independent, the classification head is task-specific, and we use a linear layer that produces a logit followed by a softmax layer to produce a probability distribution for decoding.

Operational Considerations. Transformers are typically deployed in one of two setting. (1) As a feature extractor, where we compute the hidden states for each word embedding, the model parameters are frozen, and we only train the classification head on our task. Training using this method is fast and suitable in the absence of resources to fine tune the whole model. (2) As a fine-tuning setting, where all the model trainable parameters are fine-tuned for our task. This setting requires time and computational resources depending on the model size. In our case we use BART and T5 for fine-tuning and since BART has a smaller number of parameters, its fine-tuning is faster.

Sentence Encoders

As explained in 3, we decided to use sentence encoder to represent our data. However, to make our approach more concrete, we tested several sentence encoders on multiple CVEs with their associated reports from third-party and found that the best encoders for our task are Universal Sentence Encoder (USE) [29] and MPNet sentence encoder [2]. In the following, we elaborate on each encoder and its internal architecture.

Universal Sentence Encoder (USE). USE has been introduced in [29] with two architectures; each has its own traits and trade-offs. The first architecture follows the transformer model. First, the encoder preprocesses the sequence by lowercasing and tokenizing it according to Penn Treebank (PTB). The tokenized sequence is passed through self-attention to compute context-aware representation for tokens while preserving their order within the sequence. The final step is to aggregate the produced word embeddings using element-wise sum divided by the sentence length, constructing the sentence embedding. Although providing accurate sentence representation, the time and space complexities are proportional to the sentence length n, taking $O(n^2)$. The second architecture is much simpler, using Deep Averaging Network (DAN) [54] to compute sentence embedding. In DAN, words are initialized with pre-trained (static) embeddings. The sequence's input embeddings and bi-gram embeddings are averaged together and passed through a Feed-Forward Network (FFN) to produce the final sentence embedding. DAN takes $O(n)$ time and space complexity, making it linear with respect to the sequence length.

The trade-off between the two architectures is high accuracy with intensive computation achieved by the transformer versus efficient inference and computation with reduced accuracy achieved by the DAN. Given the problem we are trying to solve, we chose DAN's architecture because (1) our data will be scraped, and its length may vary widely, indicating the need to consider time and space complexities. (2) our data is domain-specific, implying that its linguistic scope and vocabulary are limited, minimizing the effect of accuracy on the generated embedding. (3) considering that we have over $35,000$ Vulnerabilities, where each has multiple hyperlinks to be scraped with possibly hundreds or thousands of paragraphs per hyperlink, the scalability benefit of DAN outweighs the high accuracy of the transformer-based architecture.

MPNet. The second encoder we utilize is the MPNet sentence encoder. MPNet [96] is a model that leverages the advantages presented in two famous pre-trained models: BERT [38], and XLNET [112]. BERT uses a Masked Language Modeling (MLM) objective, which masks 15% of the tokens, and the model is trained to predict them. The downside of BERT is that it does not consider the dependency between the masked tokens. On the other hand, XLNET retains the autoregressive modeling by presenting Permuted Language Modeling (PLM) objective in which each token within a sequence considers the permutations of the previous tokens in the sequence but not after it. However, this causes position discrepancy between the pre-training and fine-tuning stages.

MPNet unifies the two objectives of BERT and XLNET and addresses their shortcomings by considering the dependencies between predicted tokens and considering tokens' positions within a sequence to solve the positional discrepancy issue. The MPNet sentence encoder is built by fine-tuning MPNet on many datasets, including [48, 69, 40, 65], totaling $1,170,060,424$ instance. Each entry is represented as a tuple of sentence-pair. The model is then trained using a contrastive objective to construct sentence embedding based on the true pair. Given a sentence from a pair, the model tries to predict the other sentence it was paired with. This is done by computing the cosine similarity with every other sentence in the batch and then using the cross-entropy loss with respect to the true pair.

Pre-trained Models

This work aims toward deploying pre-trained models and fine-tuning them on our datasets for vulnerability summarization and description enrichment. Pre-trained models essentially deploy transfer learning, first presented in computer vision [60, 94, 98], achieving significant success in the field. Transfer learning trains a Neural Network in an unsupervised manner on general objectives such as predicting the next token in a sequence or predicting the masked token. The learned model is then fine-tuned using a targeted dataset on a specific task. Transfer learning for NLP was first introduced in [51] and adapted afterward in the transformer models. Pre-trained language models inherit the original transformer architecture [102] or some of its sub-component.

The transformer constitutes two major components: an encoder and a decoder. The encoder's role is to build a representation for the input sequence that captures the dependencies between tokens in parallel without losing the positional information of those tokens. The transformer relies on the attention mechanism to capture interdependency within a sequence, providing a context-aware representation for each token. The decoder uses the built representation and maps it to a probability distribution over the entire vocabulary to predict the next token.

The original transformer was developed and was intended for machine translation [104]. However, it generalized to other tasks with outstanding results. We note that most modern pre-trained models use a transformer architecture that depends on the encoder only, e.g., BERT [38], the decoder only; e.g., GPT (Generative Pre-trained Transformer) [86], or both. Each architecture has its own advantages, allowing it to excel in specific tasks. The summarization task, for example, can be modeled as a sequence-to-sequence task where the model takes an input (long text) and outputs a summary, making models that constitute an encoder and a decoder ideal for its design.

In the NLP literature, the most prominent models for summarization are BART [64], T5 [89], and Pegasus [114], with BART and T5 being more widely used. BART is a denoising autoencoder for pretraining seq2seq with an encoder-decoder architecture. The idea of BART is to use a noising function to corrupt the text and train the model to reconstruct the original (uncorrupted) text. In contrast, T5 uses MLM objectives, like BERT, for training. However, T5 masks a span of the original text as its corruption strategy. The span length does not influence the model performance unless too many tokens are within that span. Moreover, T5 aims to define a framework for many NLP tasks by adding a prefix that identifies the task it tries to learn. Therefore, one model can support multiple tasks by defining those prefixes in the training data. Adding those prefixes to a sample allows the model to predict the task associated with that prefix, even if it is not a typical NLP task. In the next section, we present the pipeline in great detail.

Dataset and Data Augmentation

Data Source and Scraping. Our data source is NVD [5] because it is a well-known standard accepted across the globe, in both industry and academia, with many strengths: (1) detailed structured information, including severity score and publication date, (2) human-readable descriptions, (3) capabilities for reanalysis with updated information, and (4) powerful API for vulnerability information retrieval. In our data collection, we limit our timeframe to vulnerabilities reported between 2019 and 2021 (inclusive). Based on our analysis, CVEs reported before 2019 do not include sufficient hyperlinks with helpful content, which is our primary source for augmentation. We scrape the links for all vulnerabilities in our timeframe, totaling $35,657$ vulnerabilities. For each vulnerability, we scrape the description and the associated hyperlinks, which will be scraped next for augmentation.

Description Augmentation. First, we iterate through the scrapped hyperlinks, leading to a page hosted by a third party, which could be an official page belonging to the vendor or the developer or an unofficial page, e.g., a GitHub issue tracking page. For each page, we scrape text contained within each paragraph tag ($<p>$ $<\backslash p>$) separately and apply various preprocessing steps to clean up the text. The preprocessing includes removing additional web links, some special characters, redundant white spaces, phone numbers, and email addresses. After preprocessing, we check the paragraph's length and ensure it is more than 20 words. We conjecture that paragraphs shorter than 20 words will not have enough details to fulfill our purpose.

Next, we use a sentence encoder to encode the semantics for the extracted paragraph and the scrapped description into low dimensional vector representations as explained in section 3. We use the cosine similarity to determine the similarity between the vectorized representations, which yields a value between -1 and 1. For example, let the vector representation of the extracted paragraph be $\mathbf{v}_p$, and that of the description be $\mathbf{v}_d$. The cosine similarity is defined as:

$$\cos(\mathbf{v}_p, \mathbf{v}_d) = \frac{\overrightarrow{v_p} \cdot \overrightarrow{v_d}}{||\overrightarrow{v_p}||\ ||\overrightarrow{v_d}||} \tag{3.6}$$

If $\cos(\mathbf{v}_p, \mathbf{v}_d)$ exceeds a predefined threshold, we add/augment the paragraph to the augmented text, expressing the input text. This process is repeated with every paragraph contained within a page. We repeat this step for every hyperlink by extracting paragraph tags within the page, applying the preprocessing stage, encoding semantics, and measuring the similarity with the description. After going through all hyperlinks and augmenting similar paragraphs, we create an entry in a new dataset with the augmented text as the input text and the description as the target summary. We note that some vulnerabilities may not be added to the dataset, e.g., if the vulnerability did not have hyperlinks or its hyperlinks did not include any paragraph that meets the predefined similarity threshold. This process is repeated for each vulnerability until we cover all $35,657$ vulnerabilities,

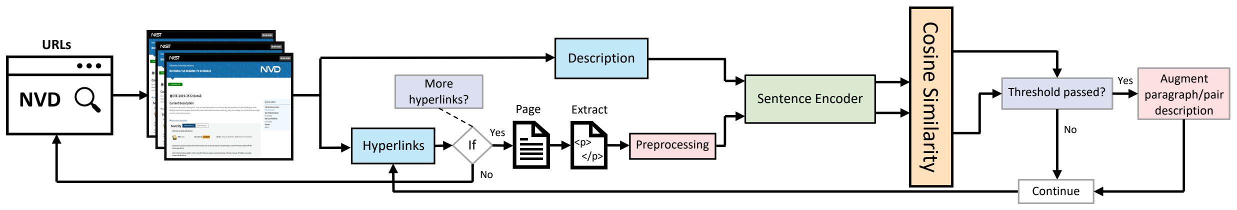

Figure 3.3: Our data collection pipeline, constituting the first part of *Zad*. The pipeline starts with a predefined set of URLs associated with CVEs (enumerated from NVD) and scrapes the description and its hyperlinks. For each hyperlink, the pipeline invokes the extraction of paragraphs, preprocesses them, and encodes them with the description. Finally, a step for measuring the similarity between the paragraph and the original description is invoked to determine whether to include the paragraph in the augmented dataset.

upon which our dataset is ready.

Figure 3.3 shows the pipeline for our data collection. The pipeline aims to collect and build the dataset by performing the aforementioned steps. However, the choice of a sentence encoder will affect the dataset because the inclusion of a paragraph is based on the similarity score between the vectorized representation of the description and the paragraph encoded by the sentence encoder. The similarity score must exceed a predefined threshold, per Figure 3.3. From our preliminary assessment of the two encoders, we found that USE is more accurate (sensitive) than MPNet in terms of the similarity score representation, meaning that when the description and the paragraph are (semantically) similar to one another, USE produces a higher score than MPNet and vice versa. Considering this insight, we set different thresholds for each encoder.

Namely, we set the similarity score for USE to be between 0.60 and 0.90 since the encoder is accurate. On the other hand, since MPNet is less accurate (sensitive) than USE, we enforce a more restrictive threshold and set it between 0.70 and 0.90. We excluded paragraphs with a similarity score above 0.90 because we found such paragraphs to be identical to the description with minor changes; thus, adding them would not serve the primary purpose of enriching the description.

Those values were picked as part of our assessment of the two encoders using a small set of vulnerabilities and following the above procedure. Therefore, our pipeline builds two datasets using each encoder.

Additionally, we build a third dataset using both encoders and enforcing multiple thresholds on the similarity criterion. Therefore, we used the same threshold for MPNet as before and lax the threshold for USE to 0.50 to ease the restrictive setting imposed by two encoders. Given the differences between the two encoders, we consider a paragraph similar if the difference between the two similarity scores is at most 0.20 to ensure their semantics are close in embedding space; otherwise, we consider them dissimilar and discard the paragraph. Here, we favored the consistency between the two encoding techniques to conceptually alleviate the discrepancy presented by using two different encoders.

Some hyperlinks processing took an extremely long time in our preliminary analysis. However, upon examining the content of those pages, we found that they contain a history of software vulnerability with updates, e.g., over 20,000 paragraph tags in some cases. Moreover, most of them were not considered by the sentence encoder because they did not meet the threshold. As such, we consider the first 100-paragraph tag in each hyperlink to speed up the process. We justify this heuristic by noting that most pages contain related textual information at the beginning, with subsequent paragraphs reiterating previously mentioned information. Finally, we consider hyperlinks with valid SSL certificates, limiting our collection to authentic content. Table 3.1 shows the datasets and the number of vulnerabilities using the above procedure. The next section will discuss the label-guided dataset and how we created the ground-truth summaries for 100 samples.

Label Guided Dataset. The adapted approach emphasizes utilizing the description as a label for the augmented text, relying on the model to produce enriched and concise summaries. However, this approach confines the model to short descriptions, restricting its ability to generate longer sum-

Table 3.1: Datasets and their high-level characteristics.

# CVEs	Encoders	Vuln.
35,657	USE	9,955
	MPNet	8,664
	Both	10,766

maries. Moreover, the content of the augmented text for some vulnerabilities is very long compared to its corresponding description, which complicates learning. As stated above, the augmented text relies on simple criteria (similarity measure) for augmentation, making it susceptible to various issues, e.g., highly similar text but with crucial differences. Therefore, we randomly select 100 samples from the mixed dataset and label them manually, creating ground-truth summaries.

We make a few considerations before creating the summaries. First, given that the augmented text could be sparsed, unstructured, or disorganized, we write our summary in an extractive style, such that the newly written summary is extracted from the augmented text while ensuring conciseness and coherence. Our justification is that the model will learn to generate a summary by extracting salience sentences/paragraphs from the augmented text. Moreover, we only include version numbers if they are very limited, as we mainly focus on the summary quality. We included the original description as part of the ground-truth summaries for very few instances when the augmented text is missing crucial detail contained in the description. Considering our extractive approach, ensuring our manual summary reflects the specific CVE associated with the augmented text is critical. However, for several cases, the augmented text included information related to other CVEs. Therefore, we had to manually verify the augmented text and use content associated with that particular CVE to write the summary.

Evaluations

Statistical Analysis. After assembling the three datasets as shown in Table 3.1, we picked the dataset produced by both encoders, given that it is the largest, for statistical analysis to better understand the dataset characteristics. (the results with other datasets are omitted for the lack of space). We found that the number of tokens for most augmented descriptions falls below 1000 tokens, in contrast to the original summary, which was below 200 tokens for most vulnerabilities. Therefore, we set the threshold for the augmented description and the summary to be 1000 and 250 tokens in our pipeline during training. We compile the augmented text and original summary's word, character, and sentence count. We found a significant difference between the augmented description and original summary (e.g., (mean, standard deviation) for word, character, and sentence in both cases: (48, 2086) vs. (49, 31), (2939, 12370) vs. (279, 186), and (43, 184) vs. (7, 5.32). This highlights the need for summarization to normalize the augmented text.

Next, we perform named entity recognition to understand which entities were presented across the summary because this is our target in the dataset. We found the following frequent named entities: (XSS, 799), (N/AC, 523), (IBM X-Force ID,463), (N/S, 343), (Cisco, 336), (SQL, 334), (Server, 315), (JavaScript, 267), (WordPress, 264), (Jenkins, 240), (IBM, 237), (Firefox, 200), (Java, 187), (VirtualBox, 174), (PHP, 164), (Java SE, 150), and (Android, 148). The common names include organizations, e.g., Cisco and IBM, technologies, e.g., JavaScript and PHP, or vulnerabilities, e.g., XSS.

We further analyze the most frequent trigram across the dataset. We found that the description trigrams are meaningful and form the basis for a good summary, unlike the augmented text trigrams, that generally do not present helpful information and appear uninformative. This might result from augmenting repeated content, highlighting specific trigrams based on frequency. However, these results emphasize the need for summarizing the augmented text.

Experimental Settings. We split the dataset with %10 reserved for testing. Then, we split the training set with %10 reserved for validation. Moreover, We train the model for 4 epochs with batch size of 8 and learning rate set to 0.0001 based on various parameters testing (results omitted for the lack of space). We use beam search as our decoding method, with a beam size of 2. We also fix several parameters: the length penalty to 8 (encouraging the model to generate longer summaries) and the repetition penalty to 2 (which instructs the model to use previously generated words). Those values are chosen based on many experiments, achieving the best results. As we stated earlier, we did extensive experimentation on the mixed dataset that uses both encoders, and based on its result, we experimented with other datasets.

Table 3.2: Results after fine-tuning the models using different hyperparameters (**R**ecall, **P**recision, *b*=number of beams, *T*=text maximum limit, *B*=batch size).

Model	R	P	F1	*T*	*b*	*B*
BART	0.51	0.50	0.49	1000	2	8
	0.51	0.46	0.47	1000	5	8
	0.52	**0.52**	**0.51**	500	2	8
	0.53	0.50	0.50	500	5	8
	0.50	0.51	0.49	500	2	4
	0.51	0.49	0.49	500	5	4
T5	0.46	0.50	0.47	500	2	8
	0.47	0.49	0.47	500	5	8
	0.47	**0.52**	**0.48**	500	2	4
	0.47	0.50	0.47	500	5	4

Computational Metrics and Results. ROUGE measures the matching n-gram between the prediction and the target. For our evaluation, we use ROUGE-1, which measures the overlapping unigram and gives an approximation of the overlap based on individual words. ROGUE consists of three sub-metrics: recall, precision, and F1-score. Recall measures the number of matching n-grams between the generated and the target summary, normalized by the number of words in

Table 3.3: Results after fine-tuning the models using different single encoder (**P**recision, **R**ecall, *b*=beams, *B*=batch).

Model	Encoder	R	P	F1	*b*	*B*
BART	USE	**0.61**	**0.60**	**0.59**	2	8
	MPNet	0.55	0.57	0.55	2	8
T5	USE	**0.58**	**0.62**	**0.59**	2	4
	MPNet	0.53	0.59	0.54	2	4

the target summary. In contrast, precision normalizes that quantity by the number of words in the generated summary. Finally, the harmonic mean or F1-score is expressed as follows:

$$F1{-}Score = 2 \times \frac{precision \times recall}{precision + recall} \tag{3.7}$$

Table 3.2 shows the ROUGE scores after fine-tuning BART and T5. We first experimented with BART because it is easy and fast to train, requiring fewer resources than T5. First, all metrics improved when the text limit shrunk to 500 tokens for the augmented text. Moreover, most metrics achieved better scores with smaller beams. This could be because as we increase the number of sequences with high beams, the risk introduced by considering the wrong sequence increases. Also, increasing the number of beams increases the time to generate a summary. Given our initial results from BART and the resource required by T5, we decided to train it on text limited to 500 tokens. However, we found that a batch size of 4 did better than 8 across all three metrics for T5. Moreover, increasing the number of beams did not improve T5 scores.

Table 3.3 shows the computational metrics for the other two datasets, which outperformed the mixed dataset. This indicates using two encoders with different architectures may harm the curated dataset. While the *USE* dataset is larger, we believe the results are better due to *USE*'s accuracy in encoding text semantics. Moreover, it proves *USE* produces a reliable representation of long

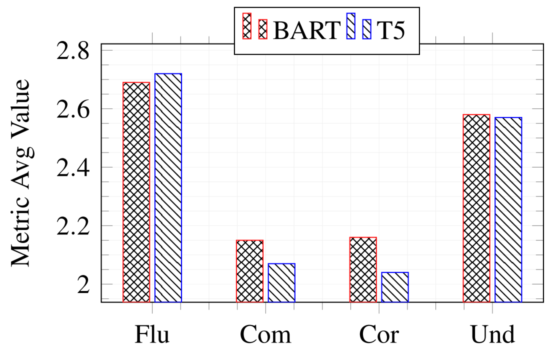

Figure 3.4: Human evaluation metric results: Fluency, Completeness, Correctness, and Understanding.

text. We reiterate that we used the DAN architecture for USE, which is less accurate than the transformer architecture.

Summary-Target Comparison: We compare the target summary with the model-generated summary using the same sentence encoders. We encode both summaries (original and new) using both encoders and measure the similarity between the target and the prediction. We found that most predictions are very close to the target, with the mean of the distribution around a similarity of 0.75, which indicates that the models are learning and generating summaries similar to the target in the embedding space of each sentence encoder.

Human Metrics Results. We consider four human metrics: fluency, correctness, completeness, and understanding. All human metrics are graded on a scale between 1-3, where 3 is the highest grade and 1 is the worse based on the metric's definition. *Fluency* measures the grammatical structure and semantical coherence of the generated summary. *Correctness* measures how accurate the model prediction is in capturing the correct vulnerability details. On the other hand, *Completeness* measures how complete is the generated summary with respect to details in the target summary. *Understanding* measures how easy it is to understand the generated summary. The evaluation is performed over 100 randomly selected samples where we report the average in Figure 3.4.

We found that both models produced fluent and generally easy-to-understand summaries with few exceptions. For example, when the generated summaries are short, they convey limited context and meaning, making them hard to understand. In contrast, completeness and correctness suffered with both models because *Zad* was not intended to excel in such tasks. Moreover, the augmented text length is not uniform across the dataset, which could be why these metrics are missing. However, those two metrics achieve good results when the augmented text is of a particular length for several instances. We can see that both models are comparable in terms of human metrics when their generated summary is compared against the corresponding target.

Qualitative Results. Both models experienced unpredictable behaviors by repeating some sentences or software multiple times or adding unrelated software to the prediction. In addition, both models tend to be extractive when the augmented text is of an average length. The length of the augmented description has a significant effect on the prediction. For instance, if the augmented text is very short (at most 20 words), both models will tend to make up summarization that was learned during training by including different sentences commonly used in vulnerability descriptions, such as gain access or code execution, even when none of these were mentioned in the augmented text. On the other hand, when the augmented description is too long, the prediction becomes repetitive and hard to understand, although it still covers different aspects of the target summary. Another behavior that we observed was using opposite adjectives or missing the software names despite being capitalized in the augmented text. Most of the explained deficiencies are present in both models with varying degrees. One possible solution is to ensure diversity among the augmented sentences and guarantee that no sentence is repeated.

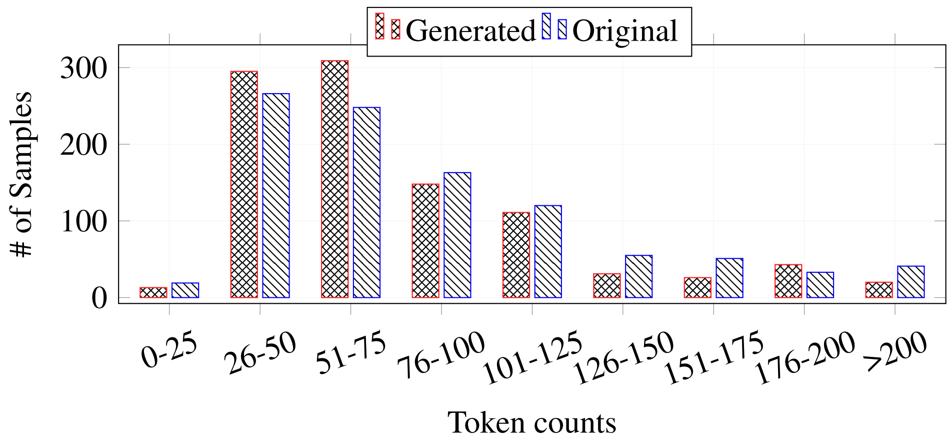

Figure 3.5: Tokens distribution with instances and their token counts in the original and generated summaries.

Analysis of Summary Contents

In this section, we perform additional post-analysis on the generated summary vs. the original (target) summary to understand the effect of our approach. For this analysis, we selected BART-generated summaries trained on the USE dataset because the model achieved the best results in terms of computational metrics. We did not rely on human metrics because (1) they are subjective, and (2) the evaluation considered only 100 samples.

First, we expected limiting the generated summary length to 250 tokens during fine-tuning should generate a uniform summary, reaching the maximum limit for most instances, considering the lengthy augmented text. However, most generated summaries were significantly below 250 tokens, as shown in Figure 3.5. we see in Figure 3.5 that the number of samples with token counts between 26-75 has increased, but as the number of tokens starts to increase, the number of instances decreases in the generated summary, even more than the original summary except for token count between 176-200. Moreover, summary statistics for the generated summary (mean and std) were also lower than the original summary, which is against our expectations.

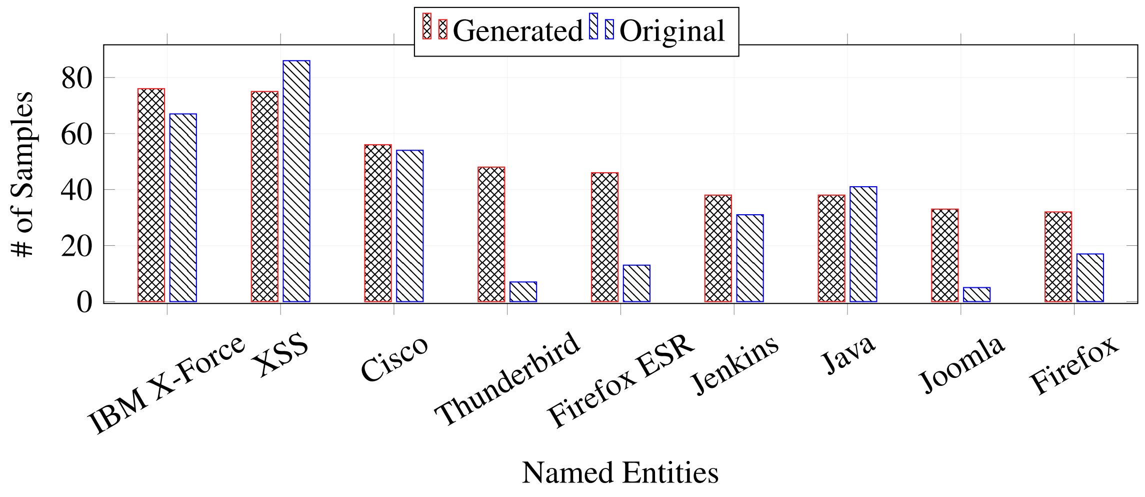

Figure 3.6: Top names presented in the original and generated summary.

Next, we analyze the named entity recognition shown in Figure 3.6. We see that many entity names appear in both summaries. However, their counts did not exactly align. XSS appeared to be the most repeated name in the original summary but came second for the generated summary. Moreover, we noticed some names appeared many times in the generated summary but were not present in the original summary, such as Thunderbird &, Firefox ESR & lt, which could indicate overfitting. This result corroborates what was reported in our human evaluation, that some predictions included software not mentioned in the original summary.

Finally, we analyze the trigrams shown in Figure 3.7 and Table 3.4. First, the model produces a meaningful and informative trigram like the original summary. However, the trend is evident for every trigram; the generated summary outnumbered the original. Most of the above results indicate the model is overfitting as it memorizes what it learned during training and uses them for inferences, even when unrelated. One justification could be attributed to the dataset quality as the augmented text includes repeated and lengthy content, triggering this behavior.

Our results and analysis suggest additional work is needed to improve the generated summaries. We also notice that bigger models do not necessarily perform better, especially for the curated

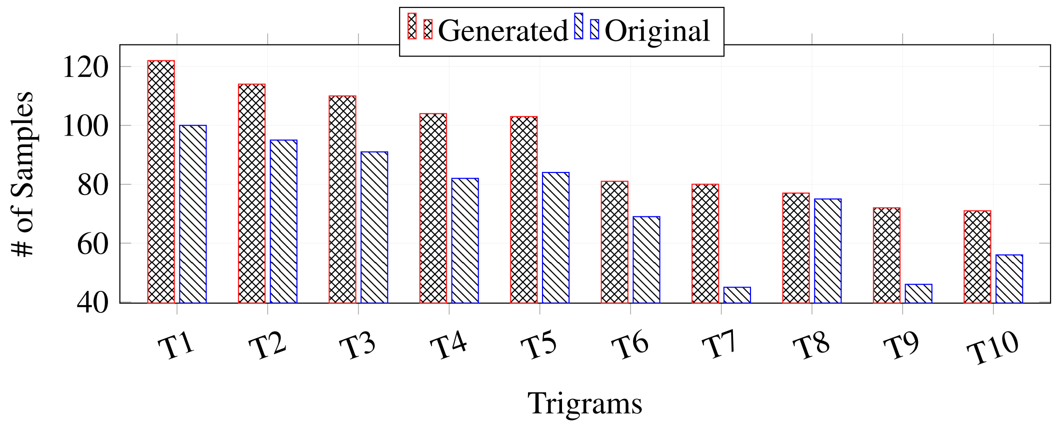

Figure 3.7: Trigrams count. The generated summary has the same trigram repeated more than the original, indicating overfitting.

Table 3.4: The full sets of the different trigrams in Figure 3.7 resulting from our model; original and generated.

Attribute	Value
T1	could allow attacker
T2	attacker could exploit
T3	could exploit vulnerability
T4	successful exploit could
T5	exploit could allow
T6	execute arbitrary code
T7	attacker execute arbitrary
T8	IBM X-Force id
T9	exploit vulnerability sending
T10	supported versions affected

dataset that targets a specific domain. In addition, we focused heavily on hyperparameters fine-tuning to achieve the best result without considering the dataset itself. Overfitting is typically noticed when the training loss decreases and the validation loss increases. However, during fine-tuning, validation loss kept decreasing while the training loss fluctuated, which is expected given the augmented text is not uniform in length across the dataset. In our case, we detected overfitting through human evaluation and post-statistical analysis.

Table 3.5: Results after fine-tuning the models using the label-guided and the description-guided methods (**P**recision, **R**ecall, *B*=batch).

Model	Dataset	R	P	F1	*B*
BART	Ground-truth	**0.59**	**0.55**	**0.54**	4
	Description	0.36	0.35	0.33	4
	Ground-truth	**0.61**	**0.55**	**0.55**	8
	Description	0.32	0.33	0.29	8
T5	Ground-truth	**0.49**	**0.58**	**0.51**	4
	Description	0.30	0.32	0.30	4

Analysis of Label Guided Dataset

In this section, we report our results using the label-guided dataset described in section 3. To make our evaluation concrete, we fine-tune both models using the same 100 samples, but with the original summary as the target to compare both approaches. We notice that the label-guided approach outperformed the description-guided, considering computational metrics as shown in Table 3.5. We have foreseen these results since our ground-truth summaries are created from the augmented text, confirming that using the description as a summary is imperfect. We hypothesize this is due to the difference in length and content between the augmented text and the description. In contrast, our summaries provide better and longer context to the model, generating coherent and plausible summaries.

For the human metrics in Figure 3.8, both models' fluency and completeness scores were relatively higher than correctness and understanding scores, especially for T5. Moreover, we found that both models generated sound and fluent summaries, including most details from the target summaries. In contrast, correctness is the most affected metric as the details included were incorrect (e.g., wrong software version or bug description) from the created summary. The understanding was problematic because the generated summary needed more context, or sentences were sparse.

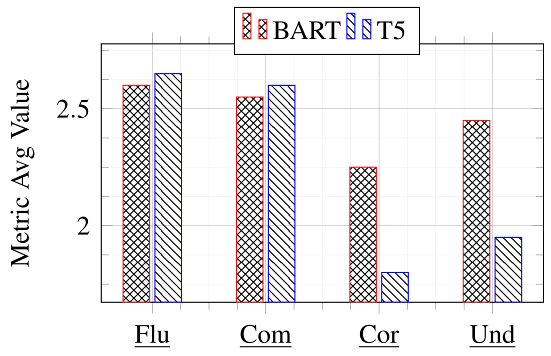

Figure 3.8: The human evaluation metric results using the label-guided approach: Fluency, Completeness, Correctness, and Understanding.

We make several notes about both models concerning their generated summaries. First, both models generated summaries extracted from the augmented text, selecting the salience span of text for inclusion, but they did not concur with the hand-crafted summaries. For example, software names included in the label were missing in multiple instances. Moreover, T5 created a summary for a few instances by copying the first few spans of text from the augmented text, which was fluent but hard to understand. Although the results of this approach are limited, they still indicate that utilizing the description as a summary is insufficient.

Enhancing Augmented Text Quality

As our pipeline aims to collect text from various sources that only appear similar in an embedding space, we rely heavily on the sentence encoder to determine whether a text should be included. However, those hyperlinks may lead us to different pages with the same paragraph, but the pipeline will still include it in the dataset. As a result, the augmented text for some entries will be excessively long and repetitive. To address this issue, we enhance the augmented text by enforcing diversity using word frequency. We hypothesize that the low score obtained upon evaluating the original dataset was due to the augmented text quality, which justifies our approach. We assert that

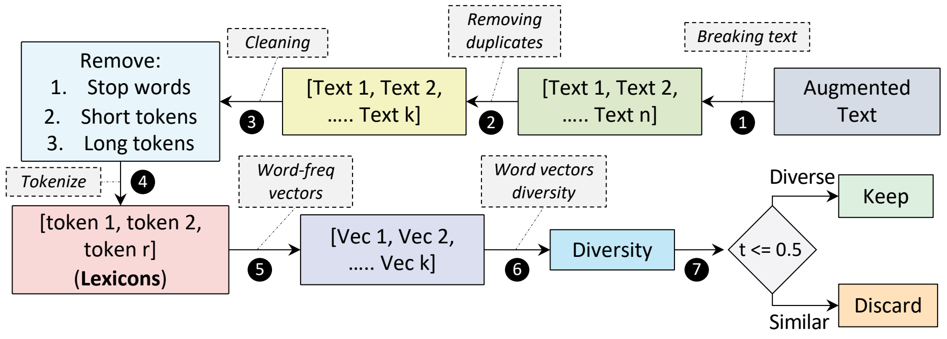

Figure 3.9: Dataset quality enhancement pipeline, added as an extension to *Zad*. The pipeline starts with the augmented text of each instance and breaks it into a group of sentences, followed by a duplicate removal pipeline. Text cleaning considers removing ineffective phrases followed by tokenization to construct the lexicons set. Finally, word-frequency vectors are created for each sentence, and the diversity between sentences is measured using a predefined threshold.

reducing this text into a few sentences that capture most of the augmented text could benefit the model.

Word Frequency. *Word count* describes a document as a vector of its words frequency. First, we find unique words or lexicons across all documents in the dataset, then represent the document as a vector of key-value pairs such that the key is the word and the value is its frequency normalized by the total number of occurrences across all documents, indicating the probability of that word. Using these vectors, we compute the cosine similarity of two documents based on their words and associated frequency.

Methodology. We perform multiple preprocessing steps before constructing the frequency vector for each sentence, as shown in Figure 3.9. First, we break the augmented text into sentences, considering a period followed by a space as a separator. We then remove duplicate sentences in the augmented text using USE. We use a pair-wise similarity to compare each sentence with every other sentence and discard the sentence with similarity of ≥ 0.98—the relaxation in the similarity is to accommodate for minor changes. We remove stop words, short tokens with less than three characters, and tokens with more than 20 characters. Long tokens primarily represent a sequence of characters that rarely appear in a sentence. We remove them to reduce the dimensionality of the vector space and include words that commonly appear in the augmented text of a vulnerability. After cleaning the text, we tokenize every sentence and store all tokens in a set representing the augmented text's lexicons, which are used to build a vector for each sentence.

The diversity of two sentences is determined using the cosine similarity between their vectors. A threshold of ≤ 0.5 is used for two sentences that share half of the words (at most), and contribute less to the diversity, thus excluded. As we move to a new sentence, we compare it against every sentence in the sentence set and include it only if the diversity is satisfied against all sentences in the sentence set. The final sentences in the set are used as the new augmented text. At this stage, we need to return those unique sentences to their original form with any words removed during cleaning. Finally, we concatenate them into one long text, resembling the new augmented text. We use a period as a separator when we concatenate the sentences to indicate the end of each one and update our dataset by overwriting the old augmented text with the new one. We applied the pipeline in Figure 3.9 with two settings. The first one was to deploy it on the augmented text without capping, meaning the augmented text could be of any length as long it satisfies the pipeline conditions. The second was with capping, where we cap the augmented text's length to 250 words. We chose 250 words because more than 83% and 90% of the augmented text in the USE and MPNet datasets contained ≤ 250 words.

Analysis of Quality Enhancement Pipeline

Considering our initial results on the three datasets, we applied our enhancement pipeline to the USE dataset because it achieved the best results. The dataset is fed into the pipeline, as shown in Figure 3.9, and processed accordingly. Capping, by limiting the length to 250 words in the

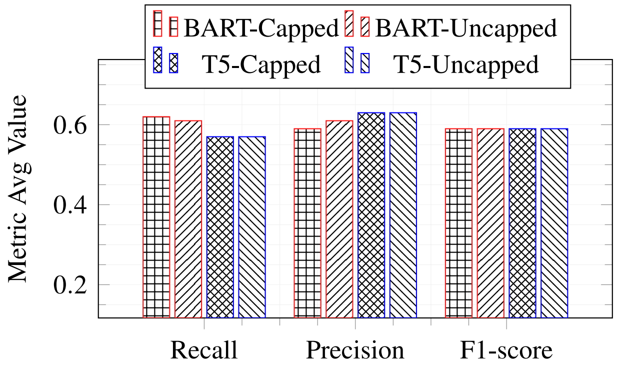

Figure 3.10: Results after fine-tuning the models after passing the USE dataset through the quality enhancement pipeline. The number of beams used across all settings is 2, and the batch size for BART is 8, while for T5 is 4.

augmented text, is applied after ensuring the diversity of a sentence. In particular, we check each sentence length to ensure the total length is ≤ 250 words. Our goal of capping is to find out whether limiting the length will help the model. After passing the USE dataset through the pipeline, we obtain two new datasets. Due to our pipeline in Figure 3.9, which ensures preserving a set of diverse sentences, both datasets' sizes have shrunk by almost 50% of their original size. We used the same setting reported when using the USE dataset as shown in Figure 3.3.

The results for both models and datasets are shown in Figure 3.10. First, we observed that the results for both datasets are identical for T5, which is almost identical to the results in Figure 3.3. In contrast, BART showed a slight improvement when applied to the capped dataset for the recall score but not for the precision. However, in terms of computational metrics, the improved dataset does not seem to improve the result, although the dataset has shrunk by almost 50%. This could indicate that transformer models can detect similar texts and emphasize less attention when fine-tuning specific tasks, explaining why the reduced-size dataset produced the same results as the original dataset. This might be the case since we know the pre-trained language models are trained on a massive amount of text that may include duplicates or highly similar entries. Moreover, the diversity pipeline modifies the flow of sentences in the augmented text, which may make the text

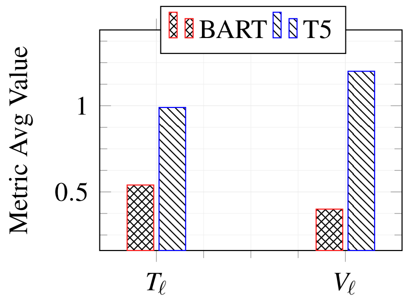

Figure 3.11: Models training T_ℓ and validation loss V_ℓ. The batch size for BART is 8, while for T5 is 4.

incoherent. However, we notice that the validation loss for both models has decreased. Moreover, the training loss has decreased for T5, as shown in Figure 3.11.

The training and validation loss dropped, proving the quality enhancement pipeline improved the dataset, and its effect can be seen in training. For BART, the validation loss decreased from 0.46 to 0.40. For T5, the training and validation loss decreased from 2.35 and 1.46 to 0.99 and 1.20, respectively.

Relying on the ROUGE score is insufficient since it measures the overlap only. The problem is that the old score (original datasets) uses more text, which forces the model to produce a summary that constitutes a wide variety of words that will most likely overlap with the target summary. In contrast, reducing the size of the dataset and maintaining the same score is promising because reducing the data typically impacts the results negatively. Therefore, we analyzed a randomly selected group of 100 instances to evaluate the summary's quality. We noticed the quality improved compared to the generated summary from the initial dataset. For example, the repetitive behavior was rarely seen (about seven samples) in the newly generated summary and was mainly aligned with the target summary in content with mostly consistent length.

The model learned to map some concepts; for example, the target summary included XSS, but the generated summary uses cross-site scripting, also known as XSS. Moreover, the generated summary is more relevant to the target concerning the software name but is still inaccurate. However, the generated summary still suffers from adding unrelated software to the summary but not as much as it was with the original dataset. Overall, our enhancement pipeline has improved the dataset's quality by reducing its size and the loss for training and validation, as well as the quality of the generated summaries. However, we believe additional steps are needed to address some deficiencies in the created dataset. For example, ensuring the content is closely related and constitutes useful detail to the vulnerability may require a more sophisticated approach instead of simple similarity criteria, especially since we are conducting our scraping blindly from third-party resources.

User study

Our user study assesses *Zad's* effectiveness. The study presented participants with the original description of a vulnerability and the enriched description generated by *Zad*. The participants were asked to rate the enriched description on enrichment, accuracy, and understanding. Each criterion was rated on a scale of 1 (worst) to 3 (best).

Enrichment refers to the extent to which the enriched description improved upon the original by adding more context and related information. A rating of 3 was given if the generated description included more details than the original, two if the two descriptions were relatively the same, and one if the enriched description had less content than the original.

Accuracy measures the ability of the enriched description to capture the software and bug information in the original description. A rating of 3 was given if the generated description included all the details, two if it missed at most two pieces of information, and one if it missed most details.

Understanding refers to the user's ability to comprehend the text's content and meaning. A rating of 3 was given if the meaning could be easily understood, two if it was more difficult but still possible, and one if it could not be understood.

For this experiment, we selected the most effective model, BART, along with the USE dataset. We randomly chose 100 samples of enriched descriptions generated by the model and their corresponding original descriptions. Due to the various constraints in recruiting participants and the required resources, we limited our experiment to five participants (evaluators), who are all familiar with this domain, including extensive knowledge and experience in threat intelligence and security operations.

The results showed consistency among evaluators. This can be seen particularly in enrichment and understanding, where *Zad* produces enriched and easy-to-understand descriptions. The average score for enrichment ranges from 2.53 to 2.60, with a median of 2.55, showing an agreement that the generated summary provides more context and details than the original. The extended vulnerability descriptions boosted the understanding score, ranging from 2.70 to 2.87, with a median of 2.78. The correlation between the two metrics is evident and consistent, proving that *Zad* significantly impacts improving vulnerability content. In contrast, accuracy fluctuated be-

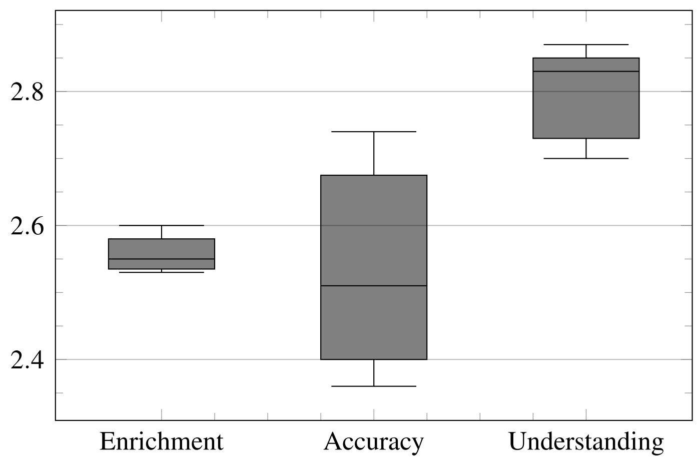

Figure 3.12: User study evaluation over 100 samples generated by BART using the USE dataset with five evaluators.

tween evaluators, averaging from 2.36 to 2.74, with a median of 2.53, demonstrating the downside of *Zad*. Identifying various vulnerability details requires support from another model to capture software and bug details. However, *Zad* was developed mainly to address the vulnerability descriptions' sufficiency. Therefore, *Zad* accuracy needs additional enhancement to handle specific details associated with a vulnerability.

Summary of Completed Work

In this work, we proposed *Zad*, a multi-staged pipeline toward addressing shortcomings in vulnerability description presented in public databases, which is typically short and lacks sufficient context. *Zad* introduces a solution by utilizing third-party reports and use them to contextualize the original description and include additional details that will aid developers and organizations in fixing their vulnerabilities. *Zad* scrape the content of such reports and find paragraphs that are semantically similar to a vulnerability description in the embedding space. The scraped text is cleaned and preprocessed to create the augmented text, resulting from concatenating the preprocessed text. The second stage of *Zad* is deploying a pre-trained model to frame this as a summarization problem, namely, summarizing the augmented text to the description. Moreover, we approach this problem through manual labeling by summarizing 100 samples from the collected dataset. Our initial results are promising, showing that *Zad* can generate summaries from the augmented text. However, the label-guided approach achieved better results in terms of computational and human metrics. Moreover, our user study confirms that *Zad* can indeed produce an enriched summary with comprehensive and easy-to-understand content.

CHAPTER 4: *MUJAZ*: A SUMMARIZATION-BASED APPROACH FOR NORMALIZED VULNERABILITY DESCRIPTION

The information in CVE/NVD varies significantly from the information in third-party reports for the same vulnerabilities [39]. Moreover, the vulnerability information in CVE/NVD could be incorrect, incomplete, or outdated, and those issues could happen for various reasons and circumstances [21, 20]. For instance, some vulnerabilities may be reanalyzed by their source contributors or updated, introducing this inconsistency with CVE/NVD. As the number of discovered and disclosed vulnerabilities increases over time, manually addressing those inconsistencies becomes significantly impractical, necessitating the development of an automated solution to unify vulnerabilities' attributes in a single, concise, and accurate context without any conflicts [79].

Organizations addressed inconsistency issues by creating their repositories for documenting vulnerabilities [6, 4]. However, they are primarily concerned with their products and hardly address other vulnerabilities in other products, particularly those that do not have such initiatives, let alone accurately. The disparity of details in a vulnerability description differs significantly among those private repositories.

Microsoft, for example, has Microsoft Security Response Center (MSRC) [6], which does not provide any description for some vulnerabilities and only offers a high-level title as the description, e.g., the description for *CVE-2021-43883* is "Windows Installer Elevation of Privilege Vulnerability". IBM X-Force Exchange [4], a cloud-based platform that monitors and tracks the latest threats and vulnerabilities, provides comprehensive information about vulnerabilities and their attributes. However, X-Force Exchange has various serious discrepancies with other databases in the description and vulnerability scoring, making it impossible to use alongside other databases. For instance, NVD assigns *CVE-2018-9116* a CVSS score of 9.1, while X-Force Exchange assigns 6.5 to the same vulnerability.

In the academic literature, researchers addressed the inconsistency issues in vulnerability reports by analyses, understanding, and mitigation using various Natural Language Processing (NLP) techniques. For instance, Dong *et al.* [39] introduced a system that detects inconsistencies between NVD/CVE and third-party reports. However, their dataset lacks vulnerability type representation and is biased toward memory corruption vulnerabilities. Similarly, Kühn *et al.* [62] developed a system to update the NVD database and improve the Information Quality (IQ) using structural data, such as name tags in the description and the CVSS score. However, their system only utilizes information within the vulnerability itself, limiting the improvement the system can achieve.

The cumulative discrepancies in the reported vulnerabilities cost security analysts significantly in terms of their man-hours for understanding the vulnerability reports, developing patches, and deploying them. Given the gap in addressing vulnerability reporting inconsistency, we propose a new approach, called *Mujaz*, focusing on the vulnerability description as a source for unified vulnerability reports.

Our Approach. *Mujaz* addresses the inconsistency by industry- and vulnerability type-independent summarization-based technique that varies the underlying learning objectives to achieve a highly accurate, consistent, and normalized vulnerability description. To evaluate *Mujaz*, we curate a *parallel* dataset that consists of three features extracted from the description. Each feature can be viewed as a task to train a multi-task model independently or jointly with another feature to produce the target summary. The rich semantics of a vulnerability description necessitates curating such a dataset because it typically includes the software name/version, the type of bug/threat, and a summary describing their interactions. Thus, the dataset's quality is essential since each instance must comprise the three features with sufficient details.

To improve the dataset quality, we offer a multi-task model to *abstractively* summarize and normalize CVE description. Multiple system variations are developed and tested to attend to the information necessary for the summary while optimizing two objectives: ① normalizing the resulting description and ② shortening the original description in the resulting one.

To achieve both objectives, we build a generic pipeline based on the transformer architecture that utilizes our curated dataset and deploys its features to generate the summary. Each feature represents a task that our multi-task model can learn and predict. Moreover, two or more tasks can be learned independently, and their predictions can be combined to produce a new summary, providing more deployment options. To the best of our knowledge, our work is the first to curate a parallel dataset from a vulnerability database and deploy it to produce a new standard and uniform vulnerability description.

Mujaz operates on a dataset that contains the original CVE description and three manually created and vetted features: a normalized summary (SUM), software name & version information (SWV), and the details of the bug itself (BUG). Moreover, our dataset only considers descriptions from reliable sources, excluding any third-party reports due to the inconsistencies mentioned earlier. Utilizing public and official sources such as CVE/NVD ensures reliability and accessibility. We propose *Mujaz*, a system that generates a uniform and normalized summary for a vulnerability regardless of the underlying structure or quality of the original description by attending to particular components of the description and utilizing them in a self-contained manner. The customizable nature of *Mujaz* allows us to deploy it for various tasks with datasets and objectives similar to ours.

We evaluate *Mujaz* using traditional metrics, measuring the overlap between the ground truth and the generated summary. However, missing or including the wrong software names/versions or the type of bug behind a vulnerability could be intolerable. To address this issue, we present our human metrics to measure the accuracy and completeness of the generated summary in relation to software bug information. Moreover, we present other metrics to evaluate the generated content to quantify distinctive aspects concerning human understanding and summary normalization. Our proposed human metrics extend the evaluation rigor, ensuring the quality of the generated target summary.

Design Challenges

While our problem statement is relatively simple, addressing it technically is challenging. In the following, we set up our design by enumerating the challenges we address.

Challenge 1: Dimensionality. As we discussed, *Mujaz* aims to curate multiple features from the original description. Utilizing such features for summarization require the underlying model to support dimensional data. Therefore, we propose deploying a multi-task model with the ability to learn different tasks simultaneously based on the selected dimension (feature), which enables the model to be self-contained and comprehensive when combining the output of two different tasks. Our multi-task model adjusts its parameters to integrate multiple tasks within the same model, giving it the ability to generate a concise and informative summary.

Challenge 2: Domain-specific Language. Our model needs to support domain-specific language (i.e., security) by the appropriate encoding and decoding. Given the nature of our summarization task, the model must constitute an encoder to represent the input and a decoder to produce the output. Unfortunately, vulnerability descriptions are very limited in scope and content, typically including 1-3 lines of text.

Training a model for vulnerability summarization will require a massive dataset from various sources for accurate output. Under realistic settings, the model's linguistic capacity will be fixed and limited for generating a summary with limited labeled data for model training. On the other hand, pretrained language models are already trained on massive textual data and provide an excellent alternative to bootstrap our model. Fine-tuning such a pretrained model is orders of magnitude simpler than training a model from scratch and works by adjusting the model's weights on a labeled dataset for the chosen task. Fine-tuning is convenient, fast, and demands much fewer resources than training a model from scratch. We exploit pretrained language models using our parallel dataset for constructing a domain-specific language model.

Challenge 3: Consistency and Evaluation. *Mujaz* aims to summarize vulnerability descriptions in a consistent and accurate manner. We approach this issue systematically for a unified structure that incorporates critical information regardless of the original description organization. We, however, note that the existing summarization evaluation metrics are limited to term overlap between the original text and the generated summary. Given the nature of our dataset, such metrics are insufficient since the input and target texts are short, and the overlap is expected to be high.

Vulnerability description also contains important aspects which render their accuracy paramount. As such, we propose human metrics to quantify the technical accuracy, such as correctness and completeness. We also devise additional metrics to judge the linguistic aspects, e.g., fluency and understanding, which cannot be measured using conventional methods.

Mujaz's Pipeline

Mujaz follows a conventional architecture of an encoder/decoder-based transformer, as shown in Figure 4.1. The pipeline depicted consists of two independent phases illustrating our design.

The first phase represents our parallel dataset curation with an example, showing how a CVE description is broken into three features as in ❶. In this phase, *Mujaz* integrates several preprocessing techniques to ensure the high quality of the input text and the extracted features. The second phase

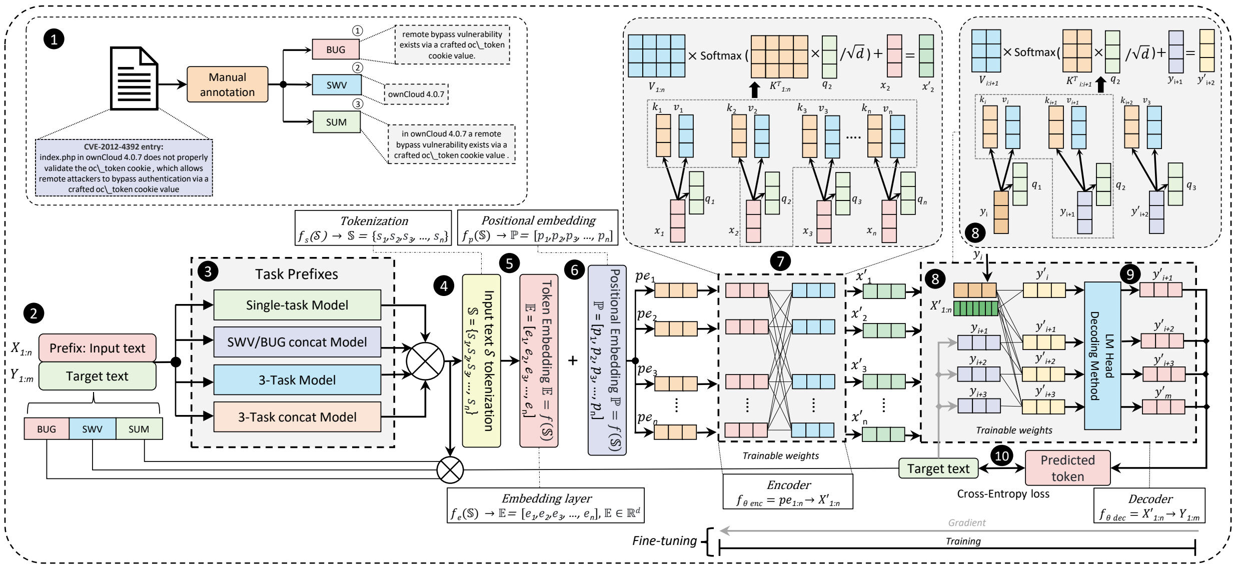
Manual annotation
BUG
SWV
SUM
remote bypass vulnerability exists via a crafted oc_token cookie value.
ownCloud 4.0.7
in ownCloud 4.0.7 a remote bypass vulnerability exists via a crafted oc_token cookie value .
CVE-2012-4392 entry: index.php in ownCloud 4.0.7 does not properly validate the oc_token cookie , which allows remote attackers to bypass authentication via a crafted oc_token cookie value
Tokenization
Positional embedding
Task Prefixes
Single-task Model
SWV/BUG concat Model
3-Task Model
3-Task concat Model
Prefix: Input text
Target text
Input text S tokenization
Token Embedding
Positional Embedding
Embedding layer
Trainable weights
Encoder
Decoder
LM Head Decoding Method
Target text
Predicted token
Cross-Entropy loss
Gradient
Training
Fine-tuning
× Softmax

Figure 4.1: Mujaz Multi-task Pipeline: The first phase includes dataset curation to build the prospective dataset. The second phase shows the input text represented as a CVE description and one of the extracted features to trigger task-specific training, allowing a model to learn multiple tasks simultaneously.

includes the entire pipeline from ❷ to ❿, depicting a multi-task transformer model, task specification, multiple embedding layers, an encoder, and a decoder, which we review in the following.

The Seq2Seq model, in general, expects two sequences, representing an input text of size n and a target text of size m, as shown in step ❷. However, the target text is represented by three distinct features, each with a corresponding prefix, which designates the task for mapping $X_{1:n} \rightarrow Y_{1:m}$. In step ❸, the selected prefix will pass the input text to step ❹ and allow the matching feature to be passed as the target text $Y_{1:m}$ for training in step ❿. The input text is then passed to the tokenizer in step ❹, breaking a text into a set of tokens followed by the embedding layer in step ❺, projecting tokens into d dimensional space to be used by the transformer.

The positional embedding in step ❻ gives an embedding of d dimensions to maintain a token position within the sequence. The final embedding is produced by adding both embeddings from steps ❺ and ❻, passed to the encoder in step ❼. The encoder consists of N layers, deploying bi-directional self-attention to produce the encoded sequence $X'_{1:n}$.

The decoder in ❽ utilizes the encoded sequence $X'_{1:n}$ to produce a probability distribution over the entire vocabulary used by the LM head in step ❾, which consolidates of various decoding methods to generate the most probable token. The decoder uses uni-directional self-attention to prevent it from looking into the next token during training.

In step ❿, the predicted token is used with the target text from step ❸ to compute the loss using the cross-entropy. The training is accomplished through teacher forcing, indicated with the gray arrows, such that the predicted token is used to compute the loss, and the correct token is fed back to the decoder for the next token prediction.

Pipeline: Technical Details

For convenience, we describe the internal architecture of the original Transformer presented in [102] within our pipeline.

Architecture Overview. The transformer is depicted in Figure 4.1 and consists of two components: an encoder and a decoder. The encoder transforms a sequence into a representation that captures the relationship between tokens, while the decoder utilizes the encoded representation to perform the summarization task. Our pipeline in Figure 4.1 reflects a multi-task model with T5, although we will highlight the differences between that and BART in each step of the pipeline.

Preprocessing. The first step in our pipeline is curating the dataset using original CVE entries. As depicted in step ❶ in Figure 4.1, the annotation process starts by having a CVE description manually annotated to extract three features, the BUG, SWV, and SUM, which we describe in the following.

The BUG feature in ① is a summarization and normalization of the description that identifies the type of bug and how it could be exploited. The key information consolidated here is the vulnerability type and a summarized version of the affected interfaces or functions. A CVE description of a library could include the library name (e.g., `myLibrary 4.1.2`) and affected headers (e.g., `myFunction` in `myHeader.h`). `myLibrary 4.1.2`, being the software with the version, would be included as the SWV feature. In contrast, `myFunction` in `myHeader.h`, being part of how the vulnerability is exploited, is part of the BUG feature.

The SWV feature in ② is a list of vulnerable software and their versions present in the CVE description. This information could be spread throughout the description, and we consolidate it and make it semi-uniform. The relevant software is represented in this feature as a list with its affected versions followed by other affected software and versions.

The SUM feature in ③ is a grammatical concatenation of the SWV and BUG features with additional context and wording in an attempt to improve the reader's understanding. This grammatical connection is an attempt to provide uniformity to the summarized SWV and BUG features. The target structure of a summary is "*in SWV (vulnerability types—BUG) are present via (method of exploitation—BUG)*". Where there is a hierarchy of vulnerabilities *i.e., software A in software B is vulnerable*, the SUM feature represents this case by listing the software in order at the beginning of the summary.

In step ❷, the input text in the original CVE entry is preceded by a prefix, indicating a feature the model has to learn. Moreover, the target text comprises the three features, functioning as a target text based on the appended prefix. In ❸, the prefix determines the task and target text used for training and passes the input with the prefix to the next step, tokenization. The detail of each task is explained thoroughly in section 4.

Tokenization. Our tokenizer in step ❹ breaks the input sequence into a set of tokens. Most pretrained models deploy sub-word tokenization, which decomposes a token into meaningful sub-word units that appear within other tokens. Sub-word tokenizers are trained separately on a corpus to learn the best set of characters representing most words within a corpus. The trained tokenizer is then integrated into the model to perform tokenization. BART uses Byte Pair Encoding [93], while T5 uses SentencePiece [61]; both sub-word tokenizers.

The tokenizer performs other related tasks required by the model, such as adding the beginning and ending symbols <START> and <EOS>, indicating the start and end of the sequence, respectively. Also, arranging a series of tokens into a fixed-length sequence such that a batch constitutes of multiple sequences of the same length.

Embedding. The embedding layer in step ❺ projects a token into a vector space of a certain dimension that can be fed into a neural network. Each token is represented with a vector of dimension d such that a token $x \in \mathbb{R}^d$. Token embedding could be initialized with random values or using precomputed embedding like Word2Vec [72, 73] or Glove [84]. In both cases, the embedding will improve and gets updated according to the training dataset. The vector's dimensionality is a hyperparameter, typically set to 512.

Positional Embedding. Tokens embeddings do not encode any positional information into their representations. In step ❻, each model deploys its own method for positional embedding. BART uses absolute positional embeddings and assigns a vector of size d representing the tokens' order within the sequence. This method learns the positional embedding as part of training and does not require any function. However, this method imposes an upper bound on the sequence length.

During the inference stage, a sequence longer than the maximum allowed length will not be encoded properly, limiting the model performance. To address this issue, T5 utilizes a relative positional encoding to capture the relationship between tokens. The importance of a token is determined by its relation to other tokens within the sequence. Tokens within a sequence are depicted as nodes in a graph, with edges connecting them, quantifying their association using the attention mechanism within the encoder.

The token embedding in step ❺ and positional embedding in step ❻ are added together to form the final embedding. Therefore, the positional embedding must have the same dimensionality as the token embedding. The resulting sequence representation is passed to the encoder.

Encoder. The encoder in step ❼ is composed of two major components: (1) multi-headed attention and (2) feed-forward network. The encoder is fed the embedded sequence $PE_{1:n}$ to produce the encoded sequences $X'_{1:n}$.

Multi-Headed Attention. Self-attention is a sequence-to-sequence operation that relies on the dot-product to capture the attention surrounding token x_i. A token x_i is multiplied by every other token in the sequence including itself to produce attention scores, which are passed through a softmax layer to produce a probability distribution over tokens' scores, summing up to 1. The scores are multiplied against their respective embedding, followed by a linear combination to obtain the final representation y_i. The following equation can represent the self-attention:

$$y_i = \sum_j \text{softmax}({x_i}^T x_j) x_j, \tag{4.1}$$

where i is the token's index for which the embedding is computed, and j is the index of the tokens within the i-th sequence.

A single self-attention layer is referred to as a "head". Multi-headed attention is obtained by deploying multiple heads for the same sequence to capture various semantic characteristics. The final output of each head is concatenated and passed through a linear transformation to construct the final embedding for each token in the sequence. The encoder in step ❼ uses a bi-directional self-attention, meaning a token incorporates the context from both sides of the sequence.

Query, Key, and Value. The transformer improves the self-attention mechanism using the concepts of query, key, and value. Each token in the sequence is passed through three linear transformations. Each transformation introduces a weight matrix that is optimized during training to fit its role.

The outputs produced by these transformations are denoted as query q, key k, and value v of a token, all with the same dimensionality $d = 512$. Each query q of a token is matched against every other token's key in the sequence. For optimization, q_i is multiplied by $K_{1:n}^T$, producing similarity scores. We down-scaled these scores by the square root of the embedding dimension d to prevent the softmax output from growing too large, which may slow down training or vanish the gradient.

The scaled scores are passed through the softmax layer, producing a probability distribution. We perform a weighted sum of the value vectors v for each token in the sequence corresponding to the key vector. The weighted sum is the new token embedding capturing attention with every other token.

The novel attention mechanism is expressed as:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{4.2}$$

The encoder consists of N layers such that the output of one layer is fed to the next one. The last encoder will produce the final sequence embedding passed to the decoder in step ❽.

Residual Connection and Normalization Layers. Both of these practical considerations are commonly deployed in deep learning to improve tensor representation and speed up training. The residual connection adds the input embedding to the embedding produced by multi-headed attention and avoids the information loss that might happen in deep neural networks. Consequently, this design choice solves the vanishing gradient problem that may arise due to the softmax layer. Moreover, the normalization layer adjusts each input to have 0 mean and 1 variance, which center the data around the origin point. The normalization layer is placed after each sub-component, multi-headed attention, and the feed-forward network. This design choice stabilizes the gradient descent step, allowing the use of a larger learning rate.

Feed-Forward Network (FFN). FFN consists of two fully connected layers with non-linear activation, typically, Rectified Linear Unit (ReLU) or Gaussian Error Linear Unit (GELU), to enhance the expressiveness of the token's embedding. The FFN is referred to as position-wise FFN because Every token embedding is passed independently through it.

Decoder. The decoder's objective in step ❽ is to learn the parameters θ of the function f, which maps the encoded sequence into the target sequence. Formally, $f_{\theta \text{ dec}} = X'_{1:n} \rightarrow Y_{1:m}$, utilizing the sequence representation built by the encoder to auto-regressively generate the most plausible token for the target sequence based on the task.

The encoded sequence $X'_{1:n}$ is fed into the decoder with a special input token y_i, indicating the start of the sequence. The decoder uses self-attention to produce y'_i, which is fed into the language model head in step ❽, responsible for selecting the most probable token.

In step ❾, the selected token produced by the language model head is then used to compute the loss against the corresponding token from the target text passed from step ❷. Formally, the cross-entropy loss for n classes is defined as:

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{n} y_j \log \hat{y}_j. \tag{4.3}$$

Instead of passing the predicted token to the decoder for the generation of the next token, the token from the target text is passed, also known as teacher forcing [108]. The decoder uses uni-directional self-attention, conditioning the new output vectors on the encoded sequence and any previously generated token, blocking the visibility of tokens from the target text.

Decoding Method. The language model head in step ❽ relies on generating a token giving a sequence of words using softmax over the vocabulary, as shown in the following:

$$P(x_i|x_{1:i-1}) = \frac{exp(u_i)}{\sum_j exp(u_j)}. \tag{4.4}$$

The generation of the next token depends on the chosen generation method. In essence, the prediction of the next word follows a probability distribution over the entire vocabulary set. The greedy search method selects the token with the highest probability, thus reducing the hypothesis space for the entire sequence and producing a repetitive model. To this end, we use the beam search, top-k sampling, and top nucleus.

Beam Search. The beam search method [97] extends the hypothesis space to include longer sequences at each timestep and select the sequence that produces the highest probability for that sequence. The number of beams is a hyperparameter that defines how far the model should look behind a token to compute the probability before choosing the next token.

Top-k. Top-k sampling selects K tokens with the highest probability across the vocabulary and picks one word randomly from such a group. The number K is a hyperparameter.

Top Nucleus. This sampling method [50] chooses a small set of tokens whose cumulative probability exceeds a predefined probability p, which is a hyperparameter; this is:

$$\sum_{x \in V^{(p)}} P(x_i | x_{1:i-1}) \geq p. \tag{4.5}$$

All in all, the sampling methods aim to restrict the number of tokens that a model can sample from at each time step.

Multi-task Model

The pretrained models are a central component in our pipeline to alleviate the need for a large amount of data in our summarization task. Fine-tuning those models, however, is essential to customize for domain-specific summarization. In the following section, we introduce the details of our fine-tuning steps of the T5 and BART models. We start with T5 by designating our normalized summarization task with the label "SUM", so the input provided to the model would be "SUM: INPUT" to designate it as a SUM task.

Our pipeline, as shown in Figure 4.1, is a multi-step process consisting of feature creation, tokenization, encoding, learning, and decoding to produce an abstractive summary task of the input. The three tasks are BUG, SWV, and SUM.

- The BUG task seeks to produce the description of the software vulnerability given a CVE description.
- The SWV task seeks to output the vulnerable software and versions, similar to the NER system of [39].

- The SUM seeks to summarize the original CVE description while keeping all necessary software versions and bug text while normalizing it so that the output structure will be similar to other entries.

We also devised four variations to attend to various portions of the input and evaluate which method is most effective.

Single-task Model. In the single-task model, T5 is trained solely on the SUM task where the input is the source vulnerability description, and the output is the SUM feature of our curated dataset. Since this is a Single-task Model, we can fine-tune T5 and BART (see §4). However, in our evaluation (see §4) the model showed serious deficiencies, e.g., omitting part of the description or the software associated with it, necessitating the development of multi-task models, as we aim to force the model to attend to specific portions of the description.

SWV/BUG Concatenation Model. The first multi-task model was devised using a naive approach: the model is trained on both the SWV and BUG tasks such that it specifically attends to those sections separately. Given a CVE sample, the output produced by the model is obtained by concatenating the output of the BUG task (the vulnerability description) to the output of the SWV task (the affected software). One can think of this step as focusing on just the necessary information without regard to the overall summary.

One shortcoming of this approach is that the model may associate the two tasks as a result of a strong correlation in the output. This could occur when the output of the SWV task contains words or sentences that overlap with the BUG task, causing the final output to have repeated sections. We fine-tune this model on BART using two separate models for each task and concatenating the output of each model.

3-Task Concatenation Model. The 3-Task Concatenation Model uses multi-task learning, but in the opposite manner In particular, rather than treating the BUG and SWV tasks as sub-tasks of the SUM task, we rather inverse the order where the SUM task is used to support the training of the BUG and SWV tasks.

The 3-Task Concatenation Model is trained on all three tasks. However, instead of using the output of the SUM task, the final output is the concatenation of the SWV task and the BUG task. The idea with this heuristic is that if the model outputs the information that it thinks is part of the SWV and BUG tasks, then all necessary information should be present. In essence, one can observe that the 3-Task Model is an effort to prioritize fluency while the 3-Task Concatenation Model is an effort to prioritize completeness (see §4).

3-Task Model. Our solution to the potential repetition issue of the concatenation model was a true multi-task model. The 3-Task Model uses all of the available information for training—that is, it trains on all three tasks (SUM, BUG, and SWV) jointly. The final output of the run of this model is the output of the SUM task for the input. The idea is that training on the BUG and SWV sub-tasks would allow the model to better attend to those sections of the input without the loss of linguistic fluency associated with the concatenation.

Finally, it is essential to remember that our methodology significantly depends on the multi-task capabilities inherited in T5. Therefore, training BART on some of these models is infeasible because it is not intended as a multi-task model.

Dataset and Data Curation

We introduce a manually-annotated parallel dataset to fine-tune and evaluate *Mujaz*. CVE entry descriptions, the input of our dataset, are abstract multi-sentence summaries of a bug and the software it affects. Our dataset seeks to remove the abstract nature in the description, put summaries into a uniform (normalized) format, and extract necessary information about the vulnerability and the affected software.

This manual process uses tokenized CVE descriptions from [39]. A traditional tokenization system like *moses* [59] could be used here, but since punctuation is highly important to CVE entries (periods in version numbers, in file names, etc.), not over-tokenizing is vital to maintain the performance. The tokenization method from [39] was highly effective. Still, one flaw noticed was the improper segmentation of namespace specifiers (i.e., *::*) where *A::B* would be tokenized as *A : :B* but the proper tokenization would be to leave it as *A::B*. In this dataset, Dong *et al.*'s tokenization [39] is used as a starting point for the features. Still, apparent errors, such as improper namespace segmentation, are corrected manually to allow the model to summarize properly.

With these tokenized descriptions, we filter out any descriptions that do not come directly from CVE descriptions, such as Exploit-DB or SecurityFocus, as many of those descriptions contain dozens of lines of code and debugging, which we view as outside the scope of this work. Moreover, we only consider CVE descriptions of 13 words or longer as those shorter tended to be too succinct to summarize. Thus, no summarization was needed or lacked crucial information, such as software names and versions. Thus, no benefit would be derived from the summarization.

After applying our filtering steps, 15,209 entries are left—from these, the dataset is manually created. From each entry, a parallel corpus of 3 features is created: summary, software/version, and bug (corresponding to the SUM, SWV, and BUG tasks in the model). The final size of the dataset is 1,583 entries, each with SUM, SWV, and BUG features. Examples of each feature on an input description are in Table 4.1.

The overarching motivation for creating the features and choosing what information to include and omit was to make the summaries complete (containing all necessary information to see if the used software is vulnerable and how one may be exploited) but increase readability by omitting more

Table 4.1: Examples of simple task labels

Type	Content
Original	index.php in ownCloud 4.0.7 does not properly validate the oc_token cookie , which allows remote attackers to bypass authentication via a crafted oc_token cookie value.
BUG	remote bypass vulnerability exists via a crafted oc_token cookie value.
SWV	ownCloud 4.0.7
SUM	in ownCloud 4.0.7 a remote bypass vulnerability exists via a crafted oc_token cookie value.

technical information that was not necessary to just seeing how one may be exploited, such as memory address or snippets.

One of the key qualities of the dataset we sought to have is lacking vulnerability-type bias. The previous work by Dong *et al.* [39] introduced a manually annotated CVE dataset for NER. However, their dataset was heavily biased towards Memory Corruption, where 66.3% of the dataset is in that category. Our dataset gives near-equal representation to each of the 13 vulnerability types.

Evaluation

Mujaz was evaluated using two types of metrics: computational and human metrics. The computational metrics provide a quantitative measure of the performance and can be used to quickly evaluate a large sample size while providing benchmarks for future work to be compared with. Human evaluation allows for the output to be judged through features that are difficult to assess computationally, such as fluency and ease of understanding. In human evaluation, reviewers assign each target summary a score for each one of the evaluation metrics. Final scores are obtained by averaging the score over the number of samples. Figure 4.2 shows our evaluation metrics.

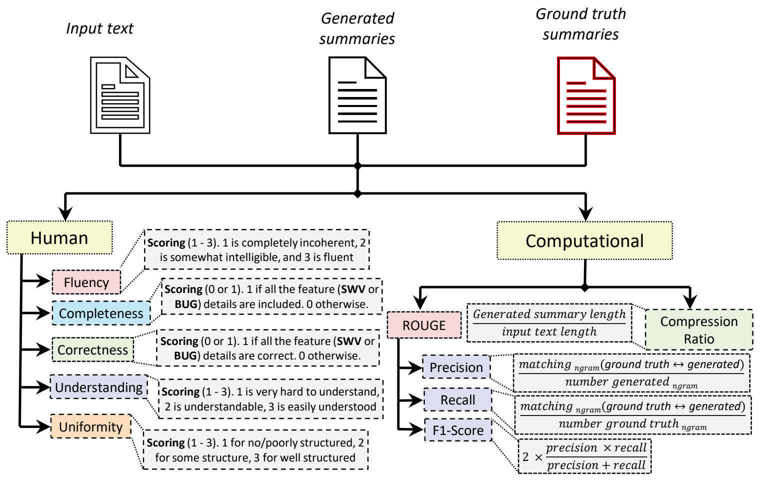
Input text
Generated summaries
Ground truth summaries
Human
Fluency
Scoring (1 - 3). 1 is completely incoherent, 2 is somewhat intelligible, and 3 is fluent
Completeness
Scoring (0 or 1). 1 if all the feature (SWV or BUG) details are included. 0 otherwise.
Correctness
Scoring (0 or 1). 1 if all the feature (SWV or BUG) details are correct. 0 otherwise.
Understanding
Scoring (1 - 3). 1 is very hard to understand, 2 is understandable, 3 is easily understood
Uniformity
Scoring (1 - 3). 1 for no/poorly structured, 2 for some structure, 3 for well structured
Computational
ROUGE
Compression Ratio
$\frac{\text{Generated summary length}}{\text{input text length}}$
Precision
$\frac{\text{matching}_{ngram}(\text{ground truth} \leftrightarrow \text{generated})}{\text{number generated}_{ngram}}$
Recall
$\frac{\text{matching}_{ngram}(\text{ground truth} \leftrightarrow \text{generated})}{\text{number ground truth}_{ngram}}$
F1-Score
$2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$

Figure 4.2: Evaluation metrics

Metrics

Computational Metrics. Two computational metrics are used to evaluate our models: ROUGE and compression ratio.

ROUGE[66]. ROUGE (or Recall-Oriented Understudy for Gisting Evaluation) is a metric that grades the generated summary against a reference via multiple sub-scores, and has been shown to correlate with human judgment. ROUGE calculates three sub-scores based on n-gram alignment between the reference and generated summary, the recall (R), precision (P), and F1 score. The recall score measures the number of matching n-grams between the generated summary and the reference divided by all the n-grams in the reference. The precision uses the same calculation as the recall but is divided by the model-generated count. F1 score is the average of precision and recall and is computed as $\text{F1} = 2(P \times R)/(P + R)$.

Compression Ratio. The compression ratio measures the reduction in sentence length and is calculated as the output length normalized by the input length. A low compression ratio is better, and

Mujaz should decrease the input length while maintaining favorable human metrics-based performance.

Human Metrics. Computational metrics, such as ROUGE, consider the number of overlapping n-grams which does not measure the coherence of the generated summary, requiring us to understand the quality of the generated summary. We came up with 5 metrics that measure the accuracy, structure, and coherence of the output. Scales for each metric are kept at a binary or small-scale level to reduce arbitrary grading.

Fluency. Graded on a 1-3 scale, where 1 is completely incoherent, 2 is somewhat intelligible, and 3 is fluent in the grammatical/semantical sense. Producing an incoherent output would render the output of the system less useful than it came in as the over-arching goal of the system is to create descriptions that are more easily understood.

Completeness. The completeness consists of two binary sub-scores for SWV and BUG. For *SWV-completeness* a 1 is awarded if all the software, and their respective versions, are included in the generated summary; otherwise, a 0 is awarded. Similarly, *BUG-completeness* is similar in that a 1 is awarded if all details of the original bug are included in the generated summary; otherwise, a 0 is awarded.

Correctness. The correctness consists of two binary sub-scores, similar to completeness. For *SWV-correctness* a 1 is awarded if all the software and their respective versions that are included in the generated summary are correct; otherwise, a 0 is awarded. *Bug-correctness* is similar in that a 1 is awarded if all bug details included in the generated summary are correct; otherwise, a 0 is awarded.

Understanding. This metric uses a 1-3 scale to judge the content and meaning of the text. If the meaning can be understood very easily, a 3 is awarded; 2 if it is more difficult but still possible, and 1 otherwise; i.e., it is very hard to understand the meaning of the generated summary. It is assumed

Table 4.2: A comparison between the different learning algorithms in terms of recall (R), precision (P), F1 score, and compression ratio (CR) when using a fine-tuned T5 model. Task-1=SUM, Task-2=SWV/BUG Concatenation, Task-3=3-Task Concatenation

Task	Model	R	P	F1	CR
Task-1	T5	0.7983	0.8571	0.8203	0.611
Task-1	BART	**0.8288**	0.8175	0.8171	0.6313
Task-2	T5	0.7829	0.9147	0.8362	**0.5523**
Task-2	BART	0.7886	0.8563	0.8115	0.5632
Task-3	T5	0.7992	**0.9153**	**0.8475**	0.5623

the input entries should grade high on understanding and fluency, as it is important to ensure the quality is not degraded as the input is processed using *Mujaz*.

Uniformity. *Mujaz's* goal is twofold (summarization and normalization), and the uniformity metric looks at the consistency in structure across the outputs of that specific model. Three (3) is awarded if the outputs are highly structurally consistent, 2 with consistency but obvious differences, and 1 otherwise.

Results

Computational Evaluation. We first report the score for all computational metrics in Table 4.2. The computational results are not meant to provide a comprehensive insight into the performance of each model, but a high-level idea of how compressed the text became and the recall quality whereas the human metrics will give an idea of the semantic quality.

As shown in Table 4.2, the 3-Task Concatenation model achieved the best score for recall, precision, and F1 compared to other T5 models. This is expected considering that the model trained on the three tasks, gives it the ability to generate summaries that overlap with the CVE description. However, the SWV/BUG concatenation model achieved the best compression ratio. This could be

Table 4.3: The impact of decoding methods on the model accuracy when using BART for training on the SUM task.

Decoding Method	Recall	Precision	F1	CR
Beam Search	**0.8288**	**0.8175**	**0.8171**	0.6313
Top K-Sampling	0.8236	0.8145	0.8129	**0.6302**
Top Nucleus	0.8275	0.8115	0.8135	0.6355

attributed to both tasks, as their content is shorter than the SUM task, considering they focus on very specific information with minimal words.

In contrast, BART results for the SUM task achieved better recall and F1 score compared to the SWV/BUG concatenation model, which excelled in precision and compression ratio. We reiterate that the SWV/BUG concatenation model was devised using two models to learn attending to each feature.

We tested each model with different decoding methods to evaluate token generation. Beam search was the best method for BART with a slightly higher F1-score than Top-*k* sampling and Top nucleus sampling. In contrast, T5 achieved very good results with the Beam search method but a very low score with the other methods. Therefore, we only consider Beam search as our decoding method for all other models. The evaluation of different decoding methods for BART is shown in Table 4.3.

Human Evaluation. The goal of human evaluation is to get a more comprehensive understanding of the generated summary in terms of its semantics, cohesion, and overall structure. A sample of 100 summaries from every model has been reviewed by 3 evaluators with a degree in computer science. This will provide an accurate score as they can judge the domain-specific content. The final evaluation of each metric is then averaged to get the score for each metric.

Almost all models did not do well for the BUG completeness metric, as shown in Table 4.4. The

Table 4.4: A comparison between the different learning algorithms when using BART and T5 as learning models in terms of subjective human evaluation metrics: fluency, completeness, correctness, Understanding, and uniformity. The Rouge score is included for the completeness of the results. The human evaluation is computed as the average score of three evaluators. Concat. stands for the task of concatenation.

Mode	Task	Fluency	Completeness		Correctness		Understanding	Uniformity	F1 score
			SWV	Bug	SWV	Bug			
T5	3-Task Concat.	2.2367	**0.9033**	**0.6633**	0.9367	**0.92**	2.63	2.48	**0.8475**
	3-Task	**2.82**	0.8933	0.5733	**0.9733**	0.8867	**2.8067**	**2.8367**	0.8235
	SUM Task	2.7767	0.8933	0.58	0.9667	0.8333	2.7267	2.8133	0.8203
	SWV/Bug Concat.	2.3033	0.8867	0.57	0.96	0.8733	2.7333	2.46	0.8362
BART	SUM Task	2.78	0.83	**0.7**	0.76	**0.92**	2.77	2.83	**0.8171**
	SWV/Bug Concat.	**2.81**	**0.9**	0.69	**0.83**	0.91	**2.84**	**2.8451**	0.8115

BUG attribute includes more descriptive information about the bug, which may span over several sentences, while SWV is usually short and self-contained, which is why its completeness metric is high.

One reason that contributed to these results is that we consider completeness as a binary metric. Therefore, when the generated summary misses some information from the BUG, the completeness is assigned 0 which affects the final score. However, the BUG correctness score is still high, meaning that even when the model does not capture the entire information about a bug, the partially captured information is correct. Fluency, understanding, and uniformity are subjective metrics, and they vary based on the evaluator's native language. However, we can see their evaluation are mostly consistent within a reasonable margin. We can justify the gap between their scores as their judgment might be lenient or conservative.

For completeness and correctness of SWV, all *Mujaz* scores remained above 0.83 except for BART on SUM task. Compared to the more abstract summarization of BUG, SWV is more extractive of distinctive words which makes it easier for the model to capture. Since it is extractive, it is unlikely an incorrect software name will be produced, resulting in extremely high correctness scores. As seen in Table 4.4, thorough vulnerability detection proved to be the most difficult task for both humans and models alike. Scores for output completeness were substantially lower compared to other metrics, showing that the models were unable to consistently grasp all necessary vulnerability details. However, the details that the models were able to capture were mostly correct.

Discussion

The BART Models

After evaluation, we conducted a thorough investigation of the 100 samples used for evaluation, and in the following, we discuss deeper insights into each model. Although we discuss each model individually, we note that some of the issues appear in more than one model.

SUM Task. Our first observation is that when our CVE input has multiple software versions, the model will produce some versions and miss the rest. Moreover, we observe that the model will wrongly predict some versions and add new other versions that were not present in the input CVE description.

In some extreme cases, the model will add new information that did not exist across the entire dataset, indicating such information was learned from the pretraining stage of the model. We notice that if the model predicts the incorrect software version for a CVE, the wrong prediction will propagate to other CVEs with similar content, e.g., adding punctuation where it is unnecessary. However, this could be explained by the often abnormal utilization of punctuation in CVEs that may confuse the model. In some cases, the software version numbers are written as a word instead of a number, although rarely encountered in our sample set.

SWV/BUG Concatenation Task. Two BART models were trained on each prefix and their predictions were concatenated. Therefore, no punctuation nor preposition to bridge this gap between the two outputs is utilized. To this end, we found that the model sometimes confuses some words that may occur interchangeably in the dataset or over using a technical concept, such as “denial of service”.

In one instance, “executing arbitrary commands” appeared in many CVEs and was summarized as “executing arbitrary code”. Other substitutions included replacing “firmware” with “software”, “and” with “or”, and “without” with “with”, which impacted the semantics of the generated summary. We found the model captures most versions, although still makes a mistake in a digit after a dot separator or generates an incorrect BUG description for a CVE with the same SWV. These deficiencies are explained by our training of two separate models.

The T5 Models

In the following, we highlight some of the observed issues in each model and contrast them with each other. We first discuss the 3-Task Concatenation Model and SWV/BUG Concatenation Model, as they both produce a summary by concatenating the prediction of SWV prefix and BUG prefix.

Concatenation Models. We observe that both models do not produce punctuation, although the 3-Task Model was trained on the SUM task, which has punctuation. We note the models learned some specific keywords, such as “by” and “via”, allowing the models to locate the bug accurately. These keywords are used interchangeably by the models, preceding the bug details. Similarly, other keywords that appeared frequently, such as “gaining information”, “csrf”, and “denial of service” are learned and used repeatedly in the generated summary.

We found the concatenation models to be inclusive, comprising CVE details except for one or two cases. Moreover, we found that the 3-Task Concatenation Model to confuse keywords that appear in context, e.g., producing “gaining privilege” instead of “gaining information” in one CVE. In contrast, SWV/BUG Concatenation Model predicts the correct words, implying that this confusion may come from the SUM prefix, and showing the impact of using multiple prefixes.

3-Task and SUM Task. Both models generated summaries with punctuation and context, producing coherent summaries. As in others, both models also learned keywords and used them in the prediction. Similarly, both models missed some information about the BUG if there were too many software versions. Besides, both models still had some shortcomings.

First, the SUM Task Model performed more abstractive summarization, making its structure different from the target summary. As such, we observed that the model sometimes adds new words or substitutes words with others, impacting the bug’s correctness. In contrast, the 3-Task Model predictions were very close to the target summary. We observe that both models replace the phrase “execute arbitrary code” with “php code execution” multiple times, although *php* was never mentioned in the source. This may show the effect of the pretraining stage or the effect of the SUM prefix during training and prediction since we use both models to predict the SUM prefix. Although the 3-Task Concatenation Model was trained with the SUM prefix, the phrase “php code execution” was rarely found in its output. Overall, the two models produced correct, concise, and easy-to-understand summaries.

We observe that all models, except SWV/BUG Concatenation, learn how to identify SWV even when it is structured hierarchically. A CVE entry may include multiple levels to describe the software, e.g., starting with the organization name, software suite, the software, or the sub-components of the software that are vulnerable. All models locate the SWV feature by following the chain until it reaches the last preposition “in”, which indicates the vulnerable software. This observation was confirmed in several similar samples.

The computational metrics can be misleading for pretrained models on a dataset like ours; while they achieved better F1 and compression ratio scores, this outcome is expected, since the SWV and BUG prefixes contain fewer words than the SUM prefix, which increases the compression ratio and F1 score. These scores could be observed when comparing the models using human metrics, supporting our claim.

The concatenation models got a lower score in terms of fluency, understanding, and uniformity, which shows that training on the SUM prefix produced better and richer summaries. Moreover, This shows that the compression ratio is not a good measure if we are interested in the summary quality.

In conclusion, we found that multi-task training on more prefixes needs to be evaluated carefully. Having more prefixes does not guarantee better results in terms of quality. Moreover, computational metrics should be backed by human evaluation, especially in sensitive cases, such as security applications.

Hyperparameter Tuning

We conducted extensive experimentation to learn the best set of hyperparameters for fine-tuning. Specifically, the tested hyperparameters were *repetition penalty*, *length penalty*, and *number of beams* for beam search. Moreover, we tested the two other decoding methods, top-*k* sampling and nucleus sampling. The experiments were conducted across batch sizes of 4, 8, and 16, and we report several patterns below. We chose the SUM task model because it is the simplest, using a single prefix for training and prediction.

For BART, we increased the number of beams and the length penalty, which improved the recall and F1 score but decreased the precision. One explanation is as we increase the number of beams, the model will have more options to choose from, thus possibly deviating from the reference. Moreover, we found that the repetition penalty has a direct effect on the recall, although it does not carry the same effect for T5.

Our results were consistent across different batch sizes, although the highest F1 score of 0.8101 was achieved with batch size 16. T5 exhibited similar patterns, although with different hyperparameters. With a batch size of 4, a number of beams of 5, and a repetition penalty of 1, T5 achieved an F1 score of 0.8277. Because some hyperparameters encourage a longer summary, the compression ratio decreased as we increased the batch size.

Decoding. Sampling methods consolidate top-k and nucleus sampling, which achieved very low scores with T5. Thus, we consider the sampling methods on BART. For the top-k sampling, k is set to 5, 10, 20 because our vocabulary size is small and p equals 0.95 for nucleus sampling. The repetition and length penalties were set to 2 based on our initial results with beam search. The recall score was consistent with k set to 5 for the top-k sampling, producing the best score across all batch sizes. However, this was not the case for precision, which fluctuated with different top-k values and batch sizes.

This unpredictability had an effect on the F1 score, which fluctuated with the best score, achieving 0.8176 with k set to 20 and batch size of 16; higher than the best F1 score for BART with beam search. On the other hand, the best compression ratio was 0.5905 with k set to 10, but it did not outperform the beam search. A batch size of 16 achieved the highest score for recall, precision, and F1 score.

Nucleus sampling considers a small subset of tokens from the top-k tokens where the cumulative probability exceeds a predefined threshold per Eq. (4.5). However, due to its stochastic behavior, the scores fluctuate for different metrics and top-k values. We found that the F1 score and precision increased with batch size for different top-k values, but the recall fluctuated. The best recall and F1 score were 0.8138 and 0.8168, with k equal 10, while the best precision score was 0.8317 with a batch size of 16. We also observed that $k = 5$ and $k = 10$ did not have any positive effect on the recall or F1 score, although affected the precision when the batch size was 4 or 16. The compression ratio score reached its lowest with $k = 5$ across the different batches but with a slim margin.

User study

Ensuring the effectiveness of *Mujaz* requires further testing to prove its usefulness. Thus, we performed a user study to examine *Mujaz* predictions and whether they support the human and computational metrics. Participants are presented with the original and normalized descriptions and asked to evaluate them across three metrics: understanding, coherence, and simplicity. These metrics estimate the impact of the generated summaries for comprehension.

Understanding follows the same criteria and scale established in 4. On the other hand, coherence refers to the clear connection of ideas within and between sentences to present easy-to-understand content. This metric is evaluated on a scale of 1-3. A score of 3 indicates that the content is coherent and the text flows smoothly. In contrast, a score of 2 implies that the text is partially coherent, with few sentences disrupting the flow. A score of 1 denotes the text is incoherent, lacking logical connections and clear structure. Our last metric is simplicity in an attempt to quantify the effect of the normalized description compared to the original description; in other words, does the normalized description make it simpler for a security analyst to understand it efficiently? This metric is also graded on a 1-3 scale, with a score of 3 confirming the normalized description is easier to understand. A score of 2 states that their differences are negligible, and both descriptions convey the same details. Finally, a score of 1 indicates the normalized description is more complicated than the original.

Due to the nature of our study, it is important to underscore the targeted audience, which could be security analysts or researchers. Therefore, our participants in this study must have solid background in security and privacy. The study was conducted with 12 participants, all of whom with at least a BS in computer science (9 are currently enrolled in the Ph.D. program in computer security) with a focus on security operation and threat intelligence). The users are presented with 100 samples of the generated normalized description and their corresponding original description. Each user assigns a value for each metric following the metric definition and scale. The scores across each metric are then averaged to represent the value for that metric and are shown in Figure 4.3.

For understanding, we see the distribution spread is between 2.44-2.95. Moreover, 75% of the scores are above the lower quartile of 2.595, with a median of 2.695, proving *Mujaz* produces easy-to-understand descriptions. In contrast, coherence exhibited greater variability ranging between 2.29-2.96. However, the distribution within the box is consistent with understanding, ranging from 2.60 to 2.86, with a median of 2.77, suggesting that a correlation exists between the two metrics. A reasonable justification for coherence variability is that participants possess different levels of proficiency in English, introducing a wider spread. Finally, simplicity showed more significant variability, with scores between 2.21-2.95. Moreover, the box distribution showed a wider range between 2.467-2.895, with a median of 2.67. However, considering that 75% fall above 2.467 is an indication that, for the most part, *Mujaz* generate simpler descriptions.

Summary and Work to be Completed

In this paper, we presented *Mujaz*, a new multi-task system that exploits pretraining language models to tackle vulnerability description summarization and normalization. We assess *Mujaz*

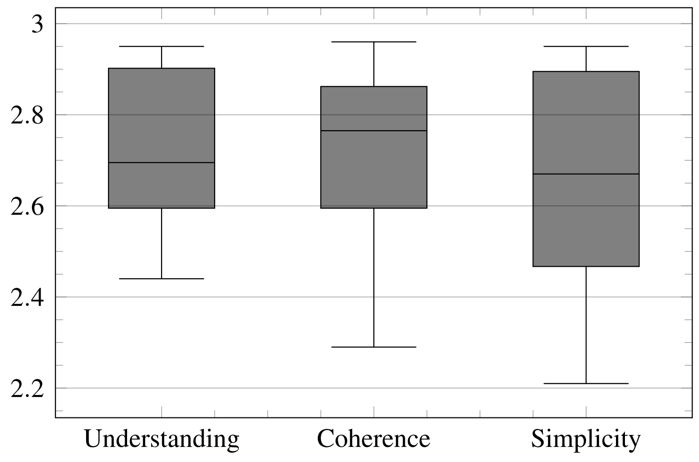

Figure 4.3: User study results summarize the evaluation of 100 samples generated by BART using the USE dataset with five evaluators.

using a parallel corpus emphasizing three different features. *Mujaz* devise several deployment options that attend to a specific detail of the vulnerability description. *Mujaz* was able to generate coherent summaries with a consistent and uniform structure and is shown effective in learning multiple features, measured by both computational and (our newly defined and justified) human metrics. Our results showed that attending to different aspects from the description is possible using *Mujaz's* architecture.

CHAPTER 5: *TASNEEF*: AUTOMATED CLASSIFICATION OF COMMON WEAKNESS ENUMERATION USING VULNERABILITY DESCRIPTION

The proliferation of software vulnerabilities is rapidly increasing. Vulnerabilities exist for many reasons, such as improper implementation, design flaws, or misconfiguration, rendering the software vulnerable to malicious adversaries. Although many systems are designed and developed diligently with extensive analysis and careful implementation, many still fail, necessitating constant updates to patch the exposure. Any individual could discover vulnerabilities. Moreover, the discovered vulnerability may not be reported to any recognized authority, and other malicious actors may exploit it repeatedly. Given the urgency to realize and patch a vulnerability, the textual details are essential in assessing the vulnerability and developing the appropriate countermeasure.

Maintaining the description's accuracy and clarity helps further analyze the vulnerability to quantify threat factors such as the attack complexity, scope, effect on confidentiality, integrity, availability, and many other characteristics. Therefore, it is critical to document and record such information in a database that could help discover/mitigate newly discovered threats. The analyzed vulnerabilities could be correlated to understanding their relationship through their features. Moreover, similar vulnerabilities are most likely to have similar characteristics and possibly similar mitigation techniques.

The Common Weakness Enumeration (CWE) provides a framework that describes weaknesses that can be exploited and triggers the vulnerability, providing additional context and details. The CWE database offers different categorization views that highlight specific aspects. For instance, software developers and architects can access the CWE from the perspective of software development, grouping weaknesses based on their relevance to different stages of the software development life cycle. On the other hand, the Research Concepts view arranges the CWE based on abstract behaviors, making it easier to analyze and identify dependencies. This view aims to encompass all weaknesses in the CWE list. We chose to utilize the Research Concepts view in our study because it aligns with our objective of linking vulnerability descriptions, which often provide context about their behaviors to their respective CWE.

Our Approach. The primary goal of our model is classification, which involves mapping a specific input to a target label. In our context, the input consists of vulnerability descriptions, and the output is the corresponding CWE class. *Tasneef* aims to evaluate various methods with different approaches to determine the best choice for predicting the CWE. Testing multiple models for classifying is essential because it allows us to explore different approaches and techniques to improve the accuracy and robustness of classification. No single model is universally superior for all types of data and scenarios, and the effectiveness of machine learning models can vary based on factors like the dataset's size, diversity, and the complexity of vulnerabilities. Moreover, we can gain insights into their strengths and weaknesses, leading to a more reliable CWE classification.

Evaluating *Tasneef* necessitates a substantial dataset covering multiple years, which brings several advantages to our dataset. First, it ensures diversity and coverage as vulnerabilities and attack vectors evolve over time. Second, it mitigates the risk of overfitting by exposing models to various types of data, enhancing their robustness for CWE classification. Furthermore, a dataset covering several years facilitates the identification of long-term trends or anomalies. Therefore, As a result, our dataset is sourced from the NVD, spanning across ten years from 2012-2022.

Contribution. Our contributions are as follows: ① We introduce *Tasneef* as an automated solution for classifying vulnerabilities into their corresponding Common Weakness Enumeration (CWE) categories. By leveraging various methods and techniques, Tasneef enhances the efficiency and scalability of vulnerability management. ② We conduct a thorough evaluation of diverse approaches, including Machine Learning (ML), Pre-trained Language Models (PLM), and Large Language Models (LLM), specifically for CWE classification. This evaluation yields valuable insights into the strengths and weaknesses of these models. ③ *Tasneef* provides practical insights for deploying vulnerability management strategies. Through the comparison of different methods, it aids organizations and individuals in selecting the most appropriate approach based on available resources.

Design Challenges

The primary goal of our model is classification, which involves mapping a specific input to a target label. In our context, the input consists of vulnerability descriptions, and the output is the corresponding CWE class. However, this task comes with several challenges. In the following sections, we outline the challenges associated with classifying CWE based on vulnerability descriptions.

Challenge 1: Domain-specific Language. Our classification problem targets a highly specialized domain language, namely security. Furthermore, the variability and ambiguity inherent in language pose an additional layer of complexity. Vulnerability descriptions exhibit wide variations in style and structure, making it challenging to establish a consistent and standardized input for the model. Additionally, some vulnerabilities involve non-technical weaknesses, such as social engineering, making their classification reliant on human judgment. To tackle this challenge effectively, we employ multiple approaches, each with its own set of choices, to identify the most suitable method for this task.

Challenge 2: Imbalanced dataset. The distribution of vulnerabilities across different CWE classes is often imbalanced, with some classes having significantly more instances than others. This disproportion can result in a model being trained with a bias towards the majority class, potentially causing difficulties in accurately classifying minority classes. Consequently, it is essential to thoroughly examine the dataset to grasp its distribution and determine how to address it impartially. After careful consideration of possible solutions for this challenge, we decided to keep the dataset as is for two key reasons. Firstly, it provides a realistic representation of the problem in reality, aligning the models with the imbalanced nature of real-world scenarios. Secondly, CWE encompasses a range of weaknesses, some of which are more prevalent than others. Therefore, it is necessary for the model to accurately identify them.

Challenge 3: Contextual Understanding. CWE classification demands a profound understanding of the context surrounding vulnerability descriptions. Certain vulnerabilities may share similar descriptions but belong to distinct CWE classes due to subtle distinctions in their behavior or consequences. Training a model to differentiate these nuances can prove to be a complex task. Furthermore, building a language model from scratch to comprehend vulnerabilities and classify them into their corresponding CWEs is not feasible due to the substantial resources and data required for such training. As a solution, we leverage various models, including Large Language Models (LLM) with billions of parameters, and apply them to our problem to determine the most efficient approach in terms of computational and financial resources.

Challenge 4: Hierarchical Labels. The CWEs are organized in a hierarchical structure, represented as a tree where each node corresponds to a type of CWE. The top-level nodes denote general classes or families of CWEs, while the child nodes represent more targeted and specific instances within those families. This hierarchy extends until it reaches leaf CWEs. However, it's important to note that the CWE tree may not have the same number of levels or children, resulting in varying depths and children among branches. Moreover, a vulnerability can be assigned multiple CWEs from different levels, essentially making it a multi-label classification with multiple paths from the top node to the leaf node. Given the inherent complexity of this hierarchical structure, we focus

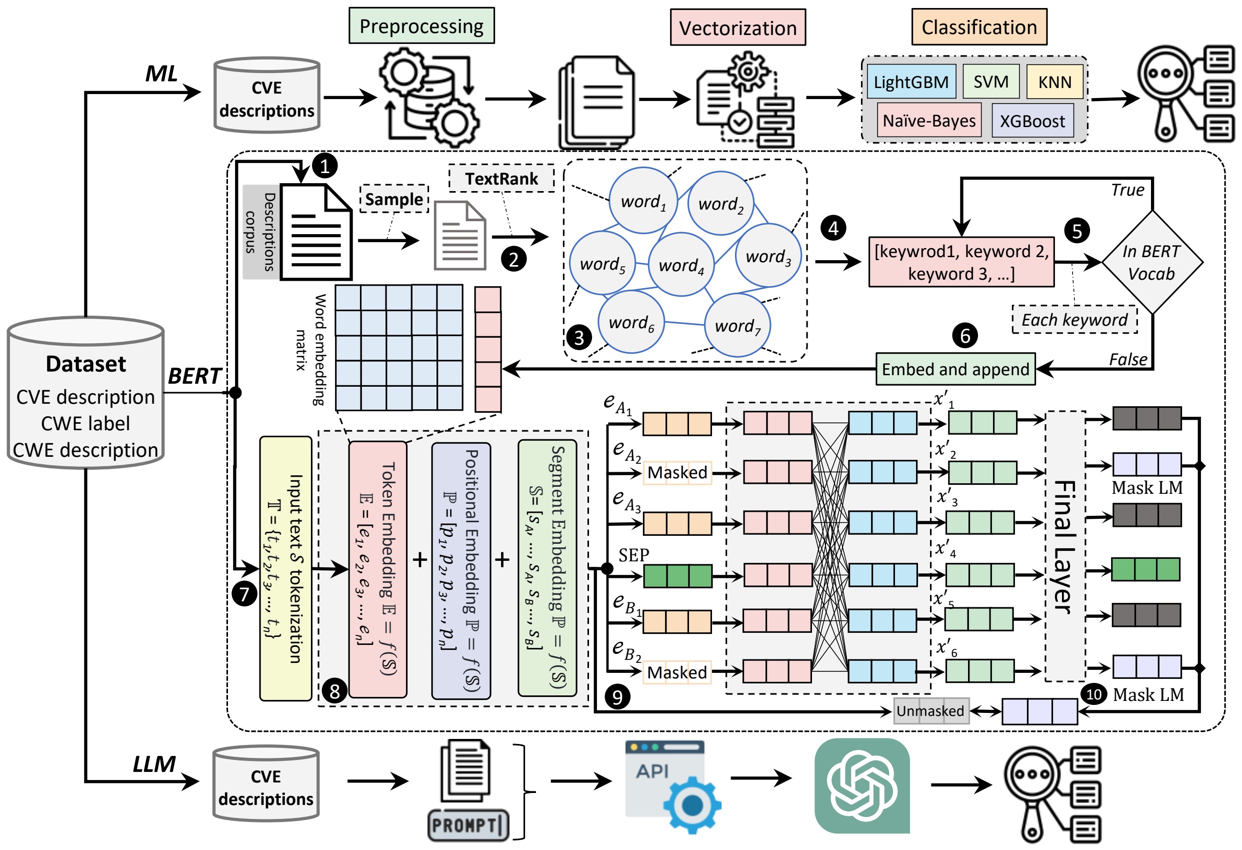

Figure 5.1: *Tasneef* CWE Labeling Pipeline

our attention on the top-level node, referred to as research concepts, to narrow down the scope of the problem.

Tasneef: Models Overview

In this section, we provide a comprehensive overview of the diverse models we have employed for CWE classification. Our selection includes machine learning models, pre-trained models, and large language models. These choices were made to facilitate a thorough comparison and performance evaluation. Figure 5.1 shows the overall design of *Tasneef*, constituting of several approaches.

Machine Learning. We start with Machine Learning (ML) methods. ML models are designed to facilitate the prediction of specific outcomes based on the input provided. For classification tasks, ML models acquire knowledge by utilizing a labeled dataset containing inputs represented by a set of features and corresponding labels. However, this dataset may require additional preprocessing steps to improve the models's performance. The top path of Figure 5.1 represents ML methods.

Pre-trained Language Models. PLMs operate through a two-step process: pre-training and fine-tuning. This approach was initially presented by Peters *et al.*in [85], using a bidirectional Long-Short-Term-Memory (biLSTM) for both stages. However, a groundbreaking shift occurred with the introduction of the Transformer architecture, presented in [102], bringing essential advancements to NLP. The Transformer introduced self-attention, enabling each word also known as a token in a sequence, to consider its relationships with all other tokens, significantly enhancing contextual understanding. Moreover, Transformers are highly efficient as they are capable of processing entire sequences in parallel during both training and inference. Lastly, Transformers are easily scalable, simply by adding more attention blocks or heads, which increases the models's size and parameters. It is worth noting that PLMs vary in size. For instance, BERT base model [38] comprises of 110 million parameters, while GPT-2 [88] has 1.5 billion parameters. The middle path of Figure 5.1 represents our approach for deploying PLM, BERT in this case. We delve into the details in section 5.

Large Language Models. LLMs are an extension of (PLMs) achieved by scaling up either the model size or the training data volume. This scaling approach has led to significant improvements in downstream tasks without altering the pre-training objective or the underlying architecture. LLMs have demonstrated exceptional abilities in reasoning and tackling complex tasks. A notable application of LLMs is ChatGPT [8], designed to generate human-like coherent text and engage in natural conversations with users. The primary distinctions between PLMs and LLMs can be summarized as follows:

1. LLM exhibits unique emergent abilities [106], which are defined as unique abilities not found in PLMs. For example, LLM can perform in-context learning, which refers to the language model's capacity to generate the desired output for test instances when provided with natural language instructions or task demonstrations without the need for additional training or gradient updates.

2. LLM displays the ability to follow instructions through what is known as instruction tuning, allowing LLM to follow the instructions to carry out the desired task effectively. Such instructions are given without explicit examples of demonstrations, improving the model's capabilities for generalization.

3. PLM falls short when solving complex tasks such as mathematical word problems. However, LLMs have demonstrated the power to solve such tasks through multi-step reasoning. Moreover, Chain-of-Thought (CoT) [107] prompting helps the model to produce intermediate results, leading toward the final outcome. A few studies suggested that this ability emerges from CoT and training on code [107, 42].

Given the popularity and success achieved by LLMs, we believe it is suitable to test such models for CWE classification to understand their capacity to classify domain-specific content. In the next section, we cover the technical details of each model used and the various options we explored to achieve the best results.

Models Technical Details

Machine Learning

Naïve Bayes. Probabilistic machine learning algorithms such as Naïve Bayes are commonly used for text classification tasks. The Bayes classifier leverages Bayes' theorem to predict the class of a new test instance by estimating the posterior probability as expressed in the following equation: $\hat{y} = \arg\max_{c_i} \{P(c_i \mid \mathbf{x})\}$ where $\hat{y}$, corresponds to the class c_i that maximizes the probability given instance $\mathbf{x}$. Bayes theorem allows us to invert the posterior probability in terms of the likelihood and the class prior probability as follows:

$$P(c_i \mid \mathbf{x}) = \frac{\overbrace{P(\mathbf{x} \mid c_i)}^{\text{likelihood}} \cdot \overbrace{P(c_i)}^{\text{prior}}}{P(\mathbf{x})} \tag{5.1}$$

The naive Bayes approach makes a simple assumption: all attributes are independent, simplifying the classifier with remarkable effectiveness. This independence assumption immediately implies that the likelihood can be broken down into a product of individual probabilities, as illustrated by the following equation:

$$P(\mathbf{x} \mid c_i) = P(x_1, x_2, \ldots, x_d \mid c_i) = \prod_{j=1}^{d} P\left(x_j \mid c_i\right) \tag{5.2}$$

Naïve Bayes uses a Bag-of-Words (BoW) model to represent text, which focuses on the frequency of words within a document and ignores the order and structure of words within text. First, we create a vocabulary of all the unique words that appear in the training data. Then, for each document in the training data, we create a feature vector that represents the frequency of each word in the vocabulary. We continue with this method for the rest of the ML methods.

K-Nearest Neighbor. Unlike Naïve Bayes, K-Nearest Neighbor (KNN) is a non-parametric algorithm that avoids making any presumptions about the underlying function. It works by finding the 'K' most similar data points to a new data point and then assigning the label based on the classes of the K nearest neighbors. The rationale behind KNN is that data points that are close to each other in a feature space are more likely to belong to the same class. Therefore, KNN can be expressed accordingly:

$$\hat{y} = \arg\max_{c_i} \{P(c_i \mid \mathbf{x})\} = \arg\max_{c_i} \{K_i\} \tag{5.3}$$

where K_i denotes the number of points among the K nearest neighbors of x that are labeled with class c_i. KNN with BoW achieved a very poor score in terms of our evaluation metrics, prompting us that such representation is unfit for such model. Therefore, we tested several embedding methods with KNN, since it relies on similarity measures in embedding space.

We tested FlagEmedding [111], which is the top-performing embedding on the Massive Text Embedding Benchmark (MTEB) [52] leaderboard, which evaluates embeddings on tasks such as semantic search and pair classification. FlagEmedding is a relatively large model, approximately 1.34GB. Given the success achieved by ChatGPT, we decided to test Open AI embedding model, named text-embedding-ada [10]. However, the model is only accessible through API by providing the required text for embedding. Finally, we tested the MPNet sentence transformer [2], which is a lightweight and fast model, approximately 0.44GB that provides overall good results. All of the embedding methods that we tested showed some improvement over the bag-of-words method, but the improvement between them was limited, achieving close scores. We report the results of the best embedding method, which is FlagEmedding.

Ensemble. Classifiers can fall into one of two extremes. On one end, high-variance classifiers demonstrate inherent instability as they are sensitive to any small perturbation, which results in the model's overfitting. On the other hand, high-bias methods tend to make strong assumptions about the underlying data, leading to underfitting. Both categories are considered unstable because they are unable to generalize to unseen data, whether the model is complex or simple, yielding significant changes in the prediction or decision boundary. Ensemble models mitigate this instability by creating a combined classifier using the output of multiple base classifiers trained on different data subsets. Mathematically, ensemble methods can be expressed as follows:

$$\hat{y} = \arg\max_{c_i} |\{M_t(\mathbf{x}) = c_i \mid t = 1, \ldots, K\}| \tag{5.4}$$

Given any test point x, it is first classified using each of the K base classifiers, M_t. The predicted class $\hat{y}$ of x is assigned by majority voting among c classes. We consider two ensemble methods: lightGBM [58], known for its speed and parallelizability, and XGBoost [31], renowned for its success and flexibility. Both models follow gradient boosting, which builds a predictive model in a sequential manner. Gradient boosting works by combining the predictions of multiple weak learners, particularly decision trees, to create a strong predictive model. Each model in the sequence aims to reduce errors from the previous ones.

Support Vector Machine. SVM [25] is a classification method that aims to find the optimal hyperplane that maximizes the margin between different classes. Finding a hyperplane separating distinctive classes may not be possible as real-world data is not perfectly separable. Soft margin SVM eases this objective by incorporating a slack term to allow for some degree of misclassification. SVM can be expressed mathematically as follows:

$$\hat{y} = \text{sign}(h(\mathbf{x})) = \text{sign}\left(\mathbf{w}^T\mathbf{x} + b\right) \tag{5.5}$$

where $(h(\mathbf{x}))$ is the optimal hyperplane function, consisting of a weight vector $\mathbf{w}$ and a bias $\mathbf{b}$. The classifier takes a point x to produce the target label.

Pre-trained Language Model

BERT. Transformers utilize attention mechanisms to capture dependencies and associations between tokens within a sequence, creating context-aware representations for each token. The original transformer [102] consists of two main components: an encoder and a decoder. The encoder constructs a comprehensive input representation, while the decoder leverages this representation to predict the next probable token from the vocabulary.

Many recent language models are designed as autoregressive generators, predicting the next token given previous tokens. PLMs following this architecture include GPT [86], GPT-2 [88], and other LLM like GPT-3 [28], Palm [33], BLOOM [91], and LaMDA [99]. These models rely solely on the decoder to encode and generate sequences. In contrast, BERT [38] was developed based on Masked Language Modeling (MLM). In MLM, a small percentage of randomly selected tokens are masked, and the model is trained to predict them. This distinction allows BERT to consider both left and right contexts during training, making it suitable for various tasks that do not involve sequence generation. BERT's Transformer follows a series of steps that involve tokenization, embedding, encoding, and finally, predicting the masked token.

Tokenization: In the tokenization step, the input sequence is segmented into a series of individual tokens. BERT leverages WordPiece sub-word tokenizer, introduced by Wu *et al.* [110], which effectively breaks down tokens into smaller, meaningful sub-word units. Subword tokenization entails the training of a tokenizer to construct a vocabulary based on a provided corpus and subsequently tokenizing new sequences in accordance with the constructed vocabulary. The tokenizer breaks words into elementary units acquired during training, enabling the tokenizer to manage complex expressions.

WordPiece starts with a predefined vocabulary containing subword units, which usually include individual characters, special symbols, and frequently encountered subword units. WordPiece works by iteratively adding the most frequent sub-tokens units to a vocabulary until the desired vocabulary size is reached. The tokenizer's training process commences with WordPiece learning to combine two sub-tokens from the foundational vocabulary by computing a score for each pair within the vocabulary, as shown below:

$$\text{score} = \frac{\text{freq of pair}}{\text{freq of first unit} \times \text{freq of second unit}} \tag{5.6}$$

The pair with the highest score gets merged. The algorithm gives preference to combining pairs in which the constituent elements are less common in the vocabulary. When processing a new token, the tokenizer initiates by searching for the longest subword unit from the vocabulary that matches a portion of the input word, and splits on it. Subsequently, it proceeds to tokenize the remaining segments of the word the same way. BERT deploys WordPiece embeddings with a $30,000$ tokens vocabulary.

Embedding: After the tokenization process, each token undergoes a transformation to create a meaningful representation that preserves its context. BERT employs a comprehensive embedding approach, including three types of embeddings: token embedding, positional embedding, and segment embedding. Token embedding assigns a unique embedding for individual tokens, while positional embedding maintains the token's position within a sequence, addressing the limitation of transformers in maintaining token order. On the other hand, segment embedding plays a vital role in distinguishing tokens within the same sequence from tokens belonging to other sequences. The final token embedding is created by combining these three types of embeddings for each token. This embedding strategy is critical in enhancing BERT's ability to encode rich contextual information effectively.

<u>*Encoder:*</u> As stated before, BERT uses only the encoder component of the transformer. The encoder consists of two sub-components: the self-attention layer and the Feed-Forward Network (FFN). The self-attention layer accepts an input sequence and produces a new one that captures how much attention each token should have with respect to other tokens. The most common function used is the scaled dot-product attention. However, the transformer enhances self-attention using query, key, and value vectors to compute attention, as shown below:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V \tag{5.7}$$

Self-attention is a way to encode the relationships between tokens in a sequence by calculating a weighted sum of their representations. In this process, each token is passed through three linear projections to generate its query Q, key K, and value V representations. The query vector is then compared to the key vector of all other tokens using the dot-product, producing a similarity score for each query-key pair. However, the average dot-product tends to rise when the embedding size increases, potentially causing gradient instability. Therefore, to stabilize the gradient and keep the weights within a specific range, the scores are divided by the square root of the embedding size d. The results are then passed through a softmax function to produce a probability distribution. The final step is multiplying each token with its corresponding value vector and summing them up to produce the final embedding for the queried token, encoding its dependencies with other tokens within the sequence.

Transformers employ multi-headed attention to improve the representation of tokens in a sequence, using multiple self-attention layers called "heads." Using multiple heads improves self-attention such that different heads can focus on different linguistic aspects of the sequence. For example, one head might focus on the semantics of the tokens, while another head might focus on the syntactic structure of the sequence. The output of each self-attention head is then concatenated and passed through a linear projection layer to produce the final sequence representation.

The Feed-Forward Network (FFN) in the encoder is a fully connected layer without an activation function, allowing the model to learn complex relationships between tokens. Each token undergoes independent processing by the FFN to refine its embedding. However, it is interesting to notice that FFN accounts for more than two-thirds of the trainable parameters in the transformer, highlighting its significant influence on its operations. In [38], BERT was introduced with two sizes: BERT-Base with 12 encoder layers and BERT-Large with 24 layers. The encoder blocks are connected in a cascading fashion such that the output of the previous encoder block is passed as an input to the next block.

Training: The training of PLMs, such as BERT, involves two main steps: pre-training and fine-tuning. During pre-training, the model is exposed to enormous text to learn various patterns and gain a comprehensive understanding of the underlying text. Following pre-training, the model's parameters are fine-tuned on a labeled dataset specific to a particular downstream task, adjusting the model's parameters to align with the task's objectives.

BERT Deployments. We explore two deployment options for BERT in our CWE classification task:

Option 1: Fine-Tuning: This approach is simple and straightforward. We fine-tune BERT using our custom dataset, which contains vulnerability descriptions as the input text and their corresponding CWE labels as the target.

Option 2: Extended Pre-Training and Fine-Tuning: This approach is represented in Figure 5.1, which constitutes of multiple stages. BERT's vocabulary has a finite number of tokens, each associated with its embedding for representation. When processing text, BERT starts by tokenizing the text, breaking it into independent tokens. If a token is part of BERT's vocabulary, it uses its representation. Otherwise, BERT will represent it by breaking it down into smaller units, such as individual letters or sequences of letters presented in its vocabulary. To enhance the model's overall capability, we extract important tokens often found in CVE and CWE descriptions but not included in BERT's vocabulary. For those new tokens, we embed them using BERT's embedding and add them to BERT's vocabulary. Then, we continue the pre-training of BERT using MLM objective on a corpus created from both vulnerability and CWE descriptions. Finally, we fine-tune the model on our CWE dataset.

Following Figure 5.1, in step ❶, we create a corpus by gathering vulnerability and CWE descriptions. Due to limited resources, we randomly sample approximately 20% of this corpus for keyword extraction in step ❷. We employ the TextRank algorithm [71] to construct a word graph, as depicted in ❸, and subsequently extract essential keywords, illustrated in ❹. We then check whether these keywords are present in BERT's vocabulary in step ❺. If a keyword is found, we move on to the next one. Otherwise, we embed the token and appended it to the word embedding matrix of BERT as shown ❻. Once this process is complete, we proceed with BERT's pre-training in step ❼, utilizing the corpus established in step ❶.

BERT starts by breaking the text into a set of tokens. The tokens are then embedded according to their representation in the embedding matrix. As explained, positional embedding is incorporated for each token To retain its order within the sequence. Additionally, BERT augments this embedding by providing segment embedding, which is consistent for tokens within the same sentence but distinct from other sentences, as shown in ❽. BERT's methodology involves randomly masking a small percentage of tokens with the objective of predicting them. Moreover, a special token, denoted as [SEP], is inserted between sentences, as visualized in ❾. The sequence passes through multiple encoder blocks, each with multiple attention heads for encoding, with the final layer responsible for predicting the masked tokens. Model parameter updates occur through the computation of cross-entropy loss between the predicted and masked tokens, as illustrated in ❿.

Other models. To provide a more comprehensive coverage of PLMs, we evaluate several models that share similarities with BERT but incorporate some distinct modifications. These models have demonstrated superior performance on certain benchmarks while retaining the core encoder-only transformer architecture inherited in BERT.

RoBERTa [68] developed by Facebook is a PLM designed to improve the pre-training of BERT. It introduces various modifications, including extended training on longer sequences and larger batch sizes with a more extensive text corpus. Moreover, RoBERTa eliminates the Next Sequence Prediction (NSP) objective and applies a dynamic masking pattern during training.

DistilBERT [90]: a compact and efficient variant of BERT developed by HuggingFace that preserves approximately 97% of BERT's language understanding capabilities. It is trained using knowledge distillation [49], a process in which the knowledge of BERT is transferred to a smaller model, enabling it to reproduce the output of the large model. DistillBERT is very practical, and its performance is sufficient for many tasks, particularly in resource-constrained scenarios.

Electra [35]: is another PLM that introduces a novel pre-training objective called replaced token detection. Instead of masking a random token, Electra replaces a token with some plausible alternatives sampled from a small generator network. Subsequently, a discriminator model is trained with the objective of distinguishing between the actual token and the plausible generated token.

Keyword Extraction. Keyword extraction is a vital NLP task that involves identifying and extracting significant words or phrases from a text document. Various techniques are available for keyword extraction. For example, frequency-based methods like term-frequency and Term Frequency-Inverse Document Frequency (TF-IDF) assess word importance based on their occurrence frequency. However, these methods lack a deep understanding of semantic context, potentially missing keywords with lower frequencies. On the other hand, machine learning techniques can also be employed for keyword extraction, utilizing both supervised and unsupervised approaches. However, ML models present several disadvantages in this context. Firstly, supervised learning for keyword extraction requires a high-quality annotated dataset, which is time-consuming and labor-intensive to create. Moreover, the textual data used for keyword extraction in our case is brief and authored by different individuals, resulting in unique structures and styles, limiting the model's generalization capacity and increasing susceptibility to overfitting. Finally, our corpus is unstructured, as it comprises various descriptions appended to a single file, and ML methods struggle, considering their contextual understanding is limited.

In contrast to previous approaches, graph-based methods leverage a graph structure to model keywords as nodes and the relationships between them as edges. An example of such method is TextRank [71], which adapts Google's PageRank [27] algorithm to assess word importance based on their co-occurrence patterns with other words. In TextRank, words with high centrality within the graph are considered keywords. Node centrality refers to the significance of a node within a graph, aiding in the identification of crucial nodes.

TextRank offers several advantages as a keyword extraction method. First, it is language-independent, making it applicable to various text data without the need for additional processing. Moreover, It operates as an unsupervised method, eliminating the need for labeled training data. In addition, TextRank is resilient to stop words as they have numerous connections (edges) to other words in the graph, lowering their centrality. Finally, it is able to capture contextual information as it identifies keywords based on their co-occurrence patterns within a corpus. Given these advantages, we chose TextRank as our method for keyword extraction.

We delve into the technical details of how TextRank is implemented for keyword extraction. First, we created a corpus using all CVE and CWE descriptions in our dataset. The next step is pre-processing to ensure the quality of the extracted keywords, including stopwords and punctuation removal, and stemming for word normalization. For our approach, we also utilize Part-Of-Speech (POS) tagging and consider only nouns, proper nouns, and verbs, as they tend to carry more weight and significance in the text. Finally, we apply the TextRank algorithm to the processed corpus. TextRank analyzes the relationships between words within the corpus to identify and extract keywords effectively.

TextRank constructs a graph where every word is considered as a node. Edges are created between any two vertices (words) whenever they co-occur within a defined window of size, set to 4. The algorithm proceeds to calculate the score for each vertex using the PageRank formula, designating the importance of that vertex.

$$S(V_i) = (1-d) + d * \sum_{j \in \mathrm{In}(V_i)} \frac{1}{\left|\mathrm{Out}\left(V_j\right)\right|} S\left(V_j\right) \tag{5.8}$$

$S(V_i)$ represents thes score of a node V_i, indicating its importance. d is a damping factor, typically set to a value between 0 and 1. In the context of TextRank, it models the probability of jumping from a given vertex to another random vertex in the graph. The damping factor ensures that nodes with low connectivity still receive some importance. $In(V_i)$ represents the neighboring nodes that have links pointing to the node V_i, while $Out(Vj)$ represents the outgoing edges from node V_j. The equation calculates the contribution of incoming links from other nodes to the target node V_i by summing over the contributions from all neighboring nodes j that have links pointing to node V_i. Each contribution from node j is weighted by the importance score V_j. After extracting keywords, we embed the top 500 keywords that do not exist in BERT's vocabulary.

Keywords Embedding. We use Flair framework [12] to embed the extracted tokens using BERT embedding. Flair is an open-source NLP library developed by Zalando Research, designed to simplify and accelerate the development of NLP models for various tasks. However, since BERT does not have such tokens in its vocabulary, its tokenizer will break each token into multiple sub-tokens. We create the embedding of the unknown token by averaging over the embedding of its sub-tokens. Finally, we add the new token and its embedding to BERT's vocabulary.

Pre-training and Fine-tuning. We continue the pre-training using the MLM objective, masking 15% of the corpus and training the model to predict them. We do not consider the Next Sentence Prediction (NSP) objective, as several studies [68, 89] indicated that it has a limited effect on the model performance. Moreover, our corpus consists of independent descriptions that have no relation with the next or previous sequences. At this phase, BERT takes a sequence with some masked tokens, and the final layer produces the most probable tokens. After pre-training, we fine-tuned the model using our dataset with the vulnerability description as the input and CWE as the target label. At this stage, the final layer of BERT needs to be modified to reflect the new objective, which is sequence classification instead of MLM. However, all other layers and their weights are retained from the pre-trained stage.

Large Language Model

ChatGPT. is created by building upon the robust GPT model series, with a particular emphasis on optimizing its capabilities for conversations. The remarkable success of the GPT models was based on compressing the world knowledge into a decoder-only transformer modeled as an autoregressive generator. Two key factors contribute to their success: training the transformer to predict the next token of the sequence accurately and scaling the size of the language model. The most significant changes appeared in GPT-3. The model size increased enormously from 1.5B in GPT-2 to 175B with approximately 11,567% increase in the number of parameters. Moreover, It introduces the concept of In-Context Learning (ICL) through few-shot or zero-shot learning. However, GPT-3 still has some limitations in its ability to solve complex tasks that require a step-by-step analysis like code completion. As a result, Codex [30] was introduced, which is a GPT-3 model fine-tuned on a massive dataset of code, including code from public GitHub repositories, allowing it to understand the relationship between human language and code.

ChatGPT [8] was built upon GPT-3.5-turbo [9], which is the most capable model for text and code generation. GPT-3.5-turbo model was trained on a blend of text and code with human alignment to learn human preferences through Reinforcement Learning from Human Feedback (RLHF) [34]. It works by training the LLM to generate text that maximizes a reward function learned from human feedback. GPT-3.5 turbo uses Proximal Policy Optimization (PPO) [92] algorithm for RLHF because it is simple, stable, and efficient.

LLM training requires massive resources in terms of budget, computing power, and technical expertise, making reproducing a model almost infeasible for individuals and institutions. Moreover, LLMs size is enormous, creating another hurdle for using them. Instead of deploying the model locally, using the Application Programming Interface (API) to access the model directly is faster and more convenient. Communicating with LLMs is done through messages referred to as prompts, expressing what the user wants the model to perform. A prompt can include multiple specifications in an attempt to guide the model to generate the desired output.

Evaluation

Experimental Setup

In our experimentation, we employed various Machine Learning (ML) algorithms implemented using libraries such as sklearn [83], LightGBM [36], and Xgboost [32]. Moreover, we used the Natural Language Toolkit (NLTK) [24] for preprocessing and cleaning. For KNN, we chose $K = 5$, and FlagEmbedding for embedding since it produced the best result. For the ensemble method, we chose the number of estimators to be 200 with a learning rate of 0.3.

For PLMs, we leveraged the HuggingFace library [109] for downloading the models used. There are many variations of BERT that differ in size and specialization. Given hardware constraints, we used BERT base model because it fits our GPU. Moreover, we chose a model that is sensitive to characters-case, as our text includes names for different software and vendors as well as other specific acronyms that are all capitalized. Similarly, DistilBERT offers a similar case-sensitive variant, in contrast to Electra and RoBERTa, which lack such options. We used AdamW [70] as an optimizer with a learning rate set to 0.00002 and batch size of 16 To evaluate our models, we divided the dataset into training and testing subsets, with 20% reserved for testing.

For LLMs, we employed ChatGPT with the GPT-3.5 turbo model. While GPT-4 offers enhanced reasoning capabilities, it incurs higher costs for usage. Since our task primarily involves text classification, we deemed GPT-3.5 turbo sufficient for our needs. We simply pass a prompt instructing ChatGPT to assign a CWE label to the given vulnerability description, and the return label will be recorded. However, we had to refine our prompt to make it more assertive in assigning a label, as we encountered instances where it did not return a CWE class.

Dataset

We chose our source to be the NVD [5] because it is a reliable source managed by NIST and supported by the US government. Our data collection spanned a ten-year timeframe, from 2012 to 2022, providing a comprehensive view of the NVD, including label distribution and reporting entities. For each vulnerability, we collected the following attributes: CVE ID, CVE description, CVSS score, source, published date, and the CWE class.

The CWE is listed according to a specific view, relating weaknesses to that view. For example, The software development view categorizes weaknesses based on concepts encountered frequently in software development, including architecture and implementation. However, we chose a more generic view, suitable for research. The research concept view tries to encompass every vulnerability listed in CWE to facilitate weakness analysis and exploration. Moreover, its primary organization is centered on abstract behaviors rather than focusing on the technical details, such as how a weakness can be triggered or detected. The CWEs are organized in a hierarchical structure represented as a tree in which each node is a type of CWE. The top-level, referred to as pillar CWEs, represent a general class or a family type of CWE.

In our initial analysis, we uncovered several key insights. First, we identified instances with repeated descriptions but different CVE IDs, totaling 5674 cases. This is significant because a new CVE ID should denote a distinct vulnerability or highlight its differences from other vulnerabilities. For instance, CVE-2022-42780, CVE-2022-42779, 2022-42774 have exactly the same description by *Unisoc*. In fact, we found more than 15 duplicates for this vulnerability. To maintain dataset integrity, we removed these duplicates, retaining a single instance for each. Moreover, we found 50 instances that had their analysis deferred by the NVD without any CWE class. Finally, we encountered over 17,461 instances that had either deprecated categories or followed a different view without a pillar CWE, which were also removed from the dataset.

NVD utilizes two CWE classes, namely NVD-CWE-Other and NVD-CWE-noinfo, to classify some vulnerabilities. For example, NVD uses only a subset of the available CWE classes, and any weakness type that is not covered by that subset is assigned NVD-CWE-Other. In contrast, NVD-CWE-info is used when there is insufficient information to classify a vulnerability into any specific CWE category. Therefore, we remove such entries from our dataset. The total number of instances in the dataset is 104155. The label and vendor breakdown is shown in Table 5.1. We did perform some statistical additional analyses such as Named Entity Recognition (NER), Tri-gram,

Table 5.1: Our dataset distribution: various vulnerability types and the number of samples in each.

Label	# of samples	Entity	# of samples
CWE-664	47157	MITRE	47714
CWE-707	31964	Microsoft	5623
CWE-284	13803	Oracle	4838
CWE-693	3299	Red Hat	4757
CWE-703	2808	IBM	4254
CWE-691	2414	Android	3443
CWE-682	2290	Cisco Systems	3172
CWE-435	221	Apple	2892
CWE-710	102	GitHub	2866
CWE-697	97	Adobe Systems	2618

and length analysis. However, they did not provide us with any meaningful insights simply because the dataset is imbalanced with respect to the label distribution.

Evaluation Metrics

We evaluate our model's predictions using several metrics, including accuracy, precision, recall, and F1-score. However, before defining these metrics, we need to define some concepts used to describe the predicted instances with respect to the ground truth. True Positives (TP) is the number of instances correctly classified as positive. In contrast, True Negatives (TN) refer to instances correctly labeled as negative. False Positive (FP) refers to instances incorrectly classified as positive, while False Negative (FN) is incorrectly classified as negative.

The accuracy measures the ratio of correctly classified samples to the total number of instances in the dataset as follows:

$$\text{Accuracy } = \frac{TP+TN}{TP+TN+FP+FN} \tag{5.9}$$

However, accuracy is not suitable for an imbalanced dataset as it can be misleading when the class distribution is skewed. In contrast, precision and recall are more useful for imbalanced datasets. Precision measures the accuracy of the model's positive predictions, considering only positive classes, as shown below. On the other hand, Recall measures the proportion of actual positive cases that were classified correctly.

$$\text{Precision } = \frac{TP}{TP+FP} \quad , \text{Recall } = \frac{TP}{TP+FN} \tag{5.10}$$

The F1-score, also known as the harmonic means, combines both precision and recall into a single value, striking a balance between the two metrics.

$$\text{F1 } = \frac{2 \times P \times R}{P+R} \tag{5.11}$$

Results

The results from all models are summarized in Table 5.2. Remarkably, traditional ML methods, such as XGBoost and SVM, achieved competitive scores of 0.865 and 0.860 on the F1 score metric, approaching the performance of more advanced techniques. In contrast, PLMs outperformed both ML and LLMs. Among PLMs, Electra demonstrated the highest performance, closely followed by other PLMs. Surprisingly, some smaller models like DistillBERT delivered outstanding results considering their size, computational constraints, and fine-tuning time.

Conversely, ChatGPT demonstrated lower effectiveness in CWE labeling, yielding the lowest scores among all models with an F1 score of 0.775. Moreover, it also performed a lower score with an F1 score of 0.546 in predicting specific CWE labels—a challenging task given the 392 labels available for classification. We focus on the F1 score because it presents a balanced score

Table 5.2: A comparison between the different methods for CWE classification in terms of accuracy, recall, precision, F1 score

Model	Accuracy	Recall	Precision	F1
Naïve Bayes	0.777	0.788	0.777	0.757
KNN [FlagEmbedding]	0.772	0.767	0.772	0.764
lightGBM	0.846	0.848	0.846	0.846
XGBoost	0.867	0.867	0.867	0.865
SVM	0.861	0.850	0.861	0.860
BERT	0.870	0.868	0.870	0.869
BERT (Pre-training)	0.849	0.847	0.849	0.847
DistilBERT	0.871	0.869	0.871	0.870
RoBERTa	0.872	0.870	0.872	0.871
Electra	0.874	0.871	0.874	0.873
ChatGPT	0.775	0.805	0.775	0.785

compared to other metrics. Table 5.3 shows the results for testing PLM and LLM on the leaf CWE label, which is a very specific label for a vulnerability. We see that DistillBERT outperformed both Electra and ChatGPT.

Discussion

We were surprised by the performance of traditional machine learning methods, achieving high scores and demonstrating their effectiveness. This is particularly noteworthy given the diverse nature of our textual dataset, comprising various vulnerabilities with inconsistent descriptions. Moreover, text representation was based on a simple method, namely, BoW. One notable advantage of these machine learning methods is their resource efficiency, as they can be applied locally on standard CPUs. However, there are differences in training times among methods, with SVM taking the longest, exceeding 3 hours for training and testing. On the other hand, Pre-trained Language Models (PLMs) achieved the best scores, but they come with increased resource requirements in

Table 5.3: A comparison between PLM and LLM on leaf node (specific) CWE labels in terms of accuracy, recall, precision, F1 score

Model	Accuracy	Recall	Precision	F1
DistilBERT	0.748	0.725	0.748	0.731
Electra	0.736	0.701	0.736	0.713
ChatGPT	0.568	0.678	0.568	0.546

terms of storage and time. Additionally, PLMs necessitate GPU resources, as fine-tuning on a CPU is not feasible.

We found that the extended pre-training of BERT followed by fine-tuning did not perform better than the direct fine-tuning of BERT. Our rationale suggests that the structure of the corpus used for pre-training played a crucial role in this outcome. The corpus was created by appending vulnerabilities and CWE descriptions all to a single file. The lack of coherent and consistent context within the corpus may have contributed to the model performance degradation. Furthermore, the difference in objectives between pre-training and fine-tuning could be another contributing factor. As mentioned in section 5, our pre-training objective was masked language modeling, which involves predicting masked tokens. However, employing this model for sequence classification required modifying the last layer to align it with classification.

Among all PLMs, Electra achieved the highest score, while the others closely followed with similar results. Moreover, all models except for DistilBERT required significant computational resources and took more than 6 hours for fine-tuning. On the other hand, DistilBERT reduces these requirements by almost half, making it an efficient choice for a wide range of tasks. This trend of creating smaller yet high-performance models also extends to newer LLMs. For instance, Meta AI recently introduced Llama 2 [101], a successor to Llama [100], with models ranging from $7B$ to $70B$ parameters. The smallest variant can be deployed on a local machine with a single GPU. On the other hand, The largest model has $70B$ parameters, and it outperformed ChatGPT on multiple benchmarks. This research direction emphasizes the development of compact models while maintaining high performance on various tasks like DistilBERT in our experiment.

In our evaluation, ChatGPT exhibited inferior performance across all metrics, underscoring certain limitations in LLMs that render them unsuitable for some tasks. Moreover, ChatGPT struggled to predict more specific labels, failing to achieve satisfactory scores. In contrast, PLMs demonstrated their reliability in such tasks, even when dealing with a substantial number of labels. Additionally, small models like DistillBERT exhibited exceptional proficiency in predicting specific labels, surpassing Electra, making DistillBERT an excellent choice for classifying CWE labels. However, It is worth noting a fundamental distinction between PLMs and LLMs. PLMs are fine-tuned using a custom dataset and employing backpropagation to update the gradient and improve the model. On the other hand, LLM lacks this option without incurring a significant cost for such a service. In our experiment, we accessed ChatGPT by sending requests for labeling, which does not provide the model with the opportunity for fine-tuning.

Summary of the Completed Work

In this paper, we presented *Tasneef*, a model encompassing several models to predict the CWE class of a vulnerability using its description. *Tasneef* exploits various approaches, including conventional ML models, PLMs, and LLMs to comprehend the context of vulnerability descriptions and assign the closest CWE label. *Tasneef* was evaluated using various metrics to measure its effectiveness, including accuracy, precision, recall, and F1 score. Our results showed that PLMs are still capable of competing and surpassing state-of-the-art LLMs for some classification tasks. ML models were able to achieve outstanding scores comparable to PLMs without the need for computing resources.

CHAPTER 6: CONCLUSION

APPENDIX A: INSTITUTIONAL REVIEW BOARD (IRB)

UNIVERSITY OF CENTRAL FLORIDA

Institutional Review Board
FWA00000351
IRB00001138, IRB00012110
Office of Research
12201 Research Parkway
Orlando, FL 32826-3246

EXEMPTION DETERMINATION

April 13, 2023

Dear Hattan Althebeiti:

On 4/13/2023, the IRB determined the following submission to be human subjects research that is exempt from regulation:

Type of Review:	Initial Study, Exempt 2i
Title:	Understanding the Impact of Using a Normalized Vulnerability Description on Security Analysts
Investigator:	Hattan Althebeiti
IRB ID:	STUDY00005405
Funding:	None
Grant ID:	None
Documents Reviewed:	• Study 5405 HRP-254 - FORM - Explanation of Research (14) IRB edits..pdf, Category: Consent Form; • Study 5405 HRP-255 - FORM - Request for Exemption (26)IRB edits.docx, Category: IRB Protocol; • Vulnerbility_description_userstudy.docx, Category: Other;

This determination applies only to the activities described in the IRB submission and does not apply should any changes be made. If changes are made, and there are questions about whether these changes affect the exempt status of the human research, please submit a modification request to the IRB. Guidance on submitting Modifications and Administrative Check-in are detailed in the Investigator Manual (HRP-103), which can be found by navigating to the IRB Library within the IRB system. When you have completed your research, please submit a Study Closure request so that IRB records will be accurate.

If you have any questions, please contact the UCF IRB at 407-823-2901 or irb@ucf.edu. Please include your project title and IRB number in all correspondence with this office.

Sincerely,

[illegible]

Harry Wingfield
Designated Reviewer

APPENDIX B: PUBLICATIONS COPYRIGHT

Licence to Publish Proceedings Papers

SPRINGER NATURE

Licensee	Springer Nature Switzerland AG	(the 'Licensee')
Title of the Proceedings Volume/Edited Book or Conference Name:	WISA 2022 (Inf. Sec. Appl.)	(the 'Volume')
Volume Editor(s) Name(s):	Taek-Young YOUN and Ilsun YOU	
Proposed Title of the Contribution:	Click here to enter text.	(the 'Contribution')
Series: The Contribution may be published in the following series	A Springer Nature Computer Science book series (CCIS, LNAI, LNBI, LNBIP or LNCS)	
Author(s) Full Name(s):	Hattan Althebeiti, David Mohaisen	(the 'Author')

When Author is more than one person the expression "Author" as used in this Agreement will apply collectively unless otherwise indicated.

Corresponding Author Name:	David Mohaisen	
Instructions for Authors	https://resource-cms.springernature.com/springer-cms/rest/v1/content/19242230/data/	(the 'Instructions for Authors')

1 Grant of Rights

a) For good and valuable consideration, the Author hereby grants to the Licensee the perpetual, exclusive, world-wide, assignable, sublicensable and unlimited right to: publish, reproduce, copy, distribute, communicate, display publicly, sell, rent and/or otherwise make available the contribution identified above, including any supplementary information and graphic elements therein (e.g. illustrations, charts, moving images) (the 'Contribution') in any language, in any versions or editions in any and all forms and/or media of expression (including without limitation in connection with any and all end-user devices), whether now known or developed in the future. Without limitation, the above grant includes: (i) the right to edit, alter, adapt, adjust and prepare derivative works; (ii) all advertising and marketing rights including without limitation in relation to social media; (iii) rights for any training, educational and/or instructional purposes; (iv) the right to add and/or remove links or combinations with other media/works; and (v) the right to create, use and/or license and/or sublicense content data or metadata of any kind in relation to the Contribution (including abstracts and summaries) without restriction. The above rights are granted in relation to the Contribution as a whole or any part and with or in relation to any other works.

b) Without limiting the rights granted above, Licensee is granted the rights to use the Contribution for the purposes of analysis, testing, and development of publishing- and research-related workflows, systems, products, projects, and services; to confidentially share the Contribution with select third parties to do the same; and to retain and store the Contribution and any associated correspondence/files/forms to maintain the historical record, and to facilitate research integrity investigations. The grant of rights set forth in this clause (b) is irrevocable.

c) If the Licensee elects not to publish the Contribution for any reason, all publishing rights under this Agreement as set forth in clause 1a above will revert to the Author.

2 Copyright

Ownership of copyright in the Contribution will be vested in the name of the Author. When reproducing the Contribution or extracts from it, the Author will acknowledge and reference first publication in the Volume.

3 Use of Contribution Versions

a) For purposes of this Agreement: (i) references to the "Contribution" include all versions of the Contribution; (ii) "Submitted Manuscript" means the version of the Contribution as first submitted by the Author prior to peer review; (iii) "Accepted Manuscript" means the version of the Contribution accepted for publication, but prior to copy-editing and typesetting; and (iv) "Version of Record" means the version of the Contribution published by the Licensee, after copy-editing and typesetting. Rights to all versions of the Manuscript are granted on an exclusive basis, except for the Submitted Manuscript, to which rights are granted on a non-exclusive basis.

b) The Author may make the Submitted Manuscript available at any time and under any terms (including, but not limited to, under a CC BY licence), at the Author's discretion. Once the Contribution has been published, the Author will include an acknowledgement and provide a link to the Version of Record on the publisher's website: "This preprint has not undergone peer review (when applicable) or any post-submission improvements or corrections. The Version of Record of this contribution is published in [insert volume title], and is available online at https://doi.org/[insert DOI]".

c) The Licensee grants to the Author (i) the right to make the Accepted Manuscript available on their own personal, self-maintained website immediately on acceptance, (ii) the right to make the Accepted Manuscript available for public release on any of the following twelve (12) months after first publication (the "Embargo Period"): their employer's internal website; their institutional and/or funder repositories. Accepted Manuscripts may be deposited in such repositories immediately upon acceptance, provided they are not made publicly available until after the Embargo Period.
The rights granted to the Author with respect to the Accepted Manuscript are subject to the conditions that (i) the Accepted Manuscript is not enhanced or substantially reformatted by the Author or any third party, and (ii) the Author includes on the Accepted Manuscript an acknowledgement in the following form, together with a link to the published version on the publisher's website: "This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms".
Under no circumstances may an Accepted Manuscript be shared or distributed under a Creative Commons or other form of open access licence.
Any use of the Accepted Manuscript not expressly permitted under this subclause (c) is

subject to the Licensee's prior consent.

d) The Licensee grants to Author the following non-exclusive rights to the Version of Record, provided that, when reproducing the Version of Record or extracts from it, the Author acknowledges and references first publication in the Volume according to current citation standards. As a minimum, the acknowledgement must state: "First published in [Volume, page number, year] by Springer Nature".

i. to reuse graphic elements created by the Author and contained in the Contribution, in presentations and other works created by them;

ii. the Author and any academic institution where they work at the time may reproduce the Contribution for the purpose of course teaching (but not for inclusion in course pack material for onward sale by libraries and institutions);

iii. to reuse the Version of Record or any part in a thesis written by the same Author, and to make a copy of that thesis available in a repository of the Author(s)' awarding academic institution, or other repository required by the awarding academic institution. An acknowledgement should be included in the citation: "Reproduced with permission from Springer Nature";

iv. to reproduce, or to allow a third party to reproduce the Contribution, in whole or in part, in any other type of work (other than thesis) written by the Author for distribution by a publisher after an embargo period of 12 months; and

v. to publish an expanded version of their Contribution provided the expanded version (i) includes at least 30% new material (ii) includes an express statement specifying the incremental change in the expanded version (e.g., new results, better description of materials, etc.).

4 Warranties & Representations

Author warrants and represents that:

a)

i. the Author is the sole copyright owner or has been authorised by any additional copyright owner(s) to grant the rights defined in clause 1,

ii. the Contribution does not infringe any intellectual property rights (including without limitation copyright, database rights or trade mark rights) or other third party rights and no licence from or payments to a third party are required to publish the Contribution,

iii. the Contribution has not been previously published or licensed, nor has the Author committed to licensing any version of the Contribution under a licence inconsistent with the terms of this Agreement,

iv. if the Contribution contains materials from other sources (e.g. illustrations, tables, text quotations), Author has obtained written permissions to the extent necessary from the copyright holder(s), to license to the Licensee the same rights as set out in clause 1 but on a non-exclusive basis and without the right to use any graphic elements on a stand-alone basis and has cited any such materials correctly;

b) all of the facts contained in the Contribution are according to the current body of research true and accurate;

c) nothing in the Contribution is obscene, defamatory, violates any right of privacy or publicity, infringes any other human, personal or other rights of any person or entity or is otherwise unlawful and that informed consent to publish has been obtained for any research participants;

d) nothing in the Contribution infringes any duty of confidentiality owed to any third party or violates any contract, express or implied, of the Author;

e) all institutional, governmental, and/or other approvals which may be required in connection with the research reflected in the Contribution have been obtained and continue in effect;

f) all statements and declarations made by the Author in connection with the Contribution are true and correct;

g) the signatory who has signed this Agreement has full right, power and authority to enter into this Agreement on behalf of all of the Authors; and

h) the Author complies in full with: i. all instructions and policies in the Instructions for Authors, ii. the Licensee's ethics rules (available at https://www.springernature.com/gp/authors/book-authors-code-of-conduct), as may be updated by the Licensee at any time in its sole discretion.

5 Cooperation

a) The Author will cooperate fully with the Licensee in relation to any legal action that might arise from the publication of the Contribution, and the Author will give the Licensee access at reasonable times to any relevant accounts, documents and records within the power or control of the Author. The Author agrees that any Licensee affiliate through which the Licensee exercises any rights or performs any obligations under this Agreement is intended to have the benefit of and will have the right to enforce the terms of this Agreement.

b) Author authorises the Licensee to take such steps as it considers necessary at its own expense in the Author's name(s) and on their behalf if the Licensee believes that a third party is infringing or is likely to infringe copyright in the Contribution including but not limited to initiating legal proceedings.

6 Author List

Changes of authorship, including, but not limited to, changes in the corresponding author or the sequence of authors, are not permitted after acceptance of a manuscript.

7 Post Publication Actions

The Author agrees that the Licensee may remove or retract the Contribution or publish a correction or other notice in relation to the Contribution if the Licensee determines that such actions are appropriate from an editorial, research integrity, or legal perspective.

8 Controlling Terms

The terms of this Agreement will supersede any other terms that the Author or any third party may assert apply to any version of the Contribution.

9 Governing Law

This Agreement shall be governed by, and shall be construed in accordance with, the laws of Switzerland. The courts of Zug, Switzerland shall have the exclusive jurisdiction.

Signed for and on behalf of the Author	Print Name:	Date:
Hattan Althebeiti	Hattan Althebeiti	10/03/2022

Address:	
Email: hattan@knights.ucf.edu	

REFERENCES

[1] —. BugTraq by Accenture Security. Online, 2021.

[2] —. all-mpnet-base-v2. Online, June 2021.

[3] —. MITRE's Common Vulnerabilities and Exposures (CVE). Online, 2022.

[4] —. IBM X-Force Exchange. Online, 2022.

[5] —. National Vulnerability Database (NVD). Online, 2022.

[6] —. Microsoft Security Response Center. Online, 2022.

[7] —. National Institute of Standards and Technology. Online, 2022.

[8] —. ChatGPT. Online, 2023.

[9] —. GPT-3.5-turbo. Online, 2023.

[10] —. New and improved embedding model. Online, 2023.

[11] E. Aghaei, W. G. Shadid, and E. Al-Shaer. Threatzoom: Hierarchical neural network for cves to cwes classification. In *Security and Privacy in Communication Networks - 16th EAI International Conference, SecureComm 2020, Washington, DC, USA, October 21-23, 2020, Proceedings, Part I*, volume 335 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 23–41. Springer, 2020.

[12] A. Akbik, T. Bergmann, D. Blythe, K. Rasul, S. Schweter, and R. Vollgraf. FLAIR: An easy-to-use framework for state-of-the-art NLP. In *NAACL 2019, 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 54–59, 2019.

[13] A. Alabduljabbar, A. Abusnaina, Ü. Meteriz-Yildiran, and D. Mohaisen. Automated Privacy Policy Annotation with Information Highlighting Made Practical Using Deep Representations. In *ACM CCS*, pages 2378–2380, 2021.

[14] A. Alabduljabbar, A. Abusnaina, Ü. Meteriz-Yildiran, and D. Mohaisen. TLDR: Deep Learning-Based Automated Privacy Policy Annotation with Key Policy Highlights. In *ACM WPES*, pages 103–118, 2021.

[15] A. Alabduljabbar, A. Abusnaina, Ü. Meteriz-Yildiran, and D. Mohaisen. Poster: Mujaz: A summarization-based approach for normalized vulnerability description. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[16] H. Alasmary, A. Anwar, J. Park, J. Choi, D. Nyang, and A. Mohaisen. Graph-based comparison of iot and android malware. In *Computational Data and Social Networks - 7th International Conference, CSoNet 2018, Shanghai, China, December 18-20, 2018, Proceedings*, volume 11280 of *Lecture Notes in Computer Science*, pages 259–272. Springer, 2018.

[17] R. Alkhadra, J. Abuzaid, M. AlShammari, and N. Mohammad. Solar Winds Hack: In-Depth Analysis and Countermeasures. In *12th International Conference on Computing Communication and Networking Technologies, ICCCNT 2021, Kharagpur, India, July 6-8, 2021*, pages 1–7, 2021.

[18] H. Althebeiti and D. Mohaisen. Enriching vulnerability reports through automated and augmented description summarization. In *23rd International Conference Information Security Applications, WISA*, volume 13009 of *Lecture Notes in Computer Science*, pages 265–277. Springer, 2022.

[19] A. Anwar, A. Abusnaina, S. Chen, F. Li, and D. Mohaisen. Cleaning the NVD: Comprehensive Quality Assessment, Improvements, and Analyses. In *51st Annual IEEE/IFIP*

International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021 - Supplemental Volume, pages 1–2, 2021.

[20] A. Anwar, A. Abusnaina, S. Chen, F. Li, and D. Mohaisen. Cleaning the NVD: Comprehensive Quality Assessment, Improvements, and Analyses. *IEEE Trans. Dependable Secur. Comput.*, 19(6):4255–4269, 2022.

[21] A. Anwar, A. Khormali, J. Choi, H. Alasmary, S. Choi, S. Salem, D. Nyang, and D. Mohaisen. Measuring the Cost of Software Vulnerabilities. *EAI Endorsed Transactions on Security and Safety*, 7(23), 2020.

[22] A. Anwar, A. Khormali, D. Nyang, and A. Mohaisen. Understanding the Hidden Cost of Software Vulnerabilities: Measurements and Predictions. In *Security and Privacy in Communication Networks - 14th International Conference, SecureComm 2018, Singapore, August 8-10, 2018, Proceedings, Part I*, volume 254 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 377–395, 2018.

[23] M. Aota, H. Kanehara, M. Kubo, N. Murata, B. Sun, and T. Takahashi. Automation of vulnerability classification from its description using machine learning. In *IEEE Symposium on Computers and Communications, ISCC 2020, Rennes, France, July 7-10, 2020*, pages 1–7. IEEE, 2020.

[24] S. Bird and E. Loper. Natural language processing with python. Online, 2023.

[25] B. E. Boser, I. Guyon, and V. Vapnik. A training algorithm for optimal margin classifiers. In D. Haussler, editor, *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992, Pittsburgh, PA, USA, July 27-29, 1992*, pages 144–152. ACM, 1992.

[26] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[27] S. Brin and L. Page. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Comput. Networks*, 56(18):3825–3833, 2012.

[28] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[29] D. Cer, Y. Yang, S. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar, B. Strope, and R. Kurzweil. Universal Sentence Encoder for English. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018: System Demonstrations, Brussels, Belgium, October 31 - November 4, 2018*, pages 169–174, 2018.

[30] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[31] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, and R. Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 785–794. ACM, 2016.

[32] T. Chen and C. Guestrin. Xgboost: A scalable and accurate gradient boosting system. Online, 2023.

[33] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022.

[34] P. F. Christiano, J. Leike, T. B. Brown, M. Martic, S. Legg, and D. Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 4299–4307, 2017.

[35] K. Clark, M. Luong, Q. V. Le, and C. D. Manning. ELECTRA: pre-training text encoders as discriminators rather than generators. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[36] M. Corporation. Lightgbm: A gradient boosting framework. Online, 2023.

[37] S. S. Das, E. Serra, M. Halappanavar, A. Pothen, and E. Al-Shaer. V2W-BERT: A framework for effective hierarchical multiclass classification of software vulnerabilities. In *8th IEEE International Conference on Data Science and Advanced Analytics, DSAA 2021, Porto, Portugal, October 6-9, 2021*, pages 1–12. IEEE, 2021.

[38] J. Devlin, M. 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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[39] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang. Towards the Detection of Inconsistencies in Public Security Vulnerability Reports. In *USENIX Security Symposium*, pages 869–885, 2019.

[40] A. Fader, L. Zettlemoyer, and O. Etzioni. Open question answering over curated and extracted knowledge bases. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 1156–1165, 2014.

[41] X. Feng, X. Liao, X. Wang, H. Wang, Q. Li, K. Yang, H. Zhu, and L. Sun. Understanding and Securing Device Vulnerabilities through Automated Bug Report Analysis. In *USENIX Security Symposium*, pages 887–903, 2019.

[42] H. Fu, Yao; Peng and T. Khot. How does gpt obtain its ability? tracing emergent abilities of language models to their sources. *Yao Fu's Notion*, Dec 2022.

[43] P. Gage. A new algorithm for data compression. *C Users Journal*, 12(2):23–38, 1994.

[44] D. Gonzalez, H. Hastings, and M. Mirakhorli. Automated Characterization of Software Vulnerabilities. In *International Conf on Software Maintenance and Evolution, ICSME*, pages 135–139, 2019.

[45] H. Guo, Z. Xing, S. Chen, X. Li, Y. Bai, and H. Zhang. Key Aspects Augmentation of Vulnerability Description based on Multiple Security Databases. In *IEEE Annual Computers, Software, and Applications Conference, COMPSAC*, pages 1020–1025, 2021.

[46] H. Guo, Z. Xing, and X. Li. Predicting missing information of vulnerability reports. In *Companion of The 2020 Web Conference 2020, Taipei, Taiwan, April 20-24,2020*, pages 81–82. ACM / IW3C2, 2020.

[47] Help Net Security. Still relying solely on cve and nvd for vulnerability tracking? bad idea. `https://www.helpnetsecurity.com/2018/02/16/cve-nvd-vulnerability-tracking/`, August 2018.

[48] M. Henderson, P. Budzianowski, I. Casanueva, S. Coope, D. Gerz, G. Kumar, N. Mrksic, G. Spithourakis, P. Su, I. Vulic, and T. Wen. A Repository of Conversational Datasets. *CoRR*, abs/1904.06472, 2019.

[49] G. E. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531, 2015.

[50] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. The Curious Case of Neural Text Degeneration. In *International Conference on Learning Representations, ICLR*, 2020.

[51] J. Howard and S. Ruder. Universal Language Model Fine-tuning for Text Classification. In *Annual Meeting of the Association for Computational Linguistics, ACL*, pages 328–339, 2018.

[52] HuggingFace. Massive text embedding benchmark (mteb) leaderboard. Online, 2023.

[53] Information Security Buzz. Why Critical Vulnerabilities Do Not Get Reported In The CVE/NVD Databases And How Organisations Can Mitigate The Risks. Online, August 2018.

[54] M. Iyyer, V. Manjunatha, J. L. Boyd-Graber, and H. D. III. Deep Unordered Composition Rivals Syntactic Methods for Text Classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 1681–1691, 2015.

[55] K. Kanakogi, H. Washizaki, Y. Fukazawa, S. Ogata, T. Okubo, T. Kato, H. Kanuka, A. Hazeyama, and N. Yoshioka. Tracing CVE Vulnerability Information to CAPEC Attack Patterns Using Natural Language Processing Techniques. *Inf.*, 12(8):298, 2021.

[56] K. Kanakogi, H. Washizaki, Y. Fukazawa, S. Ogata, T. Okubo, T. Kato, H. Kanuka, A. Hazeyama, and N. Yoshioka. Tracing CAPEC Attack Patterns from CVE Vulnerability Information using Natural Language Processing Technique. In *Hawaii International Conference on System Sciences, HICSS*, 2021.

[57] S. Karnouskos. Stuxnet worm impact on industrial cyber-physical system security. In *IECON 2011 - 37th Annual Conference of the IEEE Industrial Electronics Society*, pages 4490–4494, 2011.

[58] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 3146–3154, 2017.

[59] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, and E. Herbst. Moses: Open Source Toolkit for Statistical Machine Translation. In *Annual Meeting of the Association for Computational Linguistics, ACL*, 2007.

[60] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems, NIPS*, pages 1106–1114, 2012.

[61] T. Kudo. Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates. In I. Gurevych and Y. Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 66–75, 2018.

[62] P. Kuehn, M. Bayer, M. Wendelborn, and C. Reuter. OVANA: An Approach to Analyze and Improve the Information Quality of Vulnerability Databases. In *International Conference on Availability, Reliability and Security, ARES*, pages 22:1–22:11, 2021.

[63] M. B. Kursa, A. Jankowski, and W. R. Rudnicki. Boruta - A system for feature selection. *Fundam. Informaticae*, 101(4):271–285, 2010.

[64] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 7871–7880, 2020.

[65] P. S. H. Lewis, Y. Wu, L. Liu, P. Minervini, H. Küttler, A. Piktus, P. Stenetorp, and S. Riedel. PAQ: 65 Million Probably-Asked Questions and What You Can Do With Them. *Trans. Assoc. Comput. Linguistics*, 9:1098–1115, 2021.

[66] C.-Y. Lin. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*, pages 74–81, 2004.

[67] B. Liu, G. Meng, W. Zou, Q. Gong, F. Li, M. Lin, D. Sun, W. Huo, and C. Zhang. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In *International Conference on Software Engineering, ICSE*, pages 1547–1559, 2020.

[68] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR*, abs/1907.11692, 2019.

[69] K. Lo, L. L. Wang, M. Neumann, R. Kinney, and D. S. Weld. S2ORC: the semantic scholar open research corpus. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 4969–4983, 2020.

[70] I. Loshchilov and F. Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[71] R. Mihalcea and P. Tarau. Textrank: Bringing order into text. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing , EMNLP 2004, A meeting of SIGDAT, a Special Interest Group of the ACL, held in conjunction with ACL 2004, 25-26 July 2004, Barcelona, Spain*, pages 404–411. ACL, 2004.

[72] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[73] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems*

2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States, pages 3111–3119, 2013.

[74] A. Mohaisen and O. Alrawi. Unveiling Zeus: automated classification of malware samples. In *WWW*, pages 829–832, 2013.

[75] A. Mohaisen, O. Alrawi, and M. Mohaisen. AMAL: high-fidelity, behavior-based automated malware analysis and classification. *Computers & Security.*, 52:251–266, 2015.

[76] A. Mohaisen, A. G. West, A. Mankin, and O. Alrawi. Chatter: Classifying malware families using system event ordering. In *IEEE Conference on Communications and Network Security, CNS 2014, San Francisco, CA, USA, October 29-31, 2014*, pages 283–291. IEEE, 2014.

[77] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang. Understanding the Reproducibility of Crowd-reported Security Vulnerabilities. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 919–936, 2018.

[78] S. Mumtaz, C. Rodríguez, B. Benatallah, M. Al-Banna, and S. Zamanirad. Learning Word Representation for the Cyber Security Vulnerability Domain. In *International Joint Conference on Neural Networks, IJCNN*, pages 1–8, 2020.

[79] P. Nespoli, D. Papamartzivanos, F. G. Mármol, and G. Kambourakis. Optimal Countermeasures Selection Against Cyber Attacks: A Comprehensive Survey on Reaction Frameworks. *IEEE Commun. Surv. Tutorials*, 20(2):1361–1396, 2018.

[80] V. H. Nguyen and F. Massacci. The (un)reliability of NVD vulnerable versions data: an empirical experiment on Google Chrome vulnerabilities. In *ACM Symposium on Information, Computer and Communications Security, ASIA CCS*, pages 493–498, 2013.

[81] A. Niakanlahiji, J. Wei, and B. Chu. A Natural Language Processing Based Trend Analysis of Advanced Persistent Threat Techniques. In *IEEE International Conference on Big Data, BigData*, pages 2995–3000, 2018.

[82] Official Site Of The State Of New Jersey. Colonial Pipeline Incident: Ransomware Impacts and Mitigation Strategies. Online, May 2021.

[83] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[84] J. Pennington, R. Socher, and C. D. Manning. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543, 2014.

[85] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep Contextualized Word Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, pages 2227–2237, 2018.

[86] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. *OpenAI*, 2018.

[87] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[88] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[89] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

[90] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019.

[91] T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilic, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé, J. Tow, A. M. Rush, S. Biderman, A. Webson, P. S. Ammanamanchi, T. Wang, B. Sagot, N. Muennighoff, A. V. del Moral, O. Ruwase, R. Bawden, S. Bekman, A. McMillan-Major, I. Beltagy, H. Nguyen, L. Saulnier, S. Tan, P. O. Suarez, V. Sanh, H. Laurençon, Y. Jernite, J. Launay, M. Mitchell, C. Raffel, A. Gokaslan, A. Simhi, A. Soroa, A. F. Aji, A. Alfassy, A. Rogers, A. K. Nitzav, C. Xu, C. Mou, C. Emezue, C. Klamm, C. Leong, D. van Strien, D. I. Adelani, and et al. BLOOM: A 176b-parameter open-access multilingual language model. *CoRR*, abs/2211.05100, 2022.

[92] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[93] R. Sennrich, B. Haddow, and A. Birch. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, 2016.

[94] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations, ICLR*, 2015.

[95] F. Skopik, G. Settanni, and R. Fiedler. A problem shared is a problem halved: A survey on the dimensions of collective cyber defense through security information sharing. *Comput. Secur.*, 60:154–176, 2016.

[96] K. Song, X. Tan, T. Qin, J. Lu, and T. Liu. MPNet: Masked and Permuted Pre-training for Language Understanding. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[97] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems, NIPS*, pages 3104–3112, 2014.

[98] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 1–9, 2015.

[99] R. Thoppilan, D. D. Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, Y. Li, H. Lee, H. S. Zheng, A. Ghafouri, M. Menegali, Y. Huang, M. Krikun, D. Lepikhin, J. Qin, D. Chen, Y. Xu, Z. Chen, A. Roberts, M. Bosma, Y. Zhou, C. Chang, I. Krivokon, W. Rusch, M. Pickett, K. S. Meier-Hellstern, M. R. Morris, T. Doshi, R. D. Santos, T. Duke, J. Soraker, B. Zevenbergen, V. Prabhakaran, M. Diaz, B. Hutchinson, K. Olson, A. Molina, E. Hoffman-John, J. Lee, L. Aroyo, R. Rajakumar, A. Butryna, M. Lamm, V. Kuzmina, J. Fenton, A. Cohen, R. Bernstein, R. Kurzweil, B. A. y Arcas, C. Cui, M. Croak, E. H. Chi, and Q. Le. Lamda: Language models for dialog applications. *CoRR*, abs/2201.08239, 2022.

[100] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[101] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. Canton-Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[102] 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 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[103] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Inf. Theory*, 13(2):260–269, 1967.

[104] Q. Wang, B. Li, T. Xiao, J. Zhu, C. Li, D. F. Wong, and L. S. Chao. Learning Deep Transformer Models for Machine Translation. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 1810–1822, 2019.

[105] E. Wåreus and M. Hell. Automated CPE Labeling of CVE Summaries with Machine Learning. In *Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA*, pages 3–22, 2020.

[106] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus. Emergent abilities of large language models. *Trans. Mach. Learn. Res.*, 2022, 2022.

[107] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

[108] R. J. Williams and D. Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Comput.*, 1(2):270–280, 1989.

[109] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771, 2019.

[110] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[111] S. Xiao, Z. Liu, P. Zhang, and N. Muennighoff. C-pack: Packaged resources to advance general chinese embedding, 2023.

[112] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 5754–5764, 2019.

[113] S. Yitagesu, X. Zhang, Z. Feng, X. Li, and Z. Xing. Automatic Part-of-Speech Tagging for Security Vulnerability Descriptions. In *International Conference on Mining Software Repositories, MSR*, pages 29–40, 2021.

[114] J. Zhang, Y. Zhao, M. Saleh, and P. J. Liu. PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 11328–11339, 2020.