Dissertation Proposal

Improving Vulnerability Description Using Natural Language Generation

Hattan Althebeiti

Date: March 28, 2023

Department of Electrical and Computer Engineering
University of Central Florida
Orlando, FL 32816

Doctoral Committee:
Dr. David Mohaisen (Chair)
Dr. Murat Yuksel
Dr. Fan Yao
Dr. Cliff Zou
Dr. Yao Li

Hattan Althebeiti

Department of Electrical and Computer Engineering, University of Central Florida (UCF)
4000 Central Florida Blvd., R1-368, Orlando, FL 32816-2362 USA

EDUCATION

PH.D. IN COMPUTER SCIENCE (2020 – CURRENT)
University of Central Florida

M.SC. IN COMPUTER ENGINEERING (2018 – 2020)
University of Central Florida
CGPA: 3.71

B.SC. IN COMPUTER ENGINEERING (2008 – 2013)
Umm Al-Qura
CGPA: 3.48

Contents

Abstract

Software has become a central part of our day-to-day life. Numerous gadgets are deployed within one's premises, and they are expected to grow in the upcoming years with the increased deployment of IoT devices. Moreover, wearable devices such as smartwatches and Virtual Reality (VR) headsets are gaining momentum, and their market is thriving consistently. All these devices are run and managed by software orchestrating various processes and data to complete specific tasks. However, software vulnerabilities have been growing continuously with profound implications. For example, IoT devices are designed to complete a small set of deterministic tasks, and their development is relatively simple and quick. However, hardening their security and implementing best practices becomes less important for businesses, making IoT products teem with vulnerabilities.

Driven by this trend, documenting vulnerabilities and tracking their development becomes crucial for organizations and developers. Vulnerability databases have addressed this issue by storing a record with various properties for each discovered vulnerability. However, their contents suffer from several drawbacks, which we address in our work. In this dissertation, we investigate the flaws associated with vulnerabilities description introduced in public databases and alleviate such weaknesses through Natural Language Processing (NLP).

We start by examining the content of popular vulnerability databases and the possible ways to improve their content. We focus on vulnerability descriptions because they are the main element covering essential details of a vulnerability. We propose a new automated approach, leveraging external sources to enrich the scope and context of a vulnerability description. Moreover, we further enhance our method by considering different paths to improve the enriched description. Our approach seemed to learn how to perform extractive summarization from the collected dataset.

Similarly, we further investigate the need for uniform and normalized structure in vulnerability descriptions. We address this by breaking a description into multiple constituents, representing a particular aspect of the vulnerability. We train a multi-task model to create a new uniform and normalized summary that maintains the necessary content using the extracted features. Our approach proved effective in generating new summaries with the same structure across a collection of various vulnerabilities description.

Finally, we investigate the possibility of extending the utilization of a vulnerability description to perform feature/attribute labeling. More precisely, we study the feasibility of assigning the Common Weakness Enumeration (CWE) attribute to a vulnerability based on its description. CWE provides a standardized language and a common set of definitions for software security weaknesses. We aim to experiment with various approaches and report our results for each method using standard metrics evaluation for classification.

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 [8, 29, 47]. Moreover, vulnerabilities in software have had a catastrophic impact on vendors' profit and reputation [11].

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 [55]. 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 [37]. 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 development and deployment of patches. In 2017 Risk Based Security, also known as VulbDB, reported 7,900 more vulnerabilities than what was reported by CVE [21, 25]. Another concern with the existing framework is that the description provided for vulnerabilities is often incomplete, brief, or does not carry sufficient contextual information [9, 12]. 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.

1.1 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 summarizes the task, target output, technique used, and evaluation metrics for each pipeline. We further elaborate on each

Table 1: Overview of summarization approaches.

§	Task	Target	Technique	Metric
§ 3	Summarization	Enriched description	Augmentation	ROUGE, Human
§ 4	Summarization	Normalized summary	Multi-task	ROUGE, Compression, Human
§ 5	Classification	Common Weakness Enumeration	Classification models	Precision, Recall, F1-Score

pipeline in the following.

***Zad*: Enriching Vulnerability Reports Through Automated and Augmented Description Summarization (§ 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 (§ 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 (§ 5).**

When a vulnerability is discovered, it is populated by several attributes after being analyzed. One of these attribute is Common Weakness Enumeration (CWE). The CWE feature is significant because it can assort many vulnerabilities under one big umbrella class represented by the research concept and group other vulnerabilities under a more specific class of CWE. The accuracy of classifying a vulnerability into its respective CWE aid in analyzing similar exposures and possibly defining a framework for different groups that characterizes their features and mitigation techniques

Tasneef aims to utilize a vulnerability description to predict the CWE feature and automate this process based on the description only. CWE defines the weakness that can be exploited and trigger this vulnerability with more context and details. Therefore, we can frame the problem as a classification task, given a vulnerability description, what is the corresponding CWE associated with it. However, one must be aware of a few facts related to the CWE. (1) The CWEs are organized in a hierarchical structure represented as a tree in which each node is a type of CWE. (2) The top-level nodes represent a general category or a family type of CWE, and each child represents a more targeted and specific case of that family. This hierarchy continues until it reaches a leaf CWE node. (3) CWE tree does not necessarily have the same level or number of children, indicating some branches have more depth/children than others. Each level within the tree presents a more specific abstraction of the weakness with a description and detailed explanation that can be correlated with the vulnerability description.

2 Related Work

The contents of vulnerability databases have been scrutinized across a range of features in previous works. For instance, NVD has been analyzed in [10,45], where it is shown to be inaccurate. Given the nature of NVD, most studies rely on statistical or deep-learning methods to carry out their analysis. Statistical methods utilize feature (e.g., word) frequency to derive a numerical representation of a vulnerability summary, capturing specific patterns/characteristics. In contrast, deep learning methods utilize different neural network architectures to learn the underlying features of a summary (unsupervised) or to approximate the input to the target label (supervised). However, most studies focus on identifying software names/versions, predicting vulnerable versions, or detecting inconsistencies between different databases. We divide the studies in this space into three broad categories.

2.1 Contents Enrichment

The first category of studies focused on improving/enriching the content of NVD [20, 33] or detecting inconsistencies against third-party reports [15]. 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.* [20] 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.* [33] 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.* [15] 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 corre-

sponding 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.

2.2 Vulnerability Documentation

The diversity of vulnerabilities and their associated threats require different forms of documentation and targeted embedding, which we distinguish as the second category of works explained next.

Niakanlahiji *et al.* [46] 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.* [17] 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.

Developing domain-specific embedding for vulnerabilities allows for better representation, improving the model's performance for various tasks, as shown in Mumtaz *et al.* [43]. Similarly, Yitagesu *et al.* [65] 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.

2.3 Vulnerability Classification

The last category is studies that map/classify vulnerability to a particular attribute. For example, Gonzalez *et al.* [19] used NLP and machine learning approaches to map Vulnerability Description Ontology (VDO) to a vulnerability based on a vulnerability description. Similarly, Kanakogi *et al.* [27, 28] 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.

Table 2: Comparison with similar work

Work	Task	Target	Architecture	Metric	Score	Human Evaluation
Dong et al. [15]	NER	Software names	GRU	Accuracy	0.9764	✘
Kanakogi et al. [28]	Mapping	CAPEC	Embeddings	Recall@10	0.7500	✘
Gonzalez et al. [19]	Classification	VDO Labels	Majority Vote	RBF	0.4200	✘
Wareus et al. [62]	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	✔

Wåreus *et al.* [62] 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.

3 *Zad*: Enriching Vulnerability Reports Through Automated and Augmented Description Summarization

3.1 Summary of Completed Work

In this work, We leverage publicly available sources to enhance and enrich vulnerability description through data augmentation. Our method relies on additional resources (hyperlinks) for each vulnerability presented in the National Vulnerability Database (NVD). The text within each resource is fetched and passed through multiple filters to extract relevant paragraphs that could contribute to our dataset. We fine-tune two pre-trained models that excel in summarization tasks using our curated dataset to generate enriched summaries. The augmented text guides the models to minimize the semantic gaps between the text and the description with a length constraint on the generated summary. We report initial and promising results using computational metrics and human metrics that are devised to evaluate the accuracy of the generated summary. Moreover, we manually labeled 100 samples from the dataset by writing their summaries from the augmented text. Our results show that the label-guided method outperforms the description-guided method.

3.2 Introduction

Developing the appropriate vulnerability mitigation techniques is a challenging task [42], 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 [9].

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 [15]. 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 [9, 15, 45]. 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.

3.3 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 (§3.3.1). We then present *Zad*'s pipeline, elaborating on each subcomponent in (§3.3.2). Next, we introduce a fundamental component of *Zad*, sequence encoder in (§3.3.3), explaining its role and the different architectures available for deployment. Finally, we present two pre-trained models commonly used for summarization in (§3.3.4).

3.3.1 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 [40,48] retain the same embedding for a word regardless of its context. (2) Dynamic word embeddings [49] 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

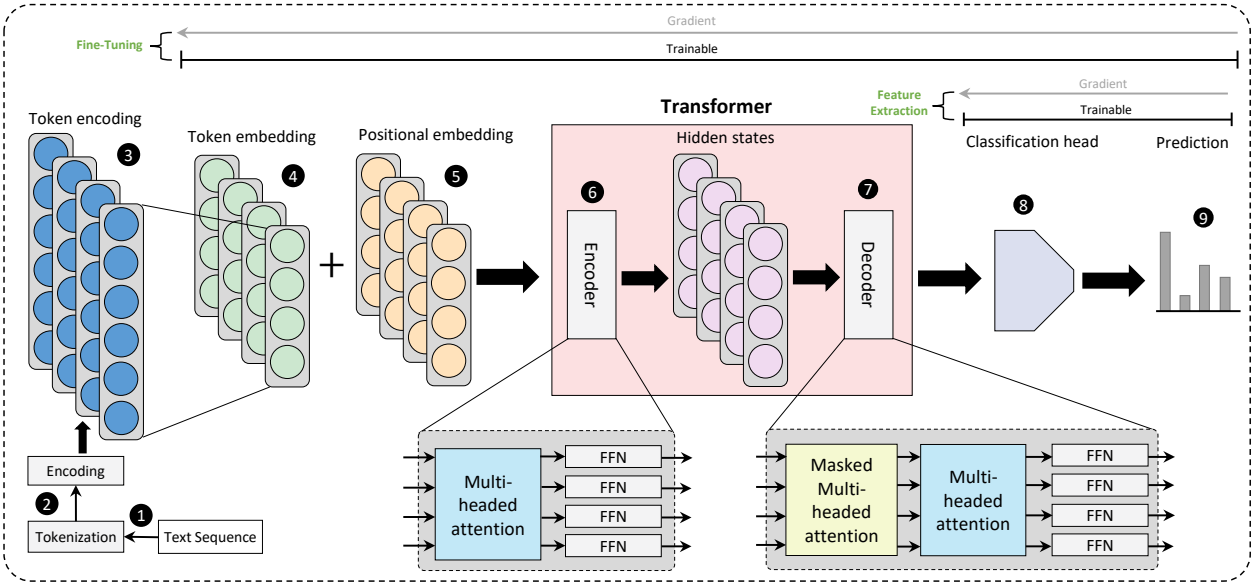

Figure 1: *Zad* pipeline. The pipeline consists of multiple steps to transform the text $X_{1:n}$ from a series of tokens into numerical representations. The encoder component then encodes them to incorporate their context, improving their embeddings through self-attention. The decoder's role is to deploy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

yet similar to the original description. However, vulnerability descriptions/reports contain crucial information, making their evaluation critical. Summarization tasks typically rely on word overlaps 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.

3.3.2 *Zad*'s Pipeline

Zad's pipeline is depicted in Figure 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 $\mathcal{F}$ 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:

$$\min_{|x|<\delta} \left(\mathcal{L}\left(\mathcal{F}_\theta(x, y)\right)\right) \tag{1}$$

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 (§3.4). Therefore, the model has two options for fine-tuning: the description-guided or the label-guided dataset. Seq2Seq models [57] 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 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 3.4. 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) [53], and T5 uses SentencePiece [32], which are both sub-word tokenizers, discussed next.

Byte Pair Encoding (BPE). was initially developed as a text compression algorithm [18]. 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 [50], GPT-2 [51], and RoBERTa [38]. 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.

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

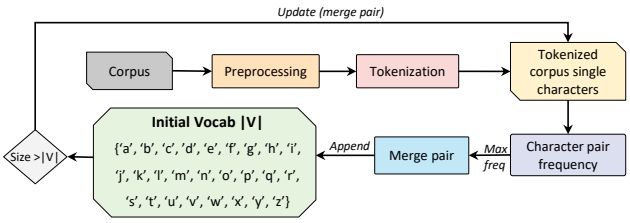

Figure 2: A schematic structure for training BPE tokenizer

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 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:

$$P(\mathrm{x}) = \prod_{i=1}^{m} p\left(x_i\right), \forall i \; x_i \in \mathcal{V} \tag{2}$$

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:

$$loss = \sum freq \times (-\log(P(\text{ word }))) \tag{3}$$

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 [60]. 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 [40, 41] or Glove [48]. 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

$$P(x^{(t+1)}|x^{(t)},...,x^{(1)}) = \prod_{t=1}^{T} P(x^{(t+1)}|x^{(t)},...,x^{(1)}) \tag{4}$$

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

$$\log(P(x^{(t+1)}|x^{(t)}, ..., x^{(1)})) = \sum_{t=1}^{T} \log(P(x^{(t+1)}|x^{(t)}, ..., x^{(1)})) \tag{5}$$

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.

3.3.3 Sentence Encoders

As explained in 3.3.1, 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) [13] 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 [13] 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) [26] 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 [56] is a model that leverages the advantages presented in two famous pre-trained models: BERT [14], and XLNET [64]. 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 [16, 22, 35, 39], 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.

3.3.4 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 [31,54,58], 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 [24] and adapted afterward in the transformer models. Pre-trained language models inherit the original transformer architecture [59] 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 [61]. 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 [14], the decoder only; e.g., GPT (Generative Pre-trained Transformer) [50], 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 [34], T5 [52], and Pegasus [66], 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.

3.4 Dataset and Data Augmentation

3.4.1 Dataset Collection and Preprocessing

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*> <*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 3.3.3. 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:

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

.

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

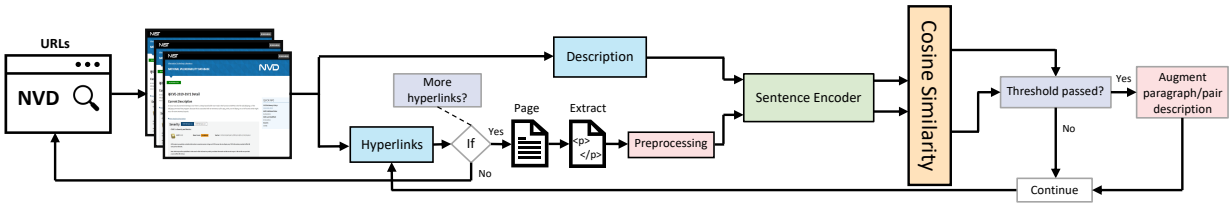

Figure 3: Our data collection pipeline, constituting the first part of *Zad*. The pipeline starts with a predefined set of URLs associated with CVEs (enumerated from NVD) and scrapes the description and its hyperlinks. For each hyperlink, the pipeline invokes the extraction of paragraphs, preprocesses 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.

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, upon which our dataset is ready.

Figure 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. 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 consis-

Table 3: Datasets and their high-level characteristics.

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

tency 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 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.

3.4.2 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 summaries. 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.

3.5 Evaluations

Statistical Analysis. After assembling the three datasets as shown in Table 3, 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 4: 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 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:

$$F1\text{-}Score = 2 \times \frac{precision \times recall}{precision + recall} \tag{6}$$

Table 4 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 5 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

Table 5: 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

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 Table 6.

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.

Table 6: Human evaluation: **F**luency, **Com**pleteness, **Cor**rectness, and **U**nderstanding.

Model	F	Cm	Cr	U
BART	2.69	**2.15**	**2.16**	**2.58**
T5	**2.72**	2.07	2.04	2.57

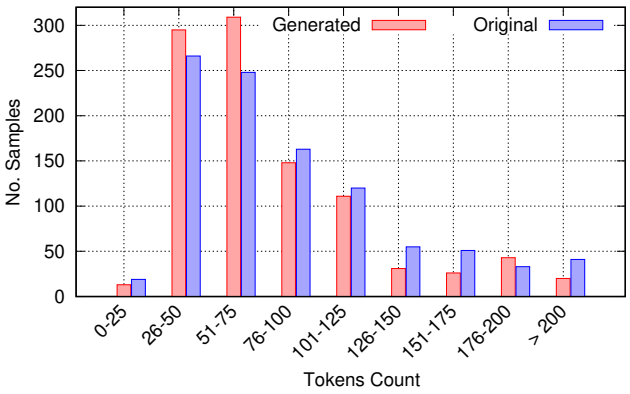

Figure 4: Tokens distribution. The distribution shows the number of instances and their token counts in the original and generated summaries.

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.

3.5.1 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 4. we see in Figure 4 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.

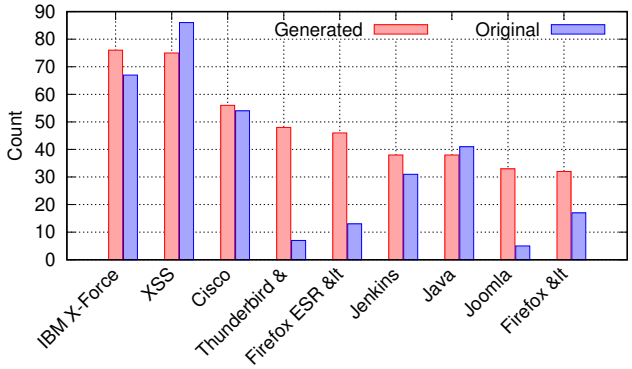

Figure 5: Top names presented in the original and generated summary.

Next, we analyze the named entity recognition shown in Figure 5. 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 6 and Table 7. 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

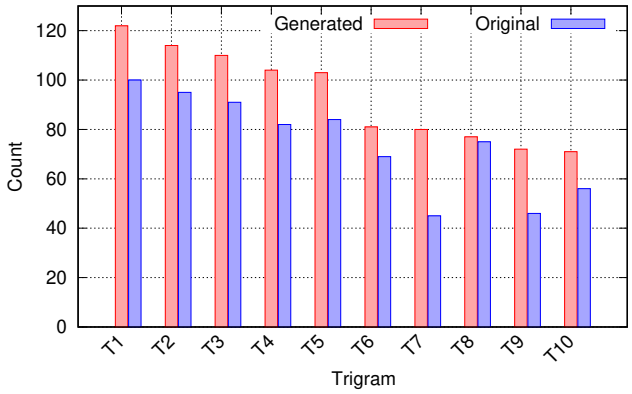

Figure 6: Trigrams count. The generated summary has the same trigram repeated more than the original, indicating overfitting.

t

Table 7: The full sets of the different trigrams in Figure 6 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

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 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.

3.5.2 Analysis of Label Guided Dataset

In this section, we report our results using the label-guided dataset described in section 3.4.2. 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 8. 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.

Next, we report our evaluation using human metrics introduced earlier. For both models, fluency, and completeness achieved good scores as opposed to correctness and understanding with lower scores, especially for T5. Both models generated sound and fluent summaries, including most details from the target summaries. In contrast, correctness is the most affected metric as the

Table 8: 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

details included were incorrect (e.g., wrong software version or bug description) from the created summary. Understanding was problematic because the generated summary needed more context, or the generated sentences were sparsed.

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

3.6 Summary and Work to be Completed

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* introduce 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.

Our future work intends to improve the dataset's quality and enhance the augmented text by enforcing diversity using word frequency. We justify our approach by hypothesizing that the low

score obtained upon evaluating the original dataset was due to the augmented text quality. We believe that reducing this text into a few sentences that capture most of the augmented text could benefit the model. Moreover, it will shed light on the impact of having more data that might be redundant or uninformative to the model.

4 *Mujaz*: A Summarization-based Approach for Normalized Vulnerability Description

4.1 Summary of Completed Work

In this work, we present *Mujaz*, a new multi-task system that exploits pretraining language models to tackle vulnerability description summarization and normalization. We assess *Mujaz* using a parallel corpus emphasizing three different features. *Mujaz* was able to generate coherent summaries with 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.

4.2 Introduction

The information in CVE/NVD varies significantly from the information in third-party reports for the same vulnerabilities [15]. Moreover, the vulnerability information in CVE/NVD could be incorrect, incomplete, or outdated, and those issues could happen for various reasons and circumstances [10, 11]. 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 [44].

Organizations addressed inconsistency issues by creating their repositories for documenting vulnerabilities [4,6]. 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.* [15] 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.* [33] 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 ear-

lier. 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.

4.3 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 in §4.2, *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

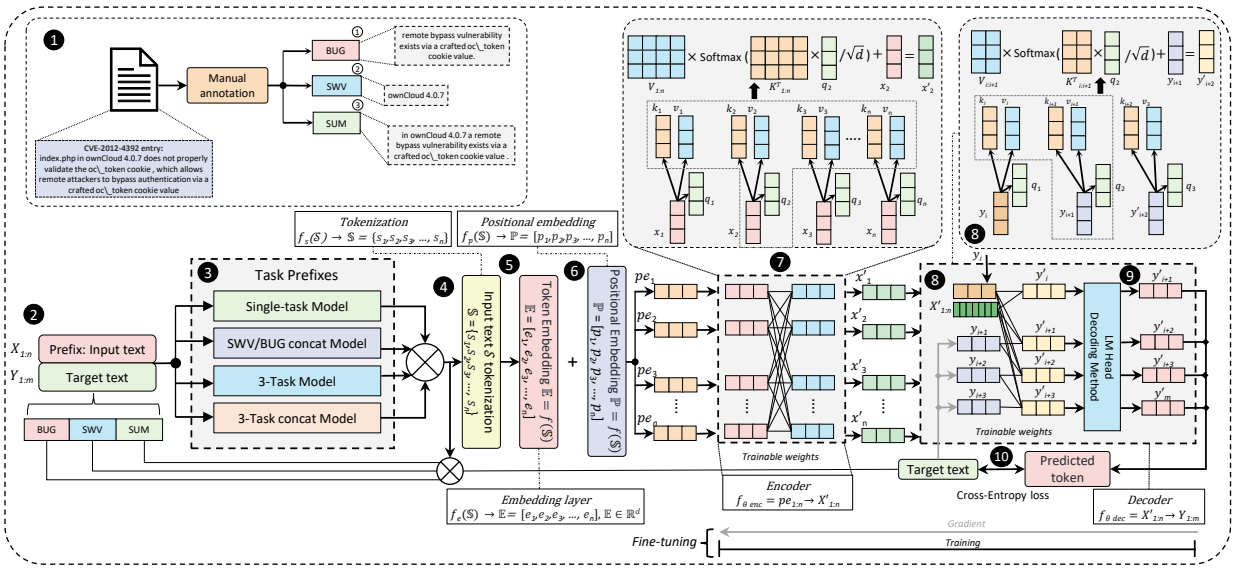

Figure 7: Mujaz Multi-task Pipeline: The first phase includes dataset curation to build the prospective dataset. The second phase shows the input text represented as a CVE description and one of the extracted features to trigger task-specific training, allowing a model to learn multiple tasks simultaneously.

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.

4.4 *Mujaz*'s Pipeline

4.4.1 High-level Overview

Mujaz follows a conventional architecture of an encoder/decoder-based transformer, as shown in Figure 7. 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 prepro-

cessing techniques to ensure the high quality of the input text and the extracted features. The second phase 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.

4.4.2 Pipeline: Technical Details

For convenience, we describe the internal architecture of the original Transformer presented in [59] within our pipeline.

Architecture Overview. The transformer is depicted in Figure 7 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 7 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 7, 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 vulner-

ability 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 4.5.

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 [53], while T5 uses SentencePiece [32]; 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 [40,41] or Glove [48]. 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:

$$y_i = \sum_j \text{softmax}({x_i}^T x_j) x_j, \tag{7}$$

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:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{8}$$

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:

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

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 [63]. The decoder uses unidirectional 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:

$$P(x_i|x_{1:i-1}) = \frac{exp(u_i)}{\sum_j exp(u_j)}. \tag{9}$$

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 [57] 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 [23] chooses a small set of tokens whose cumulative probability exceeds a predefined probability p, which is a hyperparameter; this is:

$$\sum_{x \in V^{(p)}} P(x_i|x_{1:i-1}) \geq p. \tag{10}$$

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

4.5 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 7, 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 [15].
- 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.

4.5.1 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 §4.3). However, in our evaluation (see §4.7) 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.

4.5.2 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.

4.5.3 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.

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.

4.6 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 [15]. A traditional tokenization system like *moses* [30] 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 [15] 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 [15] 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

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.

Table 9: Examples of simple task labels

input description are in Table 9.

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 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.* [15] 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.

4.7 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 8 shows our evaluation metrics.

4.7.1 Metrics

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

ROUGE [36]. 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

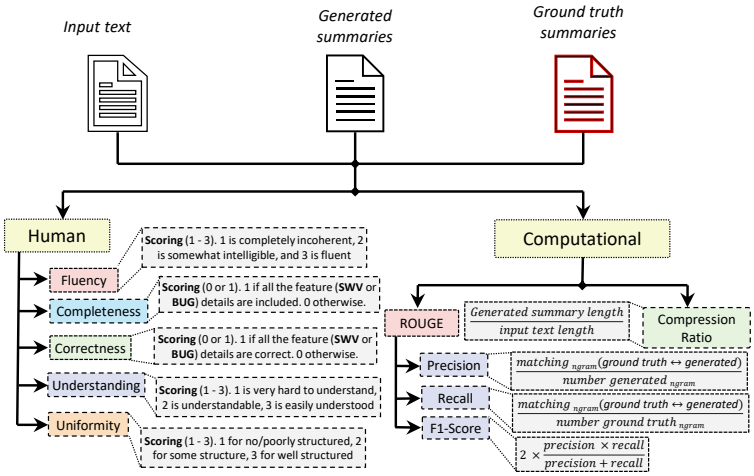

Figure 8: Evaluation metrics

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 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-*

Table 10: 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

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 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.

4.7.2 Results

Computational Evaluation. We first report the score for all computational metrics in Table 10. 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.

Table 11: 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

As shown in Table 10, 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

Table 12: 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**
	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

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 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 11.

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 12. The 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 12, 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.

4.8 Discussion

4.8.1 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.

4.8.2 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.

SUM Task. First, the model generated summaries with punctuation and context, producing coherent summaries. Moreover, it also learned keywords and used them in the prediction. However, the model still suffers from several shortcomings. First, it 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. Overall, the model produced correct, concise, and easy-to-understand summaries.

Concatenation Models. We observe that both models do not produce punctuation, although the 3-Task Concatenation 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.

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 subcomponents 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.

4.8.3 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, achiev-

ing 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. (10). 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.

4.9 Summary and Work to be Completed

In this work, we proposed *Mujaz*, a multi-task system that leverages vulnerability description and breaks it into smaller and independent constituents, representing a self-contained aspect of the description. A parallel corpus is curated to designate three different features, typically embodied in a vulnerability. *Mujaz* exploits this structure to generate a normalized and uniform description by learning the connection between the extracted features. Our curated dataset is applicable to single-task models. However, using a multi-task model provides more deployment options. *Mujaz* was evaluated using our newly designed human metrics along with computational metrics.

However, we have not tested *Mujaz* on all the possible deployment options. Therefore, we intend to extend our experiments by training a 3-Task model. The 3-Task Model uses all of the available information to it 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.

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

5.1 Introduction and Motivation

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.

5.1.1 Problem Statement

There are several problems with manually assigning attributes/features to a vulnerability. First, those attributes are assigned manually, which is time-consuming and labor-intensive as those vulnerabilities must be analyzed thoroughly, involving more information than the reported description. Second, it requires understanding and expertise, which could be subjective and dependent on the

analyst's background and level of knowledge, introducing inconsistency or disagreement on the assigned attribute. Third, it is error prone as a vulnerability might be easily misclassified with the incorrect/inaccurate attribute given insufficient details in the reported description. This issue is noted when reviewing the historical changes of a specific vulnerability.

Our goal is to address the challenges involved in manually assigning attributes to vulnerability descriptions. To achieve this, we propose several models that automate the assignment of these attributes. By doing so, we hope to reduce the workload and potential errors associated with manual assignments. We also recognize the need for an automated solution to handle the increasing rate of discovered vulnerabilities and the need for reexamination. Therefore, we argue that our proposed solution is essential in streamlining the vulnerability management process. Specifically, we plan to automate the CWE labeling process for each CVE entry by identifying the most closely related CWE based on the CVE description. Additionally, we aim to predict a series of CWEs, leading up to the top-level research concept. This approach will enable a more comprehensive understanding of the vulnerabilities and facilitate better mitigation strategies.

5.2 Design Flow

Tasneef exploits vulnerability description offered in public databases to predict its corresponding CWE label. Figure 9 shows a schematic pipeline for *Tasneef*. Given the sophistication of its hierarchical structure, predicting the correct label at the leaf node with high accuracy becomes challenging, necessitating the need for comprehensive and detailed analysis to discover any possible trend or correlation between the CVE description and its corresponding CWE label. Feature extraction is the next step, as we may use them to guide our models and produce better results. We test several classification models and use standard evaluation metrics to evaluate them accordingly. Our data is collected from reliable sources such as NVD to avoid any discrepancies presented in third-party repositories.

5.3 Work to be Done

We chose CWE labeling because the data can be viewed from different perspectives. In addition, it contains several features, text, and structural organization, which allows us to test various classification models based on different elements. Moreover, CWE classification can be regarded as a multi-label classification problem. Each entry has multiple labels representing the branch for CWEs from the root level down to the leaf node. In contrast, it can be considered as a hierarchical classification problem using a single classifier or a chain of classifiers to predict each level, giving us several possibilities to discover the most accurate approach.

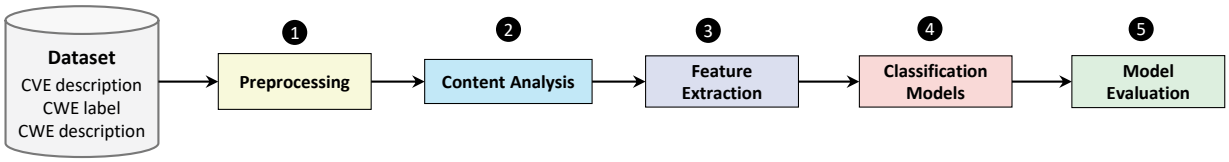

Figure 9: Tasneef schematic pipeline (simplified)

5.3.1 Implementation

We intend to implement three approaches: a pre-trained model, Naive-bays, and Graph Neural Network (GNN). A pre-trained model is a deep learning model already trained on a large dataset and can be used as a starting point for solving similar problems. The goal is to leverage the knowledge gained during the training process to improve the performance of new models. On the other hand, Naive Bayes is a popular algorithm commonly used for classification in natural language processing. It is a probabilistic model that uses Bayes' theorem to calculate the probability of a document belonging to a particular class based on the presence of certain words or features in the document. Finally, our last approach is using GNN to learn the representation of each node within the tree and to use the updated embedding for classification. A typical GNN consists of multiple layers, each applying a message-passing mechanism to propagate information between nodes in the graph. In each layer, the GNN aggregates the features of the neighboring nodes and updates the features of each node accordingly.

5.3.2 Evaluation

Our evaluation metrics are straightforward since it is a classification task, and they are as follows: accuracy, precision, recall, and F-1 score. Accuracy is the fraction of correct prediction to the model's total predictions. Precision and recall measure the accuracy concerning the true positive samples. Precision is the fraction of the true positive to the total samples predicted as positive, including true and false positives. Precision measures how often positive labels were predicted correctly. In contrast, recall is the fraction of the true positive to the number of true positives plus the false negative predictions. Recall measures how often the model identifies the correct label. Finally, The F1 score is the harmonic mean of precision and recall, considering both precision and recall.

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] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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 Sym-*

posium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018, pages 919–936, 2018.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

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

[61] 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.

[62] 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.

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

[64] 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.

[65] 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.

[66] 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.