Mujaz: A Summarization-based Approach for Normalized Vulnerability Description

Hattan Althebeiti, Brett Fazio, William Chen, Jamen Park and David Mohaisen, *Senior Member, IEEE*

Abstract—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 the prior works improve consistency, unifying and standardizing vulnerability descriptions is mostly unexplored. In this paper, 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* employs 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. Experts evaluation was conducted to investigate *Mujaz*'s effectiveness, providing simplified, coherent, and understandable descriptions.

Index Terms—Vulnerability, Neural networks, Datasets, Natural language generation, Text tagging

1 Introduction

The 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 [1]. For instance, vulnerabilities in software have had a catastrophic impact on vendors' profit and reputation, as demonstrated in [2]. 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 information [4]. The National Vulnerability Database (NVD) [5], managed by NIST [6], is synchronized with MITRE's CVE and seeks to structure CVE data to help inform stakeholders through a unified threat information sharing.

The information in CVE/NVD varies [7], calling for normalization. As the number of discovered and disclosed vulnerabilities increases over time, manually addressing those inconsistencies for a standard becomes significantly impractical, necessitating the development of an automated solution to unify vulnerabilities' attributes in a single, concise, and accurate context without any conflicts [2], [8], [9].

Researchers addressed the inconsistency issues in vulnerability reports by analyses, understanding, and mitigation using various Natural Language Processing (NLP) techniques [7], [9], [10], [11], [12], [13], [14], [15], [16]. For instance, Dong *et al.* [7] introduced a system that detects inconsistencies between NVD/CVE and third-party reports, however, their vulnerability type representation is biased toward memory corruption. Kühn *et al.* [16] 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 Common Vulnerability Scoring System (CVSS) score, although they only utilized information within the vulnerability itself, limiting the improvement the system can achieve.

Organizations addressed inconsistency issues by creating their repositories for vulnerabilities [17], [18], but they are primarily concerned with their products and hardly address other vulnerabilities in other products, particularly those without such initiatives. In academia, researchers addressed the inconsistency issues in vulnerability reports using various Natural Language Processing (NLP) techniques. Kühn *et al.* [16] developed a system to update the NVD database and improve the information quality using structural data (e.g., CVSS score). Dong *et al.* [7] introduced a system that detects inconsistencies in vulnerabilities. However, their dataset lacks vulnerability type representation and is biased toward memory corruption vulnerabilities. The lack of normalized vulnerability reporting costs security analysts significantly in terms of their man-hours for understanding the vulnerability reports. Given this gap, 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. We curate a *parallel* dataset that consists of three features extracted from the description, and each is viewed as a task to train a

H. Althebeiti and D. Mohaisen are with the Department of Computer Science at the University of Central Florida (UCF). B. Fazio is with TwoSigma. W. Chen is with the Carnegie Mellon University. J. Park is with the Department of Software at the Kyung Hee University. This work was done while all authors were at the University of Central Florida. D. Mohaisen is the corresponding author (mohaisen@ucf.edu).

multi-task model independently or jointly with another feature to produce the target summary.

The rich semantics of a vulnerability description necessitate curating such a dataset because it typically includes the software name/version, the type of bug/threat, and a summary describing their interactions. We offer a multi-task model to *abstractively* summarize and normalize CVE description to improve quality. 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 using our curated dataset and deploy 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 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 their inconsistencies. 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 bug types 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 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.

Contribution. Our contributions are as follows: ① We present *Mujaz*, an NLP-based pipeline customized on a parallel dataset and fine-tuned on two pre-trained models for vulnerability summarization with different parameters and settings to boost its performance. ② We evaluate *Mujaz* across standard metrics commonly used for summarization along with new metrics that we introduce for human evaluation to judge various aspects of the generated summaries. ③ A byproduct of our work is the curated new parallel dataset that consists of three features that define a vulnerability, which will be released along with the code upon publication. ④ Our experts' evaluations highlight the effectiveness of *Mujaz* in producing simplified, coherent, and easy-to-understand summaries.

Organization. In section 2, we present the related work. In section 4, we present *Mujaz*'s pipeline, including challenges, design overview, and details. In section 5, we present our evaluation. In section 6, we discuss our findings, followed by a user study to support the efficacy of our approach in section 7, and conclusion in section 8.

2 Related Work

Vulnerability databases have been scrutinized in the prior works. For instance, NVD has been analyzed in [9], [22], where it is shown to be inaccurate. Given the nature of NVD, most studies rely on statistical or deep learning models 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 major categories.

Contents Enrichment. The first category of studies focused on improving/enriching the content of NVD [16], [23] or detecting inconsistencies against third-party reports [7]. 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.* [23] extracted key features from X-Force Exchange [18] and SecurityFocus [24] to enrich CVE descriptions. While this improves coverage, it does not tackle description normalization. Moreover, without ground truth, the supplemented descriptions may inherit inconsistencies from the sources. Kuehn *et al.* [16] proposed OVANA, which leverages vulnerability attributes and NER-extracted features to predict CVSS scores and update the NVD. While it improves Information Quality, results varied across datasets, and the approach does not enforce consistency or guarantee description accuracy.

To resolve inconsistencies between software names and versions in NVD and third-party reports, Dong *et al.* [7] proposed VIEM, which combines a NER-based module and a Relational Extractor (RE). The NER module uses word and character embeddings to identify software names/versions in descriptions, while the RE module links scattered mentions to the correct entity. VIEM effectively addresses name/version inconsistencies but is limited to memory corruption vulnerabilities and does not generate normalized vulnerability descriptions.

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.* [25] proposed SECCMiner to analyze Advanced Persistent Threat (APT) reports. SECCMiner uses techniques such as Part-of-Speech (PoS) tagging and Context-Free Grammar (CFG) to extract Noun-Phrases (NP)

TABLE 1: Comparison with Prior Studies

Work	Task	Target	Architecture	Metric	Score	Human Eval.
Dong et al. [7]	NER	Software names	GRU	Accuracy	0.98	✘
Kanakogi et al. [19]	Mapping	CAPEC	Embeddings	Recall@10	0.75	✘
Gonzalez et al. [20]	Classification	VDO Labels	Majority Vote	RBF	0.42	✘
Wareus et al. [21]	NER	CPE Labels	Bi-LSTM	F1-Score	0.86	✘
Our work	Summarization	Normalized Summary	Multi-task T5	F1 Score	0.85	✔

from the reports and then uses count-based methods to measure the importance of NP in a report. 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.* [26] 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 a better representation, improving the model's performance for various tasks [27]. Similarly, Yitagesu *et al.* [28] built a targeted embedding using PenTreebank (PTB) to train a BiLSTM network to tag key concepts and technical tokens, creating an annotated corpus from a vulnerability description. Although this line of work utilizes some vulnerability information, it does not address the shortcomings in vulnerability databases or aim to fix them.

Vulnerability Classification. These studies map a vulnerability to a particular attribute. Gonzalez *et al.* [20] used NLP and machine learning approaches to map Vulnerability Description Ontology (VDO) to a vulnerability based on a vulnerability description. Similarly, Kanakogi *et al.* [19], [29] 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.

Wåreus *et al.* [21] 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 a Conditionally Random Field (CRF) in the last layer to predict the corresponding CPEs from the text description.

Compared to them, *Mujaz* stands out as we focus on the descriptive summary of a vulnerability rather than the structural information. Moreover, *Mujaz* could be easily adapted to various vulnerability reports, giving it generalization features. Table 1 shows some of the previous works along with their features (task, target, architecture, metrics, performance, and whether the human evaluation is used or not), in contrast to our work.

3 Background

In developing *Mujaz*, we leverage pre-trained models and, for assessment, employ two large language models for comparative evaluation. Accordingly, we review these techniques in this section.

3.1 Pre-trained Models

Pre-trained models are language models trained on large corpora with unsupervised objectives to learn patterns in text [30]. Most rely on transformers [31], an architecture that outperformed earlier baselines, e.g., BERT [32], ALBERT [33], BART [34], RoBERTa [35], and GPT models [30], [36], [37]. Transformers eliminate recurrence in RNNs by using self-attention to capture dependencies while processing inputs in parallel. Pre-trained models also leverage transfer learning, introduced in vision [38], [39], [40] and NLP [41], enabling knowledge transfer to new tasks.

Language models are categorized by training objective: *causal* models predict the next token from prior context, while *masked* models predict missing tokens given both sides of context. Pre-training produces weights that can be reused across tasks, while fine-tuning adjusts them for specific downstream objectives. We consider BART [34] and T5 [42] due to their strong performance in summarization and encoder–decoder architectures, required in Section 4.1.

BART. The Bidirectional and Auto-Regressive Transformer (BART) [34] is trained with a masked language modeling objective. It corrupts input text using a noising function and reconstructs it via a transformer encoder–decoder. The encoder processes corrupted text, while the decoder predicts the missing spans. Although not inherently multi-task, we fine-tuned separate BART instances for different features, later combining outputs into the target summary. This approach is resource-intensive since BART has 140M parameters and requires training distinct models with independent weights and hyperparameters.

T5. The Text-to-Text Transfer Transformer (T5) [42] is a multi-task model that reformulates every NLP task into a text-to-text format, requiring only task-specific prefixes. Like BART, it corrupts and reconstructs text, but corruption is applied to spans (15% of text, average span length 3), which yielded optimal performance. Unlike BART, a single T5 instance consolidates multiple tasks with shared weights and hyperparameters, making training more efficient. However, T5 is larger (220M parameters vs. BART's 140M). In our work, we fine-tuned the base versions of both models due to the significant costs of larger variants.

3.2 Large Language Models

Large Language Models (LLMs) extend Pre-trained Language Models (PLMs) by scaling model size and training data, enhancing downstream performance and enabling new capabilities without changing pre-training objectives or architecture. Examples include ChatGPT [43], Gemini [44], and LLaMA 3.1 [45], which can generate human-like text

and engage in dynamic conversations. LLMs differ from PLMs in three key aspects: (1) *In-Context Learning*: adapting to new tasks from natural language instructions without retraining; (2) *Instruction Following*: improved ability to interpret and execute diverse natural language tasks; and (3) *Complex Problem-Solving*: outperforming PLMs on intricate tasks such as mathematical word problems.

Fine-tuning LLMs also differs from PLMs due to their massive parameter size, making local hosting resource-intensive and often impractical. Fine-tuning is typically done via APIs, where datasets of input-output pairs are provided, after which the model can be queried with specific instructions to generate the desired output.

ChatGPT. ChatGPT is a conversational AI built on the GPT framework and transformer architecture, optimized for dialogue. Trained on a vast and diverse corpus, it captures grammar, context, and cultural references to engage in natural conversations. It performs tasks ranging from question answering and code generation to creative writing. ChatGPT was trained using Reinforcement Learning from Human Feedback (RLHF) [46] to align outputs with human preferences. For our experiments, we fine-tuned ChatGPT 3.5 for 3 epochs, as OpenAI currently limits fine-tuning to 3.5, while ChatGPT 4.0 [47] (released May 13, 2024) is available only to certain subscription tiers.

LLaMA 3.1. LLaMA 3.1 is Meta's most advanced language model, trained on 15 trillion tokens from public and human-annotated data. Compared to LLaMA 3, it introduces key improvements, most notably a context window of 128,000 tokens (vs. 8,192), enabling much longer text processing. Unlike the closed-source ChatGPT, LLaMA 3.1 is open-source. It is available in 8B, 70B, and 405B parameters; given the resource demands of the 405B model and the moderate complexity of our task, we used the 70B model for a balance of performance and cost.

4 *Mujaz*: Design and Technical Details

We formulate our problem as a sequence-to-sequence (Seq2Seq) learning task [48], and *Mujaz* pursues the abstractive approach to solve this task. Given a CVE description, the goal of *Mujaz* is to summarize the description by extracting the relevant information for vulnerability analysis and presenting it in a standardized format as an output. In this section, we first outline the challenges in developing a model for summarization and normalization to present a unified format. Second, we present an overview of *Mujaz*'s pipeline along with the goals we aim to achieve.

4.1 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. Per §1, *Mujaz* aims to curate multiple features from the original CVE description. Utilizing such features for summarization requires 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 should support domain-specific language (i.e., security) by the appropriate encoding and decoding. For our summarization task, the model must constitute an encoder to represent the input and a decoder to produce the output. Unfortunately, vulnerability descriptions are limited in scope and content, typically including 1-3 lines of text.

Training a model for vulnerability summarization requires a massive dataset 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, pre-trained language models are already trained on massive textual data and provide an excellent alternative to bootstrapping our model. Fine-tuning such a pre-trained model is orders of magnitude simpler than training a model from scratch and works by adjusting the model's weights on a labeled dataset for the chosen task. Fine-tuning is convenient, fast, and demands much fewer resources than training a model from scratch. We employ pre-trained 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 that render their accuracy paramount. As such, we propose human metrics to quantify the accuracy, 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).

4.2 Pipeline: High-level Overview

Mujaz follows a conventional architecture of an encoder/decoder-based transformer, as shown in Figure 1. The pipeline consists of two independent phases illustrating our design. The first phase represents our parallel dataset curation, showing how a CVE description is broken into three features as in ❶. In this phase, *Mujaz* integrates several preprocessing techniques to ensure the high quality of the input text and the extracted features. The second phase 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 expects two sequences, representing an input of size n and a target 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

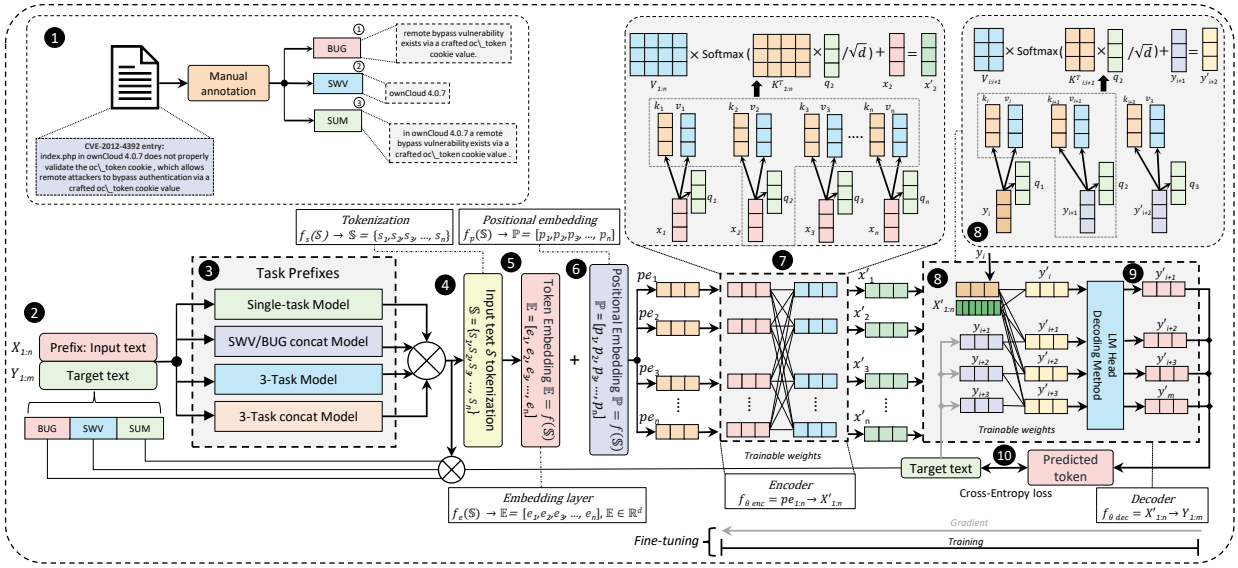

Fig. 1. Mujaz multi-task pipeline: (1) dataset curation to build the prospective dataset, and (2) task-specific training using CVE descriptions with extracted features for simultaneous multi-task learning.

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 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 with cross-entropy. The training is done using teacher forcing, indicated with the gray arrows, where the predicted token is used to compute the loss, and the correct token is fed back to the decoder for the next token prediction.

4.3 Pipeline: Technical Details

For convenience, we describe the internal architecture of the original Transformer presented in [31] within our pipeline.

Architecture Overview. The transformer is depicted in Figure 1 and consists of two components: an encoder and a decoder. The encoder transforms a sequence into a representation capturing the relationship between tokens, while the decoder utilizes this representation to perform the summarization task. Our pipeline in Figure 1 reflects a multi-task model with T5. 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 1, the annotation process starts by having a CVE description manually annotated to extract three features, the BUG, SWV, and SUM. The BUG feature in ① is a summarization and normalization of the description that identifies the type of bug and how it could be exploited. The key information consolidated here is the vulnerability type and a summarized version of the affected interfaces or functions. A CVE description of a library could include the library name (e.g., `myLibrary 4.1.2`) and affected headers (e.g., `myFunction` in `myHeader.h`). `myLibrary 4.1.2`, being the software with the version, would be included as the SWV feature. In contrast, `myFunction` in `myHeader.h`, being part of how the vulnerability is exploited, is part of the BUG feature. The SWV feature in ② is a list of vulnerable software and their versions present in the CVE description. This information could be spread throughout the description, and we consolidate it and make it semi-uniform. The relevant software is represented in this feature as a list with its affected versions followed by other affected software and versions.

The SUM feature in ③ is a grammatical concatenation of SWV and BUG 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 summary's beginning.

In 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, serving as a target text based on the appended prefix. In ❸, the prefix determines the task and target text and passes the input with the prefix to the next step, tokenization. The detail of each task is explained in section 4.4.

Tokenization. Our tokenizer in ❹ breaks the input into a

set of tokens. Most pre-trained 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 [49], while T5 uses SentencePiece [50]; 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 multiple sequences of the same length.

Embedding. The embedding layer in 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 [51], [52] or Glove [53]. In both cases, the embedding will improve and get updated according to the training dataset. The vector's dimensionality is a hyperparameter, typically set to 512.

Positional embeddings are essential in Transformer models because they provide information about the position of tokens in a sequence. Unlike recurrent neural networks that process data sequentially, Transformers process all tokens in parallel and lack inherent awareness of their order. The token embedding in step ❺ and positional embedding in step ❻ are added together to form the final embedding, which is passed to the encoder.

Encoder. The encoder in ❼ consists of two components: (1) multi-headed attention and (2) feed-forward network. The encoder is fed an embedded sequence $PE_{1:n}$ to produce 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:

$$y_i = \sum_j \text{softmax}({x_i}^T x_j) x_j; \tag{1}$$

i is the token's index where 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. 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 $K^T_{1:n}$, 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 a softmax layer, producing a probability distribution. A weighted sum of the value in v for each token in the sequence corresponding to the key vector is calculated, representing the new token embedding capturing attention with every other token.

The novel attention mechanism is expressed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{2}$$

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

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 next token generation, the token from the target is passed, also known as teacher forcing [54]. The decoder uses uni-directional self-attention, conditioning the new output vectors on the encoded sequence and any previously generated token, blocking tokens' visibility 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:

$$P(x_i|x_{1:i-1}) = \frac{exp(u_i)}{\sum_j exp(u_j)}. \tag{3}$$

The next token generation depends on the chosen generation method. 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 [48] 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 over the vocabulary and picks a word randomly from such a group, where K is a hyperparameter.

Top Nucleus. This sampling method [55] chooses a small set of tokens whose cumulative probability exceeds a predefined probability p, which is a hyperparameter; this is:

$$\sum_{x \in V^{(p)}} P(x_i | x_{1:i-1}) \geq p. \tag{4}$$

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

4.4 Multi-task Model

The pre-trained 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 for 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 1, is a multi-step process consisting of feature creation, tokenization, encoding, learning, and decoding to produce an abstractive summary task of the input. The three tasks are BUG, SWV, and SUM.

- The BUG task seeks to produce the description of the software vulnerability given a CVE description.
- The SWV task seeks to output the vulnerable software and versions, similar to the NER system of [7].
- 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 the method's effectiveness.

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 §4.1). However, in our evaluation (see §5) the model showed serious deficiencies, e.g., omitting part of the description or the software associated with it, necessitating the development of multi-task models, as we aim to force the model to attend to specific portions of the description.

SWV/BUG Concatenation Model. The first multi-task model was devised using a naive approach: the model is trained on both the SWV and BUG tasks such that it attends explicitly 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 the necessary information without regard to the overall summary.

This approach, however, may associate the two tasks as a result of a strong correlation in the output, particularly 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. Thus, we fine-tune the model on BART using two separate models for each task and concatenating the output of each model.

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

3-Task Concatenation Model. The 3-task concatenation model uses multi-task learning. However, rather than treating the BUG and SWV tasks as sub-tasks of the SUM, we inverse the order where the SUM task is used to support the BUG and SWV tasks training.

The 3-task concatenation model is trained on all three tasks. However, instead of using the output of the SUM task, the final output is the concatenation of the SWV task and the BUG task. The idea with this heuristic is that if the model outputs the information that it thinks is part of the SWV and BUG tasks, then all necessary information should be present. In essence, one can observe that the 3-Task Model is an effort to prioritize fluency while the 3-task concatenation model is an effort to prioritize completeness (see §5).

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.

5 Evaluation

5.1 Experimental Setup

We use the T5-base variation of the T5 model as the foundation for our models. The dataset is split into $1,204$ for training and 379 for testing. The training set was split with 10% to be used for validation. 100 samples were randomly selected from the testing set for evaluation on the human metrics (§5.3). All models were fine-tuned on our labeled dataset (§5.2) for 4 epochs with a batch size of 8 and a learning rate of 0.0001.

We set the repetition penalty to 2 to discourage the model from repeating already generated words. We set the length penalty to 2 to encourage the model to produce a longer summary. For the decoding method used to generate the next token, beam search was used as a decoding method to generate the next token with the number of beams set to 2. We also set early stopping to `True` to ensure all beam hypotheses have reached the end of the sequence.

For our experimental evaluation, we used Pytorch `v1.12` and Pytorch lightning `v1.9` as a framework to train both models. Preprocessing and splitting the dataset was accomplished using Sklearn `v1.0.2` and Pandas `v1.3` libraries. The models were deployed using the transformers library `v.4.10` provided by Hugging Face [56] to model

TABLE 2: Examples of simple task labels

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

BART and T5. We ran the experiment on a cloud GPU of type Tesla P100 with a memory of 16 GB.

5.2 Dataset and Data Curation

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

This manual process uses tokenized CVE descriptions from [7]. A traditional tokenization system like *moses* [57] could be used here, but since punctuation is 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 [7] 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 [7] 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, e.g., Exploit-DB or SecurityFocus, as many of those descriptions contain dozens of lines of code and require debugging, which is outside the scope of this work. We only consider CVE descriptions of 13 words or longer as shorter descriptions tend to be too succinct to summarize.

Upon the filtering steps, 15,209 entries are left, from which the dataset is manually created. From each entry, a parallel corpus of 3 features is created: summary, software/version, and bug (SUM, SWV, and BUG tasks in the model). The final size of the dataset was 1,583 entries, each with SUM, SWV, and BUG features. Examples of each feature on an input description are shown in Table 2.

Creating the features and choosing what information to include or omit is 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 a lack of vulnerability-type bias. The previous work by Dong *et al.* [7] 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.

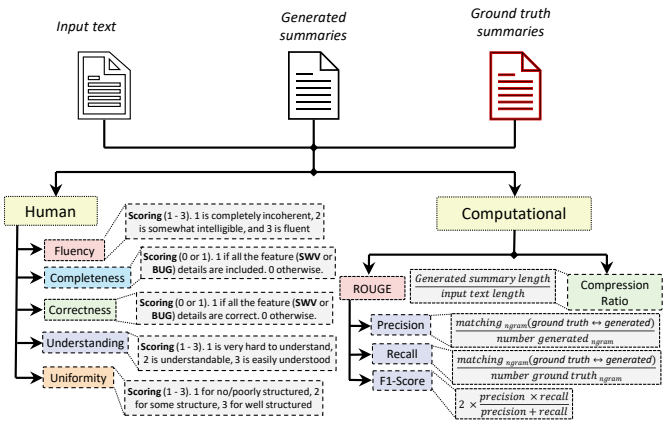

Fig. 2. Evaluation metrics.

5.3 Evaluation Metrics

Mujaz was evaluated using computational and human metrics. Computational metrics provide quantitative performance measures, enabling large-scale evaluation and future benchmarking. Human evaluation captures aspects difficult to assess automatically, such as fluency and ease of understanding. In this process, reviewers assign each summary a score for every metric, and final scores are averaged across samples. Figure 2 summarizes our evaluation metrics.

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

ROUGE [58]. ROUGE (Recall-Oriented Understudy for Gisting Evaluation) compares a generated summary with a reference using n-gram overlap and is widely correlated with human judgment. It reports recall (R), precision (P), and F1: recall is the fraction of matching n-grams over all reference n-grams, precision is the fraction over all generated n-grams, and F1 is their harmonic mean, F1 $= 2(PR)/(P+R)$.

Compression Ratio. This measures sentence reduction as output length normalized by input length. Lower values are better, as *Mujaz* should shorten input while maintaining strong human metrics.

Human Metrics. Computational metrics like ROUGE focus on n-gram overlap and fail to capture coherence, motivating additional measures of summary quality. We define five human metrics to assess accuracy, structure, and coherence, with binary or small-scale grading to reduce subjectivity.

Fluency. This score is rated 1–3: 1 for incoherent, 2 for somewhat intelligible, 3 for grammatically and semantically fluent. Since the system aims to produce clearer descriptions, incoherent outputs are unacceptable.

Completeness. This score consists of two binary sub-scores. *SWV-completeness*: 1 if all software and versions appear in the summary; 0 otherwise. *BUG-completeness*: 1 if all bug details are included; 0 otherwise.

Correctness. This score also consists of two binary sub-scores. *SWV-correctness*: 1 if all software and versions in the summary are correct; 0 otherwise. *BUG-correctness*: 1 if all included bug details are correct; 0 otherwise.

Understanding. This score is rated 1–3: 3 if meaning is very easy to grasp, 2 if somewhat difficult but possible, 1 if very hard to understand. Inputs are expected to score high in this metric to ensure *Mujaz* does not degrade quality.

TABLE 3: Comparison of fine-tuned T5 across recall (R), precision (P), F1, and compression ratio (CR).

Task	Model	R	P	F1	CR
SUM	T5	0.80	0.86	0.82	0.61
SUM	BART	0.83	0.82	0.82	0.63
SUM	ChatGPT	**0.89**	0.89	**0.88**	0.63
SUM	LLAMA 3.1	0.46	0.23	0.30	1.76
SWV/BUG Concat.	T5	0.78	0.91	0.84	**0.55**
SWV/BUG Concat.	BART	0.79	0.86	0.81	0.56
3-Task Concat.	T5	0.80	**0.92**	0.85	0.56
3-Task Model	T5	0.80	0.86	0.82	0.58

TABLE 4: 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	0.83	0.82	0.82	0.63
Top K-Sampling	0.82	0.81	0.81	0.63
Top Nucleus	0.83	0.81	0.81	0.64

Uniformity. Evaluates structural consistency across outputs. A score of 3 indicates high consistency, 2 moderate consistency with differences, and 1 little to no consistency.

5.4 Results

Computational Evaluation. We first report the score for all computational metrics in Table 3. 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.

Per Table 3, the 3-task concatenation model achieved the best score for recall, precision, and F1 compared to other T5 models. This is anticipated, considering that the model trained on the three tasks, allowing it 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.

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. The SWV/BUG concatenation model was devised using two models to learn how to attend to each feature.

We tested each model with different decoding methods to evaluate token generation. Beam search was the best method for BART with a slightly higher F1-score than Top-k sampling and Top nucleus sampling. In contrast, T5 achieved very good results with the Beam search method but a very low score with the other methods. Therefore, we only consider Beam search as our decoding method for all other models. The evaluation of different decoding methods for BART is shown in Table 4.

Human Evaluation. The goal of the human evaluation is to convey an 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 perform well for the BUG completeness metric, per Table 5. The BUG attribute includes more descriptive information about the bug, which may span over several sentences, while SWV is usually short and self-contained, explaining the high completeness.

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 SWV's completeness and correctness, all *Mujaz* scores remained above 0.83 except for BART on SUM. Compared to the more abstract summarization of BUG, SWV is more extractive of distinctive words, making it easier for the model to capture. Since it is extractive, it is unlikely an incorrect software name will be produced, resulting in high correctness scores. As seen in Table 5, 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.

6 Discussion

6.1 The BART Models

We conducted a thorough investigation of the 100 samples used for evaluation and discussed insights into each model. Although discussed individually, we note that some of the insights 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 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 overuses a technical concept, such as "denial of service".

TABLE 5: Comparison of T5 and BART across subjective metrics (fluency, completeness, correctness, understanding, uniformity) and Rouge. Concat. denotes concatenation.

Mode	Task	Fluency	Completeness		Correctness		Understanding	Uniformity	F1 score
			SWV	Bug	SWV	Bug			
T5	3-Task Concat.	2.24	0.90	0.66	0.94	0.92	2.63	2.48	0.85
	3-Task	2.82	0.89	0.57	0.97	0.89	2.81	2.84	0.82
	SUM Task	2.78	0.89	0.58	0.97	0.83	2.73	2.81	0.82
	SWV/Bug Concat.	2.30	0.89	0.57	0.96	0.87	2.73	2.46	0.84
BART	SUM Task	2.78	0.83	0.70	0.76	0.92	2.77	2.83	0.82
	SWV/Bug Concat.	2.81	0.90	0.69	0.83	0.91	2.84	2.85	0.81

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 it still makes a mistake in a digit after a dot separator or generates an incorrect BUG description for a CVE with the same SWV. Our training of two separate models explains these deficiencies.

6.2 The T5 Models

We highlight some of the observed issues in each model and contrast them. We first discuss the 3-task concatenation model and SWV/BUG concatenation model, as they both produce a summary by concatenating the prediction of SWV prefix and BUG prefix.

Concatenation Models. Both models did not produce punctuation, although the 3-Task Model was trained on the SUM task with punctuation. However, the models learned specific keywords, such as "by" and "via", allowing them 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 a few cases, and that the 3-task concatenation model confuses keywords that appear in context, e.g., producing "gaining privilege" instead of "gaining information" in one CVE. In contrast, the SWV/BUG concatenation model predicts the correct words, implying that this confusion may come from the SUM prefix and showing the impact of using multiple prefixes.

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

The SUM Task Model produced more abstractive summaries, often adding or substituting words, which sometimes affected bug correctness. In contrast, the 3-Task Model generated outputs closer to the target summary. Both models frequently replaced "execute arbitrary code" with "php code execution," despite *php* not appearing in the source, suggesting influence from pretraining or the SUM prefix used in training and prediction. Although the 3-Task Concat. Model was also trained with the SUM prefix, it rarely produced this phrase. Overall, both models generated correct, concise, and easy-to-understand summaries.

All models, except SWV/BUG Concat. correctly identified SWV even when hierarchically structured. A CVE entry may include multiple levels (organization, software suite, or sub-components), yet models followed the chain to the last preposition "in," marking the vulnerable software—an observation confirmed across multiple samples. Computational metrics, however, can be misleading for pre-trained models on our dataset. Although they showed higher F1 and compression, this is expected since SWV and BUG prefixes are shorter than SUM's, inflating both scores. These results align with human metrics, reinforcing our claim.

The concatenation models scored lower in fluency, understanding, and uniformity, indicating that training on the SUM prefix yields better and richer summaries. This also shows that compression ratio alone is not a reliable measure of summary quality.

6.3 Large Language Models

As discussed in section 3.2, LLMs have demonstrated remarkable capabilities across various language tasks. However, our task—generating uniform and consistent summaries—is relatively uncommon. Given the extensive training data of LLMs, this task could either be challenging or straightforward for these models. We fine-tuned both models with identical parameters and used the same query (instruction) to generate a normalized and consistent summary, focusing specifically on the SUM task (Task-1).

Per Table 3, ChatGPT 3.5 outperformed all models in terms of recall and F1 scores, though its compression ratio was not as efficient as that of T5, which achieved better ratios. On the other hand, LLAMA 3.1 scored lowest in computational metrics. The difference between the two models is their ability to follow specific instructions. ChatGPT generated summaries closely aligned with the fine-tuning dataset, and LLAMA 3.1 produced longer, more structured outputs with additional details beyond the dataset's scope, such as affected system, privileges required, and severity. While this structured output provides a constructive way to understand the vulnerability and its impact, it did not meet the primary objective of this experiment, which was to generate a normalized and consistent description.

As model size increases, the dataset must be sufficiently large to influence the model's output. However, this conclusion cannot be applied to ChatGPT since its number of parameters is not publicly known. ChatGPT's improved performance could also be attributed to various proprietary techniques used during its training and alignment. It is worth noting that, despite these results, LLAMA 3.1 outperforms ChatGPT 3.5 in standard NLP benchmarks. Samples of the outputs are provided in the appendix (Table 12).

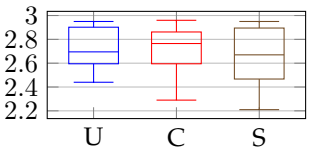

Fig. 3. User study results with 100 samples generated by BART using the USE dataset with five evaluators. The evaluation is conducted across three metrics: understanding, coherence, and simplicity.

7 User Study

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

Understanding follows the criteria and scale in 5.3. Coherence measures the logical connection of ideas within and across sentences on a 1–3 scale: 3 for fully coherent and smooth flow, 2 for partial coherence with minor disruptions, and 1 for incoherent text lacking structure. Simplicity quantifies whether the normalized description makes the content easier for a security analyst to grasp. Also graded 1–3, a score of 3 indicates the normalized description is simpler, 2 means both are equally clear, and 1 implies the normalized version is more complex.

We recruited 12 participants, all with at least a BS in computer science, including 9 Ph.D. students in computer security focused on operations and threat intelligence. Each participant evaluated 100 generated normalized descriptions and their original counterparts, assigning scores for each metric based on predefined definitions and scales. The average scores per metric are reported in Figure 3.

For understanding, scores ranged from 2.44 to 2.95, with 75% above the lower quartile of 2.595 and a median of 2.695, showing that *Mujaz* produces easy-to-understand descriptions. Coherence varied more (2.29–2.96), though its range (2.60–2.86) and median (2.77) align with understanding, suggesting correlation. This variability likely stems from participants' differing English proficiency. Simplicity showed the widest spread (2.21–2.95), with a range of 2.467–2.895 and a median of 2.67; still, 75% of scores $>$2.467 indicate that *Mujaz* generally yields simpler descriptions.

8 Conclusion

We presented *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 a consistent and uniform structure and is shown effective in learning multiple features, measured by both computational and (our newly defined and justified) human metrics. Our results showed that attending to different aspects from the description is possible using *Mujaz*'s architecture.

References

[1] 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," *Computers & Security*, vol. 60, pp. 154–176, 2016.
[2] 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*, vol. 7, no. 23, 2020.
[3] MITRE, "Common vulnerabilities and exposures (cve)," Online, 2022. [Online]. Available: https://cve.mitre.org/
[4] 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 *ICSE*, 2020, pp. 1547–1559.
[5] NIST, "National vulnerability database (nvd)," Online, 2022. [Online]. Available: https://nvd.nist.gov/
[6] ——, "National institute of standards and technology," Online, 2022. [Online]. Available: https://www.nist.gov/
[7] 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*, 2019, pp. 869–885.
[8] P. Nespoli, D. Papamartzivanos, F. G. Mármol, and G. Kambourakis, "Optimal countermeasures selection against cyber attacks: A comprehensive survey on reaction frameworks," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1361–1396, 2018.
[9] A. Anwar, A. Abusnaina, S. Chen, F. Li, and D. Mohaisen, "Cleaning the nvd: Comprehensive quality assessment, improvements, and analyses," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, pp. 4255–4269, 2022.
[10] P. Johnson, R. Lagerström, M. Ekstedt, and U. Franke, "Can the common vulnerability scoring system be trusted? a bayesian analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 1002–1015, 2016.
[11] L. K. Shar, L. C. Briand, and H. B. K. Tan, "Web application vulnerability prediction using hybrid program analysis and machine learning," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 6, pp. 688–707, 2014.
[12] H. Holm, M. Ekstedt, and D. Andersson, "Empirical analysis of system-level vulnerability metrics through actual attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 6, pp. 825–837, 2012.
[13] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "Large-scale characterization of software vulnerability life cycles," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 4, pp. 730–744, 2019.
[14] B. Zhao, S. Ji, W.-H. Lee, C. Lin, H. Weng, J. Wu, P. Zhou, L. Fang, and R. Beyah, "A large-scale empirical study on the vulnerability of deployed iot devices," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1826–1840, 2020.
[15] H. Althebeiti and D. Mohaisen, "Enriching vulnerability reports through automated and augmented description summarization," in *International Conference on Information Security Applications (WISA)*, ser. LNCS, vol. 13009. Springer, 2022, pp. 265–277.
[16] P. Kuehn, M. Bayer, M. Wendelborn, and C. Reuter, "Ovana: An approach to analyze and improve the information quality of vulnerability databases," in *ARES*, 2021, pp. 22:1–22:11.
[17] Microsoft, "Microsoft security response center," Online, 2022. [Online]. Available: https://cwe.mitre.org/
[18] IBM, "X-force exchange," Online, 2022. [Online]. Available: https://exchange.xforce.ibmcloud.com/
[19] K. Kanakogi, H. Washizaki, Y. Fukazawa, S. Ogata, T. Okubo *et al.*, "Tracing capec attack patterns from cve vulnerability information using natural language processing technique," in *HICSS*, 2021.
[20] D. Gonzalez, H. Hastings, and M. Mirakhorli, "Automated characterization of software vulnerabilities," in *IEEE ICSME*, 2019, pp. 135–139.
[21] E. Wåreus and M. Hell, "Automated cpe labeling of cve summaries with machine learning," in *DIMVA*, 2020, pp. 3–22.
[22] V. H. Nguyen and F. Massacci, "The (un)reliability of nvd vulnerable versions data: an empirical experiment on google chrome vulnerabilities," in *ACM AsiaCCS*, 2013, pp. 493–498.
[23] 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 COMPSAC*, 2021, pp. 1020–1025.
[24] Accenture Security, "Bugtraq," Online, 2021. [Online]. Available: https://bugtraq.securityfocus.com/

[25] A. Niakanlahiji, J. Wei, and B.-T. Chu, "A natural language processing based trend analysis of advanced persistent threat techniques," in *IEEE BigData*, 2018, pp. 2995–3000.
[26] 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*, 2019, pp. 887–903.
[27] S. Mumtaz, C. Rodríguez, B. Benatallah, M. Al-Banna, and S. Zamanirad, "Learning word representation for the cyber security vulnerability domain," in *IJCNN*, 2020, pp. 1–8.
[28] S. Yitagesu, X. Zhang, Z. Feng, X. Li, and Z. Xing, "Automatic part-of-speech tagging for security vulnerability descriptions," in *MSR*, 2021, pp. 29–40.
[29] K. Kanakogi, H. Washizaki, Y. Fukazawa, S. Ogata, T. Okubo *et al.*, "Tracing cve vulnerability information to capec attack patterns using natural language processing techniques," *Information*, vol. 12, no. 8, p. 298, 2021.
[30] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," OpenAI, 2018.
[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NeurIPS*, 2017, pp. 5998–6008.
[32] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019, pp. 4171–4186.
[33] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," in *ICLR*, 2020.
[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 *ACL*, 2020, pp. 7871–7880.
[35] 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*, 2019.
[36] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI, 2019.
[37] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan *et al.*, "Language models are few-shot learners," in *NeurIPS*, 2020.
[38] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NeurIPS*, 2012, pp. 1106–1114.
[39] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015.
[40] 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 CVPR*, 2015, pp. 1–9.
[41] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," in *ACL*, 2018, pp. 328–339.
[42] 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," *Journal of Machine Learning Research*, vol. 21, pp. 140:1–140:67, 2020.
[43] OpenAI, "Chatgpt," Online, 2023. [Online]. Available: https://openai.com/blog/chatgpt
[44] R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu *et al.*, "Gemini: A family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.
[45] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.12345*, 2024.
[46] P. F. Christiano, J. Leike, T. B. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," in *NeurIPS*, 2017, pp. 4299–4307.
[47] OpenAI, "Chatgpt 4o," Online, 2024. [Online]. Available: https://openai.com/index/hello-gpt-4o/
[48] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *NeurIPS*, 2014, pp. 3104–3112.
[49] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *ACL*, 2016.
[50] T. Kudo, "Subword regularization: Improving neural network translation models with multiple subword candidates," in *ACL*, 2018, pp. 66–75.
[51] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *ICLR*, 2013.
[52] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS*, 2013, pp. 3111–3119.
[53] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *EMNLP*, 2014, pp. 1532–1543.
[54] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989.
[55] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," in *ICLR*, 2020.
[56] Hugging Face, "Hugging face," Online, 2022. [Online]. Available: https://huggingface.co/
[57] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico *et al.*, "Moses: Open source toolkit for statistical machine translation," in *ACL*, 2007.
[58] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*, 2004, pp. 74–81.
[59] S. Ozkan, "Cve details," Online, 2022. [Online]. Available: https://www.cvedetails.com/
[60] Y. Huang, T. Lv, L. Cui, Y. Lu, and F. Wei, "Layoutlmv3: Pre-training for document ai with unified text and image masking," in *ACM MM*, 2022, pp. 4083–4091.

Hattan Althebeiti is an Assistant Professor at Taif University. He received the Ph.D. degree in Computer Engineering from the University of Central Florida in 2023. His research interests include machine learning, natural language processing, vulnerability analysis, and generative AI. He has published in venues such as WISA, WISE, and ACM CCS.

William Chen received the B.S. degree in Computer Science from the University of Central Florida in 2021 and is currently pursuing the Ph.D. degree at Carnegie Mellon University. His research focuses on spoken language processing, speech recognition, speech translation, and machine translation. He has co-authored publications in INTERSPEECH, ICASSP, ASRU, EMNLP, and SLT.

Brett Fazio received the B.S. degree in Computer Science from the University of Central Florida in 2021. He is a Software Engineer at Two Sigma Securities. His research background includes natural language processing, machine translation, vulnerability description summarization, and parallel search algorithms, with work appearing in LoResMT and ACM CCS.

Jeman Park received the Ph.D. degree in Computer Science from the University of Central Florida in 2020. He was a Postdoctoral Researcher at the Georgia Institute of Technology and is currently an Assistant Professor at Kyung Hee University. His research interests include computer security, malware analysis, and adversarial learning. He has published widely in top venues such as USENIX Security, ACM CCS, and NDSS.

David Mohaisen is a Professor of Computer Science at the University of Central Florida, where he directs the Security and Analytics Lab. He received the Ph.D. degree in Computer Science from the University of Minnesota in 2012. His research interests include computer security, privacy, systems, and machine learning, with applications to malware analysis, software vulnerabilities, IoT, and large-scale measurement. He has published extensively in premier venues including USENIX Security, NDSS, CCS, and IEEE/ACM Transactions on Networking.

APPENDIX

Hyperparameter Tuning. We conducted extensive experimentation to learn the best set of hyperparameters for fine-tuning. The tested hyperparameters were *repetition penalty*, *length penalty*, and *number of beams* for beam search. We also tested the two other decoding methods, top-k sampling and nucleus sampling. The experiments were conducted across batch sizes of 4, 8, and 16. We chose the SUM task because it is the simplest, using a single prefix for training and prediction, and report patterns. For BART, we increased the number of beams and the length penalty, which improved the recall and F1 score but decreased the precision. A possible explanation is that as we increase the number of beams, the model has more options to choose from, thus deviating from the reference. We also found the repetition penalty has a direct effect on the recall, although it does not carry the same effect on T5.

Our results were consistent across different batch sizes, although the highest F1 score of 0.81 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 0.83. Because some hyperparameters encourage a longer summary, the compression ratio decreased as we increased the batch size, as shown in A.

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 $p = 0.95$ for nucleus sampling. The repetition and length penalties were set to 2 based on our initial results. The recall score was consistent with k set to 5 for the top-k sampling, producing the best score across all batch sizes. However, this was not the case for precision, which fluctuated with different top-k values and batch sizes.

This unpredictability had an effect on the F1 score, which fluctuated with the best score, achieving 0.82 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 0.59 with k set to 10, but it did not outperform the beam search. A batch size of 16 achieved the highest score for recall, precision, and F1 score.

Nucleus sampling considers a small subset of tokens from the top-k tokens where the cumulative probability exceeds a predefined threshold per Eq. (4). 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 were 0.81 and 0.82, with k equal 10, while the best precision was 0.83 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.

The following results highlight the effect of hyperparameter tuning and its impact on summarization. The experiments were conducted on the SUM-Task due to its simplicity and abstract nature, considering a single feature that includes most details of a CVE entry. Table 6 shows the best hyperparameters for computational metrics on both models.

ROUGE metrics achieved the best score with the batch size of $B = 4$ and the number of beams $b = 5$. However, the compression ratio reached its peak with different settings, yet it did not reach the best compression in BART. We note that the repetition penalty affected recall, scoring 0.81 with $r = 2$ and 0.81 with $r = 1$, implying that its effect is minimal or none. In practice, a trained model cannot produce the same results due to the stochastic nature of training. Therefore, with this slim margin between $r = 2$ and $r = 1$, it is infeasible to determine if the repetition penalty affected the score. Thus, we consider the best hyperparameters for T5 to be $b = 5$, $r = 1$, $l = 4$, and $B = 4$.

In contrast, BART showed more fluctuating results. Considering the ROUGE metrics, we exclude batch size because it is consistent across them. The precision with repetition penalty $r = 2$ achieved 0.83, close to the reported score. On the other hand, the F1 and recall scores with $r = 1$ were 0.79 and 0.77, respectively, indicating the precision influence on other metrics. The perfect combination of ROUGE metrics aims to find the optimal score between recall and precision as they form the basis for computing the F1 score. In contrast, recall achieved 0.81 with the following settings $b = 2$, $r = 2$, and $l = 2$, which is almost the best score. Therefore, we consider the best hyperparameters for T5 to be $b = 2$, $r = 2$, $l = 2$, and $B = 4$.

The batch size impacts each model as a larger size requires more resources, which could be costly, especially for T5. The compression ratio hyperparameters were consistent for both models. However, that does not guarantee the summary is conclusive. Unifying the compression ratio hyperparameters to be consistent with others will produce a compression ratio of 0.58 for BART and 0.60 for T5.

TABLE 6: Best hyperparameters for T5 and BART in terms of recall, precision, F1 score, and compression ratio on the SUM task. Parameters: b=beams, r=repetition penalty, l=length penalty, B=batch size.

Model	Metric	Score	b	r	l	B
T5	Recall	0.81	5	2	4	4
	Precision	0.8642	5	1	4	4
	F1 score	0.83	5	1	4	4
	Compression	0.5786	2	1	2	8
BART	Recall	0.81	5	2	4	16
	Precision	0.83	2	1	2	16
	F1 score	0.81	2	2	2	16
	Compression	0.53	2	1	2	8

Table 7 shows the best results using Top-k sampling (the top part) and Top-k with nucleus sampling (the bottom part) as decoding methods. Top-k uses a parameter k, defining the number of tokens to consider as the next token for text generation. The k tokens are selected according to the probability distribution over the entire vocabulary. We fix the other hyperparameters based on our findings in Table 6. We observe that precision and F1 score have the same parameters for the best score. However, recall reached 0.81 for the same set of parameters. The margin is yet negligible as the highest recall score is 0.81. In contrast, considering the recall parameters $k = 5$, $r = 2$, and $l = 2$, the precision

and F1 score reached 0.83 and 0.82, which is also very close. This shows that the model produces its best results when $k = 5$ or $k = 20$.

Top-k with nucleus sampling has very similar results but with $k = 10$. The precision score deviates from the other metrics, with $k = 20$ achieving 0.83. However, with $k = 10$, the precision score reaches 0.83. We conclude that the best k value for Top-k with nucleus sampling is $k = 10$. The compression ratio showed similar patterns as the batch size was inconsistent with the parameters.

TABLE 7: Best hyperparameters for BART on the SUM task, evaluated by recall, precision, F1, and compression ratio. Parameters: k=top-k, r=repetition penalty, l=length penalty, B=batch size. Top block: top-k decoding; bottom block: nucleus sampling ($p = 0.95$).

Decoding	Metric	Score	k	r	l	B
Top-k	Recall	0.81	5	2	2	16
	Precision	0.8355	20	2	2	16
	F1 score	0.82	20	2	2	16
	Compression	0.60	10	2	2	4
Nucleus	Recall	0.81	10	2	2	16
	Precision	0.83	20	2	2	16
	F1 score	0.82	10	2	2	16
	Compression	0.59	5	2	2	8

Table 8 shows the result of an additional experiment where we use to duplicate the training dataset and flip the content of the two columns so that the input text will be the SUM feature and the target is the CVE entry. We emphasize that the test set has not been duplicated or modified. Our intuition is that this could benefit the model as the input text and the target content could be interchangeable. T5 exhibited improvement in the ROUGE metrics compared to the same set of parameters on the SUM Task without augmentation. However, this improvement could be attributed to having more data for training. The compression ratio achieved better results under this setting.

On the other hand, BART exhibited slight yet consistent degradation across all ROUGE metrics. However, The compression ratio has significantly improved, encouraging further investigation and evaluation.

TABLE 8: Augmented dataset training. Both models were fine-tuned using beam search. Metrics: recall (R), precision (P), F1 score, and compression ratio (CR) on the SUM task.

Model	R	P	F1	CR	b	r	l	B
T5	0.80	0.86	0.83	0.59	5	1	4	8
T5	0.80	0.87	0.83	0.54	5	1	4	4
BART	0.79	0.82	0.80	0.46	2	2	2	16
BART	0.80	0.82	0.81	0.53	2	2	2	8
BART	0.81	0.78	0.79	0.50	2	2	2	4

Dataset and Vulnerability Distribution. The sampled dataset in Table 9 is not exactly balanced because some vulnerabilities are more common than others, according to [59]. Therefore, we fluctuate the number of instances for a few types to project them toward their existence in reality. Moreover, this will ensure the robustness of the trained model against all types of vulnerabilities, even if they do not appear frequently. We believe the increased numbers do not bias the dataset toward any particular type.

TABLE 9: Dataset distribution by vulnerability type and number of instances.

Type	# Instances
DoS	192
Code Execution	100
Overflow	100
Memory Corruption	100
SQL Injection	100
XSS	100
Directory Traversal	98
HTTP Response Splitting	148
Bypass	100
Gain Information	169
Gain Privileges	170
CSRF	105
File Inclusion	100

We envision that *Mujaz*'s flexibility and parallel nature could be deployed to any textual data that includes separable and well-defined features. For example, *Mujaz* could accommodate financial reports or hospital records if it is possible to curate a dataset from these documents. For example, a medical record typically includes patient history (diseases or surgeries), medications, special conditions, and possibly a summary of each visit. Each part of the record resembles a feature that can be attended to independently or jointly with other features. The number of features can increase as long as it is feasible to derive them consistently from the record/report, either manually or through a predefined framework capable of separating them. *Mujaz* paves the way to approach summarization through a parallel corpus, making developing a separate pipeline for feature extraction critical. Recent work [60] showed the possibility of implementing a multi-modal transformer to extract textual data from forms/documents through image recognition. Such work complements and empowers *Mujaz* to perform multi-task in various domains.

Mujaz Demonstrations. The following tables show examples generated by *Mujaz*, showcasing its effectiveness, how the human evaluators graded each metric, and the LLM-generated summaries. The main objective of *Mujaz* is to generate consistent and uniform vulnerability descriptions, regardless of the original structure. These demonstrations highlight how *Mujaz* transforms complex and varied input descriptions into a standardized and easy-to-understand format using our curated dataset.

Sample Outputs. Sample outputs and original descriptions for Mujaz are in Table 10, for ChatGPT in Table 11, and for LLaMA 3.1 in Table 12. Success and failure cases with reasoning of Mujaz are shown in Table 13.

TABLE 10: Mujaz Evaluation Examples

Original	Prediction
heap-based buffer overflow in w in sun solaris 8 through 10, and opensolaris before snv_124, allows local users to gain privileges via unspecified vectors.	sun solaris 8 through 10 gaining privileges is possible via unspecified vectors for the w in heap-based buffer overflow. **(3-Task Concatenation model)**
	in sun solaris 8 through 10, and opensolaris before snv_124, gaining privileges is possible via unspecified vectors for the w in the heap-based buffer overflow. **(3-Task model)**
	in sun sun solaris 8 through 10, and opensolaris before snv_124, gaining privileges is possible via unspecified vectors for the w in heap-based buffer overflow in w in sun-based buffer overflow. **(SUM-Task model)**
	sun solaris 8-10 10 opensolaris before snv_124 gaining privileges is possible via unspecified vectors for w in w in w in w in w. **(SWV/BUG Concatenation model)**
cross-site request forgery (csrf) vulnerability in the cucreports page in cisco unity connection 11.0 (0.98000.225) and 11.0 (0.98000.332) allows remote attackers to hijack the authentication of arbitrary users, aka bug id cscut33659.	in cisco unity connection 11.0 (0.98000.225) and 11.0 (0.98000.332) a csrf vulnerability exists allowing remote attackers to hijack the cucreports page in the cucsrf vulnerability allows remote attackers to hijack the authentication of arbitrary users. **(SUM Task model)**
memory corruption in gemalto hasp srm, sentinel hasp and sentinel ldk products prior to sentinel ldk rte version 7.55 might cause remote code execution.	in sentinel hasp and sentinel ldk products prior to sentinel ldk rte version 7.55, executing arbitrary commands is possible via a srm, sentinel hasp and sentinel ldk products prior to sentinel ldk rte version 7.55. **(3-Task model)**
multiple cross-site scripting (xss) vulnerabilities in oauth_callback.php on accellion kiteworks appliances before kw2016.03.00 allow remote attackers to inject arbitrary web script or html via the (1) code, (2) error, or (3) error_description parameter.	accellion kiteworks appliances before kw2016.03.00 xss is possible via the code, error, error_description parameter. **(SWV/BUG Concatenation model)**
cross-site scripting (xss) vulnerability in search.php in myre business directory allows remote attackers to inject arbitrary web script or html via the look parameter.	myre business directory xss is possible via the look parameter of search.php. **(3-Task Concatenation model)**

TABLE 11: ChatGPT Demonstrations

Original	Prediction
unspecified vulnerability in the dynamic tracing framework (dtrace) in sun solaris 10 allows local users with priv_dtrace_user or priv_dtrace_proc privileges to obtain sensitive kernel information via unspecified vectors, a different vulnerability than cve-2007-4126.	in sun solaris 10, gaining information is possible via unspecified vectors for the dynamic tracing framework (dtrace).
an issue was discovered in certain apple products. ios before 11.1 is affected. safari before 11.0.1 is affected. icloud before 7.1 on windows is affected. itunes before 12.7.1 on windows is affected. tvos before 11.1 is affected. the issue involves the ”webkit” component. it allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption and application crash) via a crafted web site.	in ios before 11.1, safari before 11.0.1, icloud before 7.1 for windows, itunes before 12.7.1 for windows, tvos before 11.1, memory corruption is possible in the webkit component via a crafted web site.
pioctls in openafs 1.6.x before 1.6.13 allows local users to read kernel memory via crafted commands.	in openafs 1.6.x before 1.6.13, gaining information is possible via crafted commands for the pioctls.

TABLE 12: LLAMA 3.1 Demonstrations

Original	Prediction
unspecified vulnerability in the dynamic tracing framework (dtrace) in sun solaris 10 allows local users with priv_dtrace_user or priv_dtrace_proc privileges to obtain sensitive kernel information via unspecified vectors, a different vulnerability than cve-2007-4126.	(Here is a normalized and consistent summary: **Vulnerability:** Unspecified vulnerability in Dynamic Tracing Framework (DTrace) in Sun Solaris 10 **Affected System:** Sun Solaris 10 **Privileges Required:** Local users with priv_dtrace_user or priv_dtrace_proc privileges **Impact:** Allows local users to obtain sensitive kernel information **Vector:** Unspecified vectors **CVE:** Not specified (different from CVE-2007-4126))
an issue was discovered in certain apple products. ios before 11.1 is affected. safari before 11.0.1 is affected. icloud before 7.1 on windows is affected. itunes before 12.7.1 on windows is affected. tvos before 11.1 is affected. the issue involves the ”webkit” component. it allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption and application crash) via a crafted web site.	Here is a normalized and consistent summary of the issue: **Vulnerability Summary** * **Affected Products:** Apple iOS (before 11.1), Safari (before 11.0.1), iCloud (before 7.1 on Windows), iTunes (before 12.7.1 on Windows), and tvOS (before 11.1) * **Component:** WebKit * **Vulnerability Type:** Remote Code Execution and Denial of Service (Memory Corruption and Application Crash) * **Attack Vector:** Visiting a crafted website * **Impact:** Allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption and application crash)
pioctls in openafs 1.6.x before 1.6.13 allows local users to read kernel memory via crafted commands.	Here is a normalized and consistent summary: **Vulnerability:** PIOCTL in OpenAFS 1.6.x (before 1.6.13) **Impact:** Local users can read kernel memory **Attack Vector:** Crafted commands **Affected Version:** OpenAFS 1.6.x (before 1.6.13) **Severity:** Medium to High Note: PIOCTL is a control function in OpenAFS that allows users to perform various operations. The vulnerability allows local users to craft malicious commands that can read kernel memory, potentially leading to information disclosure or other security issues.

TABLE 13: Mujaz Success and Failure Cases with reasons

Original	Prediction
buffer overflow in libpng 1.2.34 allows remote attackers to cause a denial of service via a crafted PNG file.	in libpng 1.2.34 a buffer overflow allows remote attackers to cause a denial of service via a crafted PNG file. **(3-Task model)**
	Reason: succeeds because SWV and BUG are explicit and short, matching common training patterns.
cross-site scripting (xss) in admin.php in webapp-cms 3.1 and 3.2 allows remote attackers to inject script via the q parameter.	webapp-cms 3.1 and 3.2 xss is possible via the q parameter in admin.php. **(SWV/BUG Concatenation model)**
	Reason: succeeds because multiple versions are simple to enumerate, and BUG tokens are concise.
vulnerability in app-suite 1.0.0, 1.0.1, 1.0.12, 1.1 and 2.0-beta; improper input validation in parser.c may allow remote code execution.	in app-suite 1.0.0, 1.0.1, 1.1 and 2.0-beta remote code execution is possible via input validation in parser.c. **(3-Task model)**
	Reason: fails by omitting version 1.0.12, showing Mujaz struggles with long/irregular version lists.
multiple heap overflows triggered by malformed headers in the custom protocol handler of netproxy prior to 5.0; exploitation requires authentication and may result in privilege escalation via race condition combined with session reuse.	in netproxy prior to 5.0 heap overflow in protocol handler allows remote attackers to execute arbitrary code. **(SUM Task model)**
	Reason: fails by dropping conditions (authentication required) and overgeneralizing consequence (privilege escalation → arbitrary code execution).