

## Article

# VULREM: Fine-Tuned BERT-Based Source-Code Potential Vulnerability Scanning System to Mitigate Attacks in Web Applications

Remzi Gürfidan 

Computer Programming Department, Yalvaç Technical Sciences Vocational School, Isparta University of Applied Science, Isparta 32200, Turkey; remzigurfidan@isparta.edu.tr

**Abstract:** Software vulnerabilities in web applications are one of the sensitive points in data and application security. Although closing a vulnerability after it is detected in web applications seems to be a solution, detecting vulnerabilities in the source code before the vulnerability is detected effectively prevents malicious attacks. In this paper, we present an improved and automated Bidirectional Encoder Representations from Transformers (BERT)-based approach to detect vulnerabilities in web applications developed in C-Sharp. For the training and testing of the proposed VULREM (Vulnerability Remzi) model, a dataset of eight different CVE (Common Vulnerabilities and Exposures)-numbered critical vulnerabilities was created from the source code of six different applications specific to the study. In the VULREM model, fine-tuning was performed within the BERT model to obtain maximum accuracy from the dataset. To obtain the optimum performance according to the number of source-code lines, six different input lengths were tested with different batch sizes. Classification metrics were used for the testing and performance evaluation of the model, and an average F1-score of 99% was obtained for the best sequence length according to eight different vulnerability classifications. In line with the findings obtained, this will play an important role in both vulnerability detection in web applications of the C-Sharp language and in detecting and correcting critical vulnerabilities in the developmental processes of web applications, with an accuracy of 99%.

**Keywords:** fine-tuned BERT; source-code vulnerability; detection system



**Citation:** Gürfidan, R. VULREM: Fine-Tuned BERT-Based Source-Code Potential Vulnerability Scanning System to Mitigate Attacks in Web Applications. *Appl. Sci.* **2024**, *14*, 9697. <https://doi.org/10.3390/app14219697>

Academic Editor: Christos Bouras

Received: 26 September 2024

Revised: 17 October 2024

Accepted: 22 October 2024

Published: 23 October 2024



**Copyright:** © 2024 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Today, all commercial and individual applications have turned into web-based applications due to the ease of remote access and updating. Many applications developed as desktop applications have also started to support web-based solutions. Backend and frontend processes are involved in the transformation of web-based applications into products, and these stages are usually developed by different developers. The backends of web applications are developed with a programming language and the frontends are developed with a framework. A web application has at least a three-layered architecture and is the result of project management [1].

Vulnerability refers to situations where a system or software is vulnerable or lacking in security. Vulnerabilities are weaknesses that malicious actors or attackers can use to take over the system, access data, cause service interruptions or cause other damage [2]. Software-based vulnerabilities are situations that allow attackers to exploit the system due to errors or deficiencies in the code or design of the software [3]. Web-based software comprises applications that can be accessed and used over the internet. The web-based use of many applications brings potential risks.

In web applications, risks and threats are usually uncovered during testing phases after the project is completed [4]. Such software can be vulnerable to various security risks. Especially in web-based software systems, SQL injection, XSS (cross-site scripting), CSRF (cross-site request forgery), sensitive data exposure, XML parsing vulnerabilities, insecure

file uploads, insecure authentication and authorization, insecure connection and communication and insecure routing and communication are common vulnerability points [5–7]. Penetration tests are commonly preferred to test vulnerabilities. Penetration tests reveal the existing vulnerabilities of the web application [8].

Common Vulnerabilities and Exposures (CVE) represents processes to identify, describe and catalog cyber vulnerabilities that have been identified and publicly disclosed. CVE is a vulnerability database in which each vulnerability contains a unique identifier code along with a description of the vulnerability and other useful information [9]. In the CVE database, vulnerabilities are shared on a scale of 1–10 to determine the severity of each vulnerability. The scaling of vulnerabilities in the CVE database between 1 and 10 is called the Common Vulnerability Scoring System (CVSS) [10]. CVSS assigns scores out of 10 to vulnerabilities using certain algorithms and metrics. Vulnerabilities are then grouped according to their scores. These groups are formed according to the importance of the vulnerabilities they contain. CVSS then groups vulnerabilities into two different versions, CVSS v2.0 and CVSS v3.0 [11].

Vulnerability detection systems need to be continuously expanded and improved in order to instantly discover new threats. There are many important criteria to assess the challenges of detection to protect the security of applications and systems as the scale, time, technical and computing requirements of applications increase. This emphasizes the need to quickly and clearly identify vulnerabilities in complex systems. In addition, it can be difficult to strike a balance between true positives and false positives when optimizing coverage. Increasing the sensitivity of scanning tools both increases measurements and overwhelms security teams with alerts [12]. Reducing the sensitivity and power has the opposite effect, giving the application a false sense of security. Finally, attackers have started to exploit multiple vulnerabilities in a system in parallel or in a chain. These are much harder to detect, and a few low-risk vulnerabilities alone may not trigger an alarm on any system. All these issues and requirements reveal the many advantages of evaluating vulnerability detection at the source-code stage. In line with this, we propose VULREM, a BERT-based vulnerability scanning tool that performs vulnerability scans of the backend source code of web applications. The contributions and innovations of the VULREM model are as follows:

- VULREM is a C#-based tool for the backend of web applications that can perform static source-code analyses based on eight different vulnerabilities.
- The original study-specific dataset was used to train and test the VULREM model.
- VULREM uses a fine-tuned BERT model on a pre-trained BERT.
- According to performance evaluations, VULREM achieved a high accuracy in vulnerability extraction, with an F1-score of 96% and a Matthew's correlation coefficient (MCC) of 83%.

In the following sections of the study, we first review the literature on source-code-based vulnerability detection and evaluate its originality. Then, the development process of the VULREM model is explained step-by-step and the model is tested. The findings are compared and discussed with the existing literature and the performance is evaluated. In the last step, the results and future work of the VULREM model are briefly explained.

## 2. Related Works

In this section of the study, approaches that detect vulnerabilities through source code are examined. In source-code vulnerability analyses, language-based learning algorithms are generally focused on.

Xiaomeng et al. utilized deep learning methods to analyze source code based on a code feature graph. The study used the Software Assurance Reference Dataset (SARD), a publicly available dataset of C/C++ instruction injections. The f-measure, precision, false-positive rate, true-positive rate and false-negative rate of the proposed deep learning code property graph-based vulnerability analysis (CPGVA) method were calculated [13]. Sahin and Abualigah used a real-world SA dataset based on three open-source PHP applications

to detect vulnerabilities in open-source software programs and to detect malware. A new deep-learning-based vulnerability detection model was proposed to identify features [14]. Jeon and Kim proposed a deep-learning-based automatic vulnerability analysis system (AutoVAS) that effectively represented source code as embedding vectors using datasets from various projects in the National Vulnerability Database (NVD) and Software Assurance Reference Database (SARD) [15].

In their study, Hegedus et al. examined how machine learning techniques performed when predicting potentially vulnerable functions in JavaScript programs. For this research, they applied eight machine learning algorithms to build prediction models using a new dataset generated from vulnerability information in the public databases of the Node Security Project and the Snyk platform, as well as code fix patches on GitHub. They used static source-code metrics as predictors and an exhaustive grid-search algorithm to find the best performing models. The KNN machine learning algorithm showed the most successful results [16]. Zhang presented a machine learning classifier designed to detect SQL injection vulnerabilities in PHP code. Both classical and deep-learning-based machine learning algorithms were used to train and evaluate classifier models using input validation and sanitization features extracted from source-code files. In a ten-fold cross-validation, a model trained using a convolutional neural network (CNN) achieved the highest precision (95.4%), while a multilayer perceptron (MLP)-based model achieved the highest recall (63.7%) and the highest f-measure (0.746) [17].

Kim et al. aimed to detect software vulnerabilities using a BERT-based model. As a method, vulnerabilities were classified more accurately by enabling the model to understand the syntactic and semantic aspects of the code. According to the numerical results obtained, the model performed with an F1-score of over 95% and high accuracy values. It was also reported that the model gave lower false-positive and -negative results compared with traditional methods. This reveals that BERT is an effective method to detect software vulnerabilities [18]. Huang et al. focused on the detection of software vulnerabilities. The BBVD approach aims to automatically detect vulnerabilities using BERT-based models, treating high-level programming languages as natural languages. This approach provides more efficient results when traditional methods cannot effectively scan large amounts of source code. Experimental results showed that BBVD performed well in tests using SARD and Big-Vul datasets [19]. Quan et al. introduced XGV-BERT, a deep-learning-based software vulnerability detection framework. XGV-BERT detected software vulnerabilities by combining a pre-trained CodeBERT model and graph neural network (GCN). The model learned the contextual relationships of source-code attributes by exploiting large datasets and using transfer learning. The results showed that XGV-BERT provided a higher accuracy than existing methods such as VulDeePecker and SySeVR, with an F1-score of 97.5% for VulDeePecker and 95.5% for SySeVR [20]. Zhu et al. aimed to detect types of software vulnerabilities. Using a BERT-based model and four different slicing methods, vulnerabilities in software code were classified in detail. The study represented code slices with four slicing methods, including API calls, arithmetic expressions, pointer usage and array usage. According to the experimental results, the BERT-based model outperformed other methods such as VulDeePecker and SySeVR, with a higher accuracy and F1-score (93.3%) [21].

C and C++ languages are widely used in the literature. First of all, the fact that today's applications are web-based and the majority of data vulnerabilities are not experienced in web-based applications motivated our study to scan for web-based vulnerabilities. In addition, as there is no web-based application development with C and C++ languages, we did not focus on these languages. When the literature studies are examined, it can be seen that vulnerability detection is more common for open-source languages and applications. We focused on languages that are shared on open-source code-sharing platforms such as GitHub and that have critical vulnerabilities. With this motivation, in our study, we first focused on the C-Sharp language, which is one of the most widely used languages in web applications. The vulnerabilities used in the literature have received CVE numbers, but it can be seen that these were detected in a very old time period. Currently, there is a

possibility that these vulnerabilities have been fixed or removed due to version updates. Therefore, in this study, we focused on current vulnerabilities in particular. The reason we chose this language is that the applications developed at our university and which constitute the dataset of the study use C-Sharp and it is widely used in the backend of web applications. In particular, the fact that the number of critical vulnerabilities experienced in the scripting languages on the frontend side is low and because data vulnerabilities can be corrected in the backend side limited the study only to the backend.

### 3. Vulnerability Detection Methods

A vulnerability is a term, also known as a weakness, that refers to a situation in which the protection methods and techniques of a system or application are weak or incomplete. These vulnerabilities can allow malicious actors to gain unauthorized access to the system or unintended use of the service. Security problems can be caused by many different factors, including the following:

- Software bugs;
- Design flaws;
- Weak encryption methods;
- Poor configuration;
- Lack of updates and patches;
- User errors;
- Social engineering;
- Factors such as malware infiltration, which can contribute to the emergence of vulnerabilities [22].

Likewise, vulnerabilities can be detected by different methods such as vulnerability scans, penetration tests, source-code analyses, log analyses, social engineering tests and audits. All these methods are basically characterized as vulnerability analyses. To detect security problems, the following three basic features must be present in the scanning process [23]:

- Discovery: The process of providing a snapshot or continuous view of the assets within the target system and application.
- Assessment: The detection of anomalies or known vulnerabilities registered in the CVE database in the target system and application.
- Prioritization: The prioritization of assets within the target system and application using CVE data, data science or threat intelligence to resolve vulnerability results.

Almost all applications developed recently provide web-based services. Finding vulnerabilities in web applications is a difficult and comprehensive process. Modern web applications are created with a wide network of third-party frameworks, services and APIs. All of this scope expansion can lead to the discovery of new vulnerabilities or exacerbate existing vulnerabilities, making vulnerability detection even more difficult. At the same time, the development of comprehensive web applications using a development process with many developers has popularized Agile and DevOps approaches [24]. As a result, rapid version releases and enhancements have increased, and unintended security vulnerabilities have also emerged through these approaches.

#### 3.1. Manual Application Security Testing (MAST)

This is known as the processes completed by the evaluation of security experts instead of automatic scanning software in the processes of detecting vulnerabilities in applications and systems. The expert who conducts the test finds the vulnerabilities and determines an importance assessment. At the same time, they also give advice on solving the vulnerabilities detected. In MAST methods, the knowledge and experience of the expert is very effective, especially in the context of business/process logic errors, access control issues and other complex security threats. In some cases, the process detects attack and vulnerability routes that automated scanning tools overlook [25]. MAST is effective in penetration testing, threat modeling, source-code reviews and security-oriented design

reviews. However, due to the increasing complexity of modern web applications and the high speed of software development, MAST methods can be difficult to implement.

### 3.2. Static Application Security Testing (SAST)

SAST is a method that enables the automatic detection of vulnerabilities at a very early stage of the software development lifecycle (SDLC) because it does not require an application to run and occurs without code execution. It is an important process because the vulnerabilities detected during the development phase do not have any critical consequences and there is an opportunity to fix them [26]. SAST processes use state-of-the-art analysis methods such as data flow analysis, control flow analysis and flaw analysis to detect vulnerabilities. Initially, SAST methods focused on input validation and memory overflow vulnerabilities. More recently, SAST has been based on development language and platform-dependent parsers, decoders and syntax tree generators, limiting the types of projects they can be used for [27].

### 3.3. Dynamic Application Security Testing (DAST)

Unlike SAST methods, DAST methods evaluate behaviors and program responses with different usage scenarios to detect potential vulnerabilities without providing access to the source code. Thus, DAST methods help to identify vulnerabilities arising from the application runtime environment. The first-use cases of DAST were mostly for online applications such as SQL injection, cross-site scripting and weak authentication [28]. With the inclusion of modern software in comprehensive development processes such as APIs and mobile applications, DAST methods have expanded to include this development. At the same time, firewalls, intrusion detection, configuration errors, interactions of application components and access control problems are among the tools to find risks and threats that can be overlooked in static analysis. Recently developed DAST tools are supported by artificial intelligence models to reduce false positives and anomalies suggestive of vulnerabilities or attacks.

### 3.4. Interactive Application Security Testing (IAST)

The sophisticated coverage network of modern applications can reveal disadvantages in both SAST and DAST methods. Building on the advantages of SAST and DAST methods, IAST techniques provide a hybrid method that analyzes applications during both the test phase and runtime [29]. By integrating a static code analysis with a dynamic runtime analysis, IAST methods minimize false positives and false negatives compared with SAST and DAST. With current AI-supported IAST methods, IAST tools are included in processes such as integrated development environment (IDE), build systems and continuous integration and continuous delivery (continuous integration (CI)/continuous development (CD)) to understand the processes of software and produce effective results [8].

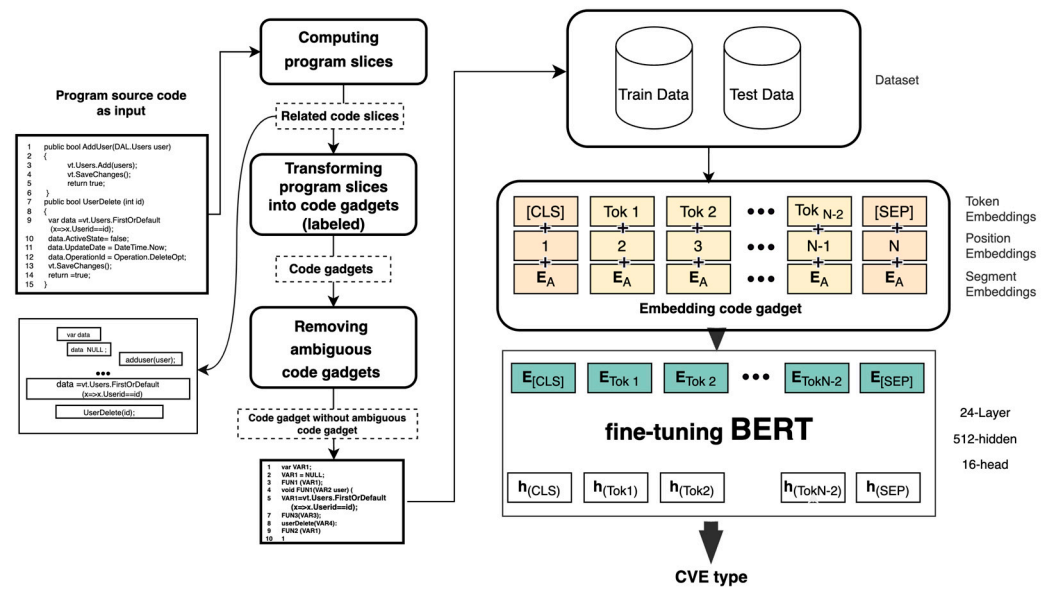
## 4. Proposed Methodology

Introduced in Section 3, vulnerability detection methods such as SAST, DAST and IAST are traditional vulnerability detection strategies used in web applications. These methods are often enriched with manual and dynamic analyses but, given the complexity of modern web applications, these processes are time-consuming and error-prone. The VULREM model was developed to address the shortcomings of these methods and improve security by detecting vulnerabilities at the source-code stage. Our model combined the advantages of early-stage detection methods such as static code analysis (SAST) with a deep-learning-based solution. Thus, the difficulties and error rates encountered in manual processes were significantly reduced. In particular, a direct inspection of the source code allowed for the proactive detection of vulnerabilities that may have been missed in dynamic analyses.

In the proposed model, the improved BERT model was used to perform vulnerability scanning over the source code of web applications and to detect identified CVE-coded vulnerabilities. The BERT method is a new language representation model that uses a bidirectional



transducer network to pre-train the language model on a corpus and fine-tune the pre-trained model for other tasks [30]. The problem-specific BERT design can be sequentially represented as a single line of code or a block of code. The input representation is constructed by collecting token, segment and position code fragment embeddings corresponding with a given code [31]. At the same time, vulnerability prediction in the BERT model is bidirectional, both left-to-right and right-to-left. The development architecture of the proposed model, named VULREM, is shown in Figure 1. In the following sections of the paper, the development and performance evaluation of the VULREM model are presented.



**Figure 1.** Architectural representation of the proposed VULREM model.

#### 4.1. Dataset

A study-specific dataset was created during the development process of the VUMREM model. The C-Sharp programming language is used when developing web applications with ASP.NET and ASP.NET Core. Web applications developed with C-Sharp language are rarely shared as open-source. Due to these limitations, the source codes of the applications developed at our university's IT department were labeled according to the vulnerability descriptions in the Common Vulnerability and Exposures (CVE) and National Vulnerability Database (NVD) archives. If there was a vulnerable function in the source code, we noted in which file and software it was located and labeled it as vulnerable, and we labeled the code blocks that did not have any vulnerabilities as non-vulnerable. Table 1 shows the distribution of the vulnerable and non-vulnerable data we obtained from the preferred web applications. In the development of the VULREM model, the source code of six different C-Sharp-based web applications were labeled according to CVE vulnerabilities. In Table 1, the number of lines of source code examined in the application is "Samples", the number of lines of source code with a vulnerability is "Vuln Code" and the ratio of the total number of lines is "Vulnerable Rate".

In the obtained dataset, there were 2,141,783 lines of source-code samples, and 17,128 lines of source code had CVE-numbered vulnerabilities. There were eight different types of vulnerabilities in the vulnerability scanning of the VULREM model, and the number of samples in the dataset for these vulnerabilities is given in Table 2.

**Table 1.** Distribution of the dataset obtained from the source code of applications developed with ASP.NET and ASP.NET Core according to the sample, vulnerable and non-vulnerable code blocks.

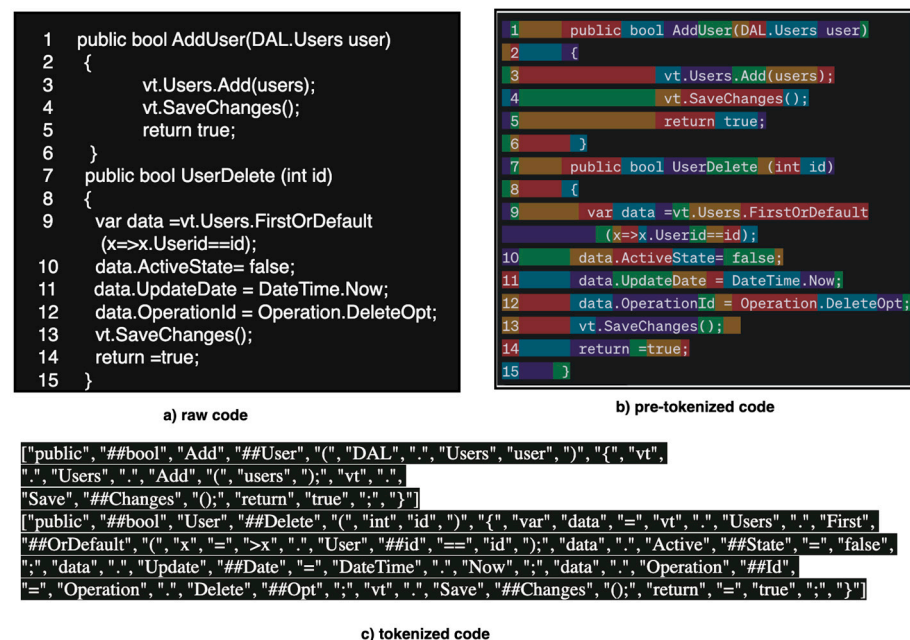
Dataset	Access	Samples	Vuln Code	Vulnerable Rate (%)
Document Management System	Private	540,223	2539	0.469
Student Information System	Private	1,131,201	5345	0.472
Personal Information System	Private	389,870	1320	3.385
Quality Management System	Private	123,892	1672	1.349
nopCommerce [32]	Public	675,120	4450	0.659
Miniblog Core [33]	Public	412,678	1802	0.436
$\Sigma$ Sum		2,141,783	17,128	

**Table 2.** Distribution and description of vulnerabilities according to CVE codes.

CVE Code	Description	Samples
CVE-2023-36899	ASP.NET Elevation of Privilege Vulnerability	3420
CVE-2023-36560	ASP.NET Security Feature Bypass Vulnerability	1253
CVE-2023-36558	ASP.NET Core Security Feature Bypass Vulnerability	4610
CVE-2023-36038	ASP.NET Core Denial of Service Vulnerability	570
CVE-2023-35391	ASP.NET Core Signal R and Visual Studio Information Disclosure Vulnerability	3190
CVE-2023-33170	ASP.NET and Visual Studio Security Feature Bypass Vulnerability	230
CVE-2015-6099	Cross-Site Scripting (XSS) Vulnerability in ASP.NET	3500
CVE-2014-4075	Cross-Site Scripting (XSS) Vulnerability in System	588

#### 4.2. Token Embeddings Phase

The BERT model of the proposed approach used a structure called WordPiece to partition the input source-code blocks into word tokens [34]. The WordPiece structure divides each input source code into subwords called tokens. As the midpoint between words and characters, word particles retain their linguistic meaning. Thus, even with a small-sized vocabulary, it can be successful in the case of out-of-vocabulary word fragments (Figure 2).

**Figure 2.** The tokenizer process of raw C-Sharp code.

In the first process for source-code vulnerability detection in the proposed approach, code slices were extracted from the source-code files using the tokenizer training phase. Code fragments are candidates for syntax- and semantics-based vulnerabilities in particular. The open-source code-analysis tool joern was used to parse the code slices and source-code files and to extract the control flow graph corresponding with each code. In addition,

user-defined variable and function names were mapped to symbolic names by preserving the original name of the keywords in the source-code language.

#### 4.3. Transformer

In the proposed approach, the transformer architectures used for the BERT model were a natural language processing (NLP) structure that included an attention mechanism and an encoder–decoder structure that weighed the effect of different parts of the input source code on the sentence [35]. The encoder–decoder structure can be understood by explaining the working structures of the attention mechanisms [36]. The transformer architecture included input and output encoders, attention mechanisms, feed-forward networks and the SoftMax function for optimal probability computation. The tokenization and word representation part was performed in the input embedding part, which was the first processing element of the encoder. Here, a vector matrix was obtained after adding the encoded positions of the numeric word representations to the generated numeric word representations. This matrix array was processed in an attention mechanism to determine the relationships between the words in the sentence and to determine the probability of them being related. In this way, the BERT model performed contextual language detection tasks.

Attention mechanisms determine the level of association between input tokens by treating the currently processed token as a query, all previously queried tokens as a key and tokens in the sentence as vector matrices with a named value, as given in Equation (1) [37,38].

$$Attention \in \{Q, K, V\} = \sigma_{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) \quad (1)$$

The value  $d_k$  represents the number of dimensions of the key matrix. The Q matrix is multiplied by the transpose of the K matrix. A new vector matrix is obtained by dividing this matrix product by  $\sqrt{d_k}$ . The multiplication of this vector matrix by the matrix V with  $\gamma_{softmax}$  indicates the association rate in the group of tokens for each Q value [39]. Here, the function  $\sigma_{softmax}$  represents a probabilistic function indicating the level of association of the input values.  $\sigma_{softmax}$  produces a value between 0 and 1, no matter whether the input elements are positive or negative. The values given by  $\sigma_{softmax}$  are the probability score of the association.

$$\sigma \left( \frac{\rightarrow}{Z} \right)_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \quad (2)$$

Here  $\frac{\rightarrow}{Z}$  represents the input vector of  $\sigma_{softmax}$ ,  $z_i$  represents the  $i$ -indexed element of the input vector  $z_i$  and  $e^{z_i}$  represents the exponential function applied to each input.  $\sum_{j=1}^K e^{z_j}$  ensures that all inputs are reduced to 0–1 and the output is 1.

AutoModel and AutoTokenizer models from Python libraries were added to the proposed model approach to implement the above transformer structure.

#### 4.4. Model Implementation

As the BERT model in the proposed VULREM approach could undertake more than one task, the task type needed to be specified as a classification. During training and testing, the first token ([CLS]) of each input sequence was fixed as a special classification label. The output layer of the transformer ( $C \in R^H$ ) then used it as a sequence representation to perform the classification. Here, H is the hidden size.

When fine-tuning the BERT algorithm in the VULREM model,  $W \in R^{K \times H}$  was added, where K is the number of three CVE-coded vulnerabilities detected in the model. The exit probabilities for each K class were calculated, as in Equation (3).

$$P = \frac{e^{c.w^T}}{\sum_K C.w^T} \quad (3)$$



where  $P \in R^K$  represents the probabilities of the classification labels. However, in the BERT model, the pre-trained parameters, the uncased model and the parameters in the classification were jointly fine-tuned to maximize the probability value corresponding with the correct vulnerability. For the training of parameters, the Adam optimizer algorithm, which is also proposed by Google, is preferred [40]. The Adam algorithm has the ability to compute adaptive learning values for each input parameter [41]. It is also known as a combination of an Adam optimizer, RMSprop and a traditional stochastic gradient descent with momentum [42]. The Adam optimizer used in the BERT model in the proposed approach updated the parameter values according to Equations (4) and (5).

$$r_t = \beta_1 r_{t-1} + (1 - \beta_1) g_t \quad (4)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \quad (5)$$

where  $r_t$  and  $v_t$  are the estimate and variance, respectively, and the decay rate is  $\beta_1 - \beta_2$ . The errors in the  $r_t$  and  $v_t$  moments were corrected according to Equation (6).

$$\hat{r}_t = \frac{r_t}{1 - \beta_1^t} \quad (6)$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad (7)$$

where  $\hat{r}_t$  and  $\hat{v}_t$  are the corrected versions of  $r_t$  and  $v_t$ , respectively. All these values were used to obtain the value of  $W$ , as shown in Equation (8).

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{r}_t \quad (8)$$

where  $\eta$  represents the learning rate and  $\epsilon$  represents smoothing.

## 5. Result and Discussion

Computational studies to test and explore the performance of REMBERT, a vulnerability scanning tool in proposed web applications, were performed using a computing system with NVIDIA RTX 4080 GPU, 64 GB system RAM and INTEL 16 processor cores. During the development process of REMBERT, the python programming language was used and pytorch and BERT libraries were preferred.

### 5.1. Fine-Tuned BERT

A fine-tuned BERT approach was proposed for the VULREM model developed for vulnerability scanning in web applications. The VULREM model was based on 24 transformer layers, 16 attention heads and 512 hidden layers. The fine-tuned hyperparameters used in the training of the VULREM model are given in Table 3. In the training of the VULREM model, a batch size of 32 was used for 64, 128, 256 and 320 sequence lengths. In order to avoid memory problems, the batch size was set to 16 and 8 for 384 and 512 sequence lengths, respectively.

**Table 3.** Hyperparameters to train VULREM.

	Hyperparameter	Value
VULREM	Sequence length	64-128-256-320-384-512
	Batch size	32-32-32-32-16-8
	Epochs	5
	Learning rate	0.001
	Epsilon	0.008

### 5.2. Bag-of-Words-Based Approaches

In order to evaluate the performance of the proposed fine-tuned BERT model, it was trained with the following three text classifiers: k-NN, naïve Bayes (NB) and support vector machine (SVM). Textual features extracted from the source code using the bag-of-words model and term frequency–inverse document frequency (TF-IDF) were used to train k-NN, NB and SVM.

The creation of the unigrams required to train the models went through the steps of case conversion, symbolization, filter stop words and stemming, respectively. The preprocessing steps resulted in hyperparameters that were optimized using grid searches to train the models (Table 4).

**Table 4.** Hyperparameters to train bag-of-words-based models.

Model	Hyperparameter	Value
NB	smoothing	0.5
k-NN	neighbors dm (distance measure)	cosine similarity 12
SVM	svm type kernel type gamma nu cache size $\epsilon$ (epsilon) weight	nu-SVC rbf (radial basis function) 0.1 0.5 100 0.001 1

### 5.3. Evaluation Metrics

Performance metrics were used to evaluate the output performance of the model. Confusion matrix, accuracy, recall, precision and F1-score metrics were preferred to evaluate the performance of the proposed model. The purpose of using the preferred metrics in the model is briefly explained below.

**Accuracy:** This is a metric that produced a percentage of how many correct results the proposed model produced for the whole dataset (Equation (8)).

**Recall:** This permitted the determination of how many of the outputs that the proposed model should have positively predicted were correctly predicted (Equation (10)).

**Precision:** This was used to measure the performance of the accuracy of the proposed model's positively correct predictions (Equation (11)).

**F1-score:** Although precision and accuracy metrics were used for the performance of the proposed model, it was also verified using this metric, which gave a more precise result and was a combination of other metrics (Equation (12)). The F1-score provided more sensitive evaluations for an accurate evaluation, especially if the distribution of some vulnerability clusters in the dataset was not equal.

**Matthew's correlation coefficient (MCC):** This produced a value between  $-1$  and  $1$  to evaluate the performance of the model in multiple vulnerability classifications within the VUMREM model. The closer this value was to  $1$ , the higher the interpreted classification success (Equation (13)).

$$\gamma_{accuracy} = \frac{TP + TN}{TP + FN + FP + TN} \quad (9)$$

$$\gamma_{Precision} = \frac{TP}{TP + FP} \quad (10)$$

$$\gamma_{recall} = \frac{TP}{TP + FN} \quad (11)$$

$$\gamma_{f1score} = 2 \times \frac{\gamma_{Precision} \times \gamma_{recall}}{\gamma_{Precision} + \gamma_{recall}} \quad (12)$$

$$MCC_{multiclass} = \frac{c \cdot s - \sum_k^k p_k \cdot t_k}{\sqrt{(s^2 - \sum_k^k p_k^2)(s^2 - \sum_k^k t_k^2)}} \quad (13)$$

The explanation of the abbreviations of the metrics used in Equations (1)–(5) are as follows:

- True Positive (TP): When data were correctly predicted to be vulnerable or non-vulnerable.
- True Negative (TN): The prediction that a non-vulnerable code block was not vulnerable.
- False Positive (FP): The prediction that a non-vulnerable code block was vulnerable.
- False Negative (FN): A prediction state that indicated a vulnerable code block to be not vulnerable.
- k: The number of classes in the model labelled as vulnerable and non-vulnerable.
- $x = \sum_k^K X_{KK}$ : The number of correct predictions of those labelled as vulnerable and not vulnerable.
- $s = \sum_i^K \sum_j^K X_{ij}$ : The total dataset was the number.
- $p_k = \sum_i^K X_{ki}$ : The number of predictions of k classes in the dataset.
- $t_k = \sum_i^K X_{ik}$ : The number of correct recognitions of class k in the dataset.

#### 5.4. VULREM Performance Assessment

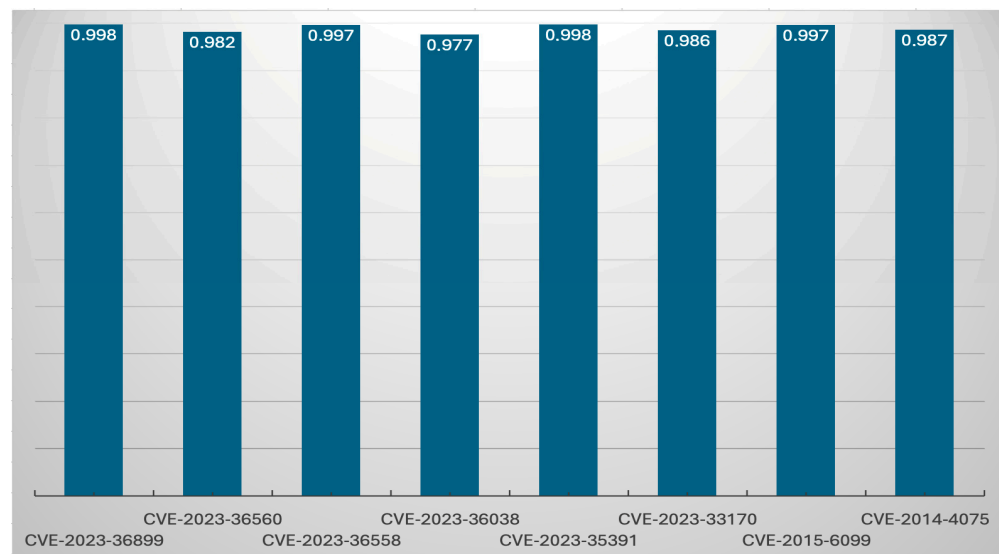
The vulnerability classification accuracies of the proposed model and bag-of-words-based models were compared, as shown in Table 5. All models of the proposed VULREM with different lengths showed higher vulnerability classification performance than the bag-of-words models. At the same time, the VULREM classification model was presented with inputs of different string lengths according to the number of source-code lines, and the highest accuracy performance was achieved at a string size of 320.

**Table 5.** Evaluation performance of models.

Model	Precision	Recall	Accuracy	F1-Score	MCC
k-NN	0.796	0.798	0.783	0.790	0.530
NB	0.810	0.828	0.840	0.823	0.590
SVM	0.8723	0.870	0.880	0.872	0.660
RoBERTa	0.9112	0.911	0.925	0.911	0.712
BiLSTM	0.9265	0.922	0.931	0.926	0.730
VULREM-64	0.940	0.945	0.951	0.952	0.780
VULREM-128	0.957	0.954	0.961	0.954	0.850
VULREM-256	0.970	0.968	0.972	0.973	0.861
VULREM-320	0.982	0.983	0.982	0.990	0.870
VULREM-384	0.960	0.968	0.962	0.963	0.840
VULREM-512	0.947	0.945	0.951	0.946	0.820

In the dataset obtained from six different web applications specific to the study, there was a total of eight different CVE-numbered vulnerabilities. The distribution of the sample numbers of these vulnerabilities was not homogeneous. Therefore, in the training and testing phase of the model, it was seen that there was a change in the F1-score values in direct proportion to the number of samples, as given in Figure 3, but the accuracy changes were very small.

In the test phase of the VULREM source-code scanning tool, scanning time performance was evaluated along with accuracy performance. According to the results given in Table 6, the model had a detection time that increased in direct proportion to the number of vulnerabilities detected in the source code. In the test phase of the dataset obtained from six different web applications, a maximum of 20 vulnerabilities were detected and the total detection time of these vulnerabilities was measured to be 12.4 s maximum. In terms of continuous vulnerability testing in software development processes, this time is acceptable.



**Figure 3.** F1-score accuracies of the model according to eight different CVE vulnerability classifications.

**Table 6.** Detection times by vuln count.

Vuln Count	Detection Time (s)
1	2.1
3	3.3
5	5.8
10	7.3
20	12.4

## 6. Conclusions and Future Work

The development and intensive use of many web-based applications brings data security risks. Many different languages and frameworks are used in the development process of web applications. There are many alternative ways to detect vulnerabilities in these web applications. Many security tests are performed in live or test environments after the application development process and if there are possible risks, serious time is required for correction. Therefore, web applications should be developed according to the clean code philosophy and vulnerabilities should be evaluated at the source-code stage. In line with this goal, the proposed VULREM approach aimed to detect eight different types of vulnerabilities common in ASP.NET and ASP.NET-Core-based web applications developed using the C-Sharp language at the source-code stage. The VUMREM vulnerability tool was based on the BERT model and performed vulnerability scanning using a dataset specifically created for the study. The VUMREM model could use 64, 128, 256, 320, 384 and 512 sequence lengths, according to the source code to be scanned for vulnerability. The main model performed the CVE vulnerability-type classification by going through the token, position and segment embedding stages in a BERT model improved according to a dataset specific to the study. The VUMREM model was evaluated using different metrics in the training and testing phases, and showed the highest performance among similar language-based learning algorithms, with an average accuracy of 96%. The source-code vulnerability detection times varied directly proportional to the number of lines of source code and the number of vulnerabilities detected. As a result of the tests, VUMREM classified the vulnerabilities in the dataset within 12 s at the longest, showing that it performed the classification in an acceptable time.

The vulnerability detection performance of the VULREM model offers significant advantages compared with traditional methods. For example, while SAST methods usually focus on specific vulnerabilities such as input validation and memory overflow, the VULREM model covers a wide spectrum of vulnerabilities (e.g., CVE-based vulnerabilities)

and successfully detects even those that may be missed by a manual analysis. Likewise, although DAST methods try to detect vulnerabilities at runtime, our VULREM model solved vulnerabilities at the source using a static code analysis, thus minimizing the risks that the application might encounter later. In this context, the 99% F1-score provided by our model showed a significant improvement in terms of both accuracy and speed compared with traditional methods.

In VULREM's future work, we plan to expand the classification of automated scanning tools by continuing to collect source codes and the vulnerability tagging of open-source or developed web applications. At the same time, the original dataset will be published on platforms such as GitHub and Kaggle to contribute to research.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References

1. Kaluža, M.; Kalanj, M.; Vukelić, B. A comparison of back-end frameworks for web application development. *Zb. Veleučilista U Rijeci* **2019**, *7*, 317–332. [\[CrossRef\]](#)
2. Amankwah, R.; Kudjo, P.K.; Antwi, S.Y. Evaluation of software vulnerability detection methods and tools: A review. *Int. J. Comput. Appl.* **2017**, *169*, 22–27. [\[CrossRef\]](#)
3. Chernis, B.; Verma, R. Machine learning methods for software vulnerability detection. In Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics, Tempe, AZ, USA, 21 March 2018; pp. 31–39.
4. Hu, L.; Chang, J.; Chen, Z.; Hou, B. Web application vulnerability detection method based on machine learning. *J. Phys. Conf. Ser.* **2021**, *1827*, 012061. [\[CrossRef\]](#)
5. Rafique, S.; Humayun, M.; Gul, Z.; Abbas, A.; Javed, H. Systematic Review of Web Application Security Vulnerabilities Detection Methods. *J. Comput. Commun.* **2015**, *03*, 28–40. [\[CrossRef\]](#)
6. Alazmi, S.; De Leon, D.C. A systematic literature review on the characteristics and effectiveness of web application vulnerability scanners. *IEEE Access* **2022**, *10*, 33200–33219. [\[CrossRef\]](#)
7. Bhor, R.V.; Khanuja, H.K. Analysis of web application security mechanism and Attack Detection using Vulnerability injection technique. In Proceedings of the 2016 International Conference on Computing Communication Control and automation (ICCUBEA), Pune, India, 12–13 August 2016; pp. 1–6.
8. Altulaihan, E.A.; Alismail, A.; Frikha, M. A Survey on Web Application Penetration Testing. *Electronics* **2023**, *12*, 1229. [\[CrossRef\]](#)
9. Babalau, I.; Corlatescu, D.; Grigorescu, O.; Sandescu, C.; Dascalu, M. Severity prediction of software vulnerabilities based on their text description. In Proceedings of the 2021 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNAS), Timisoara, Romania, 7–10 December 2021; pp. 171–177.
10. Figueroa-Lorenzo, S.; Añorga, J.; Arrizabalaga, S. A survey of IIoT protocols: A measure of vulnerability risk analysis based on CVSS. *ACM Comput. Surv. (CSUR)* **2020**, *53*, 1–53. [\[CrossRef\]](#)
11. Joh, H.; Malaiya, Y.K. Defining and assessing quantitative security risk measures using vulnerability lifecycle and cvss metrics. In Proceedings of the 2011 International Conference on Security and Management (SAM'11), Las Vegas, NV, USA, 18–21 July 2011; pp. 10–16.
12. Tudosi, A.D.; Graur, A.; Balan, D.G.; Potorac, A.D. Research on Security Weakness Using Penetration Testing in a Distributed Firewall. *Sensors* **2023**, *23*, 2683. [\[CrossRef\]](#)
13. Wang, X.; Zhang, T.; Wu, R.; Xin, W.; Hou, C. CPGVA: Code Property Graph based Vulnerability Analysis by Deep Learning. In Proceedings of the 2018 10th International Conference on Advanced Infocomm Technology (ICAIT), Stockholm, Sweden, 12–15 August 2018; pp. 184–188.
14. Şahin, C.B.; Abualigah, L. A novel deep learning-based feature selection model for improving the static analysis of vulnerability detection. *Neural Comput. Appl.* **2021**, *33*, 14049–14067. [\[CrossRef\]](#)
15. Jeon, S.; Kim, H.K. AutoVAS: An automated vulnerability analysis system with a deep learning approach. *Comput. Secur.* **2021**, *106*, 102308. [\[CrossRef\]](#)
16. Ferenc, R.; Hegedus, P.; Gyimesi, P.; Antal, G.; Ban, D.; Gyimothy, T. Challenging Machine Learning Algorithms in Predicting Vulnerable JavaScript Functions. In Proceedings of the 2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), Montreal, QC, Canada, 28–28 May 2019; pp. 8–14.



17. Zhang, K. A machine learning based approach to identify SQL injection vulnerabilities. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 1286–1288.
18. Kim, S.; Choi, J.; Ahmed, M.E.; Nepal, S.; Kim, H. Vuldebert: A vulnerability detection system using bert. In Proceedings of the 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Charlotte, NC, USA, 31 October–3 November 2022; pp. 69–74.
19. Huang, W.; Lin, S.; Chen, L. Bbvd: A bert-based method for vulnerability detection. *Int. J. Adv. Comput. Sci. Appl.* **2022**, *13*, 890–898. [\[CrossRef\]](#)
20. Quan, V.L.A.; Phat, C.T.; Van Nguyen, K.; Duy, P.T.; Pham, V.H. Xgv-bert: Leveraging contextualized language model and graph neural network for efficient software vulnerability detection. *arXiv* **2023**, arXiv:2309.14677.
21. Zhu, C.; Du, G.; Wu, T.; Cui, N.; Chen, L.; Shi, G. Bert-based vulnerability type identification with effective program representation. In *International Conference on Wireless Algorithms, Systems, and Applications*; Springer Nature Switzerland: Cham, Switzerland, 2022; pp. 271–282.
22. Chu, H.; Zhang, P.; Dong, H.; Xiao, Y.; Ji, S.; Li, W. A survey on smart contract vulnerabilities: Data sources, detection and repair. *Inf. Softw. Technol.* **2023**, *159*, 107221. [\[CrossRef\]](#)
23. Nie, X.; Li, N.; Wang, K.; Wang, S.; Luo, X.; Wang, H. Understanding and Tackling Label Errors in Deep Learning-Based Vulnerability Detection (Experience Paper). In Proceedings of the ISSTA'23: 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Seattle, WA, USA, 17–21 July 2023.
24. Gill, S.S.; Wu, H.; Patros, P.; Ottaviani, C.; Arora, P.; Pujol, V.C.; Haunschild, D.; Parlikad, A.K.; Cetinkaya, O.; Lutfiyya, H.; et al. Modern computing: Vision and challenges. *Telemat. Inform. Rep.* **2024**, *13*, 100116. [\[CrossRef\]](#)
25. Pargaonkar, S. Advancements in Security Testing: A Comprehensive Review of Methodologies and Emerging Trends in Software Quality Engineering. *Int. J. Sci. Res. (IJSR)* **2023**, *12*, 61–66. [\[CrossRef\]](#)
26. Chaleshtari, N.B.; Pastore, F.; Goknil, A.; Briand, L.C. Metamorphic Testing for Web System Security. *IEEE Trans. Softw. Eng.* **2023**, *49*, 3430–3471. [\[CrossRef\]](#)
27. Ami, A.S.; Moran, K.; Poshyvanyk, D.; Nadkarni, A. “False negative—That one is going to kill you”: Understanding Industry Perspectives of Static Analysis based Security Testing. In Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2024. [\[CrossRef\]](#)
28. Vyas, B. Security Challenges and Solutions in Java Application Development. *Eduzone Int. Peer Rev. Ref. Multidiscip. J.* **2023**, *12*, 268–275.
29. Singh, M.; Chauhan, S. A hybrid-extreme learning machine based ensemble method for online dynamic security assessment of power systems. *Electr. Power Syst. Res.* **2022**, *214*, 108923. [\[CrossRef\]](#)
30. Müller, M.; Salathé, M.; Kummervold, P.E. COVID-Twitter-BERT: A natural language processing model to analyse COVID-19 content on Twitter. *Front. Artif. Intell.* **2023**, *6*, 1023281. [\[CrossRef\]](#)
31. Aftan, S.; Shah, H. A Survey on BERT and Its Applications. In Proceedings of the 2023 20th Learning and Technology Conference (L&T), Jeddah, Saudi Arabia, 26–26 January 2023; pp. 161–166.
32. GitHub—nopSolutions/nopCommerce: ASP.NET Core eCommerce Software. nopCommerce Is a Free and Open-Source Shopping Cart. Available online: <https://github.com/nopSolutions/nopCommerce> (accessed on 16 October 2024).
33. GitHub—MadsKristensen/Miniblog. Core: An ASP.NET Core Blogging Engine. Available online: <https://github.com/madskristensen/Miniblog.Core> (accessed on 16 October 2024).
34. Ersoy, A.; Yıldız, O.T.; Özer, S. ORTPiece: An ORT-Based Turkish Image Captioning Network Based on Transformers and WordPiece. In Proceedings of the 2023 31st Signal Processing and Communications Applications Conference (SIU), Istanbul, Türkiye, 5–8 July 2023; pp. 1–4.
35. Liang, W.; Liang, Y. DrBERT: Unveiling the Potential of Masked Language Modeling Decoder in BERT pretraining. *arXiv* **2024**, arXiv:2401.15861.
36. Yang, Z.; Mitra, A.; Liu, W.; Berlowitz, D.; Yu, H. TransformEHR: Transformer-based encoder-decoder generative model to enhance prediction of disease outcomes using electronic health records. *Nat. Commun.* **2023**, *14*, 7857. [\[CrossRef\]](#) [\[PubMed\]](#)
37. Zheng, Y.; Lu, F.; Zou, J.; Hua, H.; Lu, X.; Min, X. De Novo Design of Target-Specific Ligands Using BERT-Pretrained Transformer. In *Chinese Conference on Pattern Recognition and Computer Vision (PRCV)*; Springer Nature Singapore: Singapore, 2023; pp. 311–322.
38. Vaswani, A. Attention is all you need. *Advances in Neural Information Processing Systems*. *arXiv* **2017**, arXiv:1706.03762.
39. Alissa, S.; Wald, M. Text Simplification Using Transformer and BERT. *Comput. Mater. Contin.* **2023**, *75*, 3479–3495. [\[CrossRef\]](#)
40. Kingma, D.P.; Ba, J. Adam: A method for stochastic optimization. *arXiv* **2014**, arXiv:1412.6980, 5.
41. Hinton, G.; Srivastava, N.; Swersky, K. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited On* **2012**, *14*, 2.
42. Robbins, H.; Monro, S. A stochastic approximation method. *Ann. Math. Stat.* **1951**, *22*, 400–407. [\[CrossRef\]](#)

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.