Robust and Uncertainty-Aware Software Vulnerability Detection Using Bayesian Recurrent Neural Networks

Orune Aminul
The University of Texas Rio Grande Valley

ROBUST AND UNCERTAINTY-AWARE SOFTWARE VULNERABILITY DETECTION USING BAYESIAN RECURRENT NEURAL NETWORKS

A Thesis

by

ORUNE AMINUL

Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: Electrical Engineering

The University of Texas Rio Grande Valley

August 2023

ROBUST AND UNCERTAINTY-AWARE SOFTWARE VULNERABILITY DETECTION USING BAYESIAN RECURRENT NEURAL NETWORKS

A Thesis
by
ORUNE AMINUL

COMMITTEE MEMBERS

Dr. Dimah Dera
Chair of Committee

Dr. Nantakan Wongkasem
Committee Member

Dr. Heinrich D. Foltz
Committee Member

August 2023

ABSTRACT

Aminul, Orune, Robust and Uncertainty-Aware Software Vulnerability Detection using Bayesian Recurrent Neural Networks. Master of Science (MS), August, 2023, 55 pp., 7 tables, 13 figures, references, 76 titles.

Software systems are prone to code defects or vulnerabilities, resulting in several cyberattacks such as hacking, identity breach and information leakage leading to system failure. Vulnerabilities in software systems have severe societal implications, including threats to public safety, financial damage, and even risks to national security. Identifying and mitigating software vulnerabilities is critical to protect organizations and societies from potential threats. Machine learning algorithms have employed models and classify possible distributions in software source code automatically. However, these algorithms are not robust to noise or malicious attacks and cannot quantify uncertainty in the model's output. Quantifying uncertainty in the vulnerability detection mechanism can inform the user of possible noise or perturbation in the source codes and holds the promise for the safe deployment of trustworthy algorithms in real-world security applications. We develop a robust software vulnerability detection framework using Bayesian Recurrent Neural Networks (Bayesian SVD). The proposed models detect source code vulnerabilities and simultaneously learn uncertainty in output predictions. The proposed Bayesian SVD adopts variational inference and optimizes the variational posterior distribution defined over the model parameters using the evidence lower bound (ELBO). Within each state, the first two moments of the variational distribution are transmitted through the recurrent layers. At the SVD models' output, the predictive distribution's mean indicates the vulnerability class, while the covariance matrix captures the uncertainty information. Extensive experiments on benchmark datasets reveal (1) the robustness of proposed models under noisy conditions and malicious attacks compared to the deterministic counterpart and (2) significantly higher uncertainty when the model encountered high levels of natural noise or malicious attacks, which serves as a warning for safe handling.

DEDICATION

To my parents and sister, for your unwavering support and inspiration. Thank you for believing in me and helping me achieve my goals. I dedicate this thesis to you both with all my love.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis supervisor, Dr. Dimah Dera, for her invaluable guidance, support, and encouragement throughout my Master's program. Her insightful feedback and constructive criticism have been instrumental in shaping this thesis.

I extend my appreciation to my thesis committee members, Dr. Nantakan Wongkasem and Dr. Heinrich D. Foltz, for their time and feedback. Their feedback and suggestions have greatly improved the thesis. Additionally, I sincerely appreciate Dr. Hasina Huq, the Chair of the Department of Electrical & Computer Engineering, for her support and guidance throughout my academic journey. I am grateful for the opportunities and resources provided by the department, which have been instrumental in completing this thesis.

Finally, I would like to thank my parents and my elder sister for their unwavering support and encouragement. Without their love and guidance, I would not have been able to complete this Master's program.

Thank you all for contributing to this thesis and helping me achieve this important milestone in my academic journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Motivation: Importance of Vulnerability Detection

The widespread usage and continuous evolution of software technology have significantly contributed to the growth of businesses, finances, medical care, and society [67, 75]. Software vulnerabilities, on the other hand, have emerged as an important safety concern to companies and institutions. A software vulnerability is described as a security hole, flaw, or defect within a software system that an unauthorized user can abuse, causing harm or theft [4]. Security vulnerabilities can have disastrous financial and societal consequences, causing significant economic and reputational harm to businesses, individuals, and countries [5,6,56,63]. For example, the well-known Heartbleed bug and the WannaCry ransomware attacks have shown current security flaws [11,42,52]. In recent years, the frequency and severity of software vulnerabilities targeting critical infrastructure systems have increased [14]. According to the national vulnerability database (NVD), the annual report of the United States Computer Emergency Readiness Team (US-CERT) in 2021 recorded 20,113 vulnerabilities, making it the fifth year in a row with a record number of high-risk vulnerabilities [1]. Identifying security vulnerabilities in software code has become difficult due to the rapid rise in software size and complexity, as these vulnerabilities rarely share similar features. In the following sections, we will provide two examples of critical applications where detecting vulnerabilities in software source codes is crucial.

1.1.1 Financial department

Due to the sensitivity of financial data and transactions, software vulnerability detection is critical in the financial sector. Financial institutions deal with enormous amounts of sensitive data, such as customer details, transaction records, and intellectual property [38]. The operations, clients, and financial assets of banks and insurance firms are significantly at risk as a result of cybersecurity issues. The risks include insider threats from employees with malicious intentions, data breaches that result in the theft of private customer information, and ransomware attacks that can disrupt operations and demand large ransom payments [31]. These phishing and social engineering attacks target customers and employees to gain unauthorized access and take advantage of human weaknesses. Cybersecurity problems can result in monetary losses, damage to credibility, regulatory non-compliance, legal liabilities, and losing customer confidence. In order to defend against these threats and safeguard software assets and activities, banks and insurance companies must implement robust cybersecurity measures, including software vulnerability detection. Therefore, in dealing with evolving cybersecurity threats, proactive software vulnerability detection is vital for retaining the integrity of financial systems.

1.1.2 Airport security

The detection of software vulnerabilities is essential in airport security. Airports depend heavily on software systems for various functions such as security monitoring, access control, passenger screening, baggage handling, and flight operations [35, 61]. Detecting software vulnerabilities is important to avoid potential security breaches. Protecting airport systems against unauthorized access, data breaches, and possible disruption to essential operations can be achieved by identifying and addressing vulnerabilities in software systems [46]. Moreover, software vulnerability detection assists in promoting legal compliance requirements at airports, such as those established by aviation security organizations, as well as ensuring the integrity and confidentiality of private passenger information. Unauthorized individuals may be able to pass weapons or other prohibited items through security checkpoints if airport security systems are compromised. This could pose a major security risk because it could allow terrorists to bring dangerous items onto an aircraft, endangering the safety of passengers. As a result, strong cybersecurity measures, such as robust software vulnerability detection, are vital for preventing breaches that could be used to circumvent airport security and compromise flight security.

1.2 Background and Problem Statement

Vulnerability analysis is the process of analyzing software source codes to identify vulnerabilities or flaws. For detecting vulnerabilities in the source codes, conventional static code analysis tools use preset rules and known patterns [72]. Examples of static methods include template-based analysis [43], code similarity detection [32], and symbolic execution. Although static vulnerability detection does not involve any code execution, Dam *et al.* claims that it can be more likely to have false alarms [19]. In contrast, dynamic analyzers repeatedly run programs with numerous test inputs on real or virtual processors to discover flaws. However, dynamic analyzers have their own restrictions, such as limited code coverage [64]. Examples of dynamic analysis include fuzziness testing and taint analysis [42].

Traditional machine learning algorithms, such as support vector machines (SVM) [30, 53], Random Forest (RF) [59], and k-nearest neighbor (KNN) [37], have been used for vulnerability detection. However, these traditional algorithms perform poorly on large datasets, including millions of source codes. Deep neural networks have shown great success in detecting vulnerabilities in software source codes [42]. The one-dimensional convolutional neural network (CNN) [19, 58] and recurrent neural networks [19] are the most popular in the literature. The challenge remains that deep neural networks are not robust to malicious attacks and noisy environments, which makes them unreliable as cyberattacks become increasingly diverse and complicated [13]. Building a robust model that can automatically identify various types of vulnerabilities in software source codes becomes difficult due to noisy and imperfect source codes and unknown or missing values in the training data.

1.2.1 Problem Statement

The current ML algorithms and deep neural networks suffer from two major limitations that affect their performance and reliability. Firstly, most of these algorithms are not "Robust" against noise. Even though many machine learning models have shown success at accurately predicting outputs under controlled environments, they often fail when exposed to noise, artifacts, and adversarial perturbations. In other words, they are not immune to environmental noise in the input data, and even tiny perturbations might result in erroneous outputs. Therefore, these techniques are unlikely to be suited to real-world applications where the data may be noisy or contain irregularities that could impair the accuracy of the model's decisions. Secondly, none of these algorithms quantify uncertainty, making it difficult to assess the reliability of their decision. As a result, we are unsure of how much we can trust the assessments made by these models, especially if they are implemented in security-related critical applications. Quantifying uncertainty in vulnerability detection can alert the user to potential noise or perturbation in the source codes and be deployed as trustworthy algorithms for real-world security applications. Uncertainty quantification is critical for reliable decision-making since it serves as a measure of the model's confidence or reliability. The effects of incorrect or uncertain predictions can be severe in many real-world situations, especially those involving security. As a result, to appropriately evaluate the model's trustworthiness, decision-makers need to be aware of the level of uncertainty in the output of decision-making algorithms.

1.3 Research Objectives and Contributions

In this work, we propose a novel software vulnerability detection framework employing a Bayesian Recurrent Neural Network, referred to as (SVD-RNN). We propose a mathematically established framework to estimate the uncertainty in model predictions using the Bayesian formulation. Assuming the RNN network parameters as random variables, we introduce a Gaussian distribution as a prior distribution over the parameters. After observing training data, we approximate the posterior distribution of the parameters using the variational inference (VI). The mean and covariance matrix of the variational distribution is then propagated across the hidden states, gates of the RNN networks, and non-linear activation functions in the SVD-RNN models. We use the first-order Taylor series approximation to estimate the variational moments (mean and covariance) after the non-linear activation functions of all gates and layers of the RNN networks. At the output of the SVD-RNN framework, we obtain the mean and covariance of the predictive distribution. The mean represents the source code vulnerability information, while the covariance matrix expresses uncertainty in the output decision.

In particular, the thesis has major contributions that can be summarized as follows:

1. We propose a Bayesian Recurrent Neural Network that can detect vulnerability in source code and quantify the uncertainty associated with the predicted vulnerability.

2. We adopt variational inference to approximate the posterior distribution of the parameters given the data. We propagate the mean and covariance of the variational posterior distribution through the network layers and non-linear functions.

3. We use first-order Taylor series approximation to estimate the first two moments of the variational distribution (mean and covariance) through the non-linear activation functions in the RNNs.

4. We conduct comprehensive experiments on three benchmark datasets, i.e., Draper VDISC [58], Big-Vul [21], and Semantics-based Vulnerability Candidate (SeVC) [41] datasets that contain labeled C/C++ source code functions.

5. We extensively analyze all three datasets under noisy environments by corrupting the test samples with Gaussian and adversarial attacks and comparing the performance of our Bayesian SVD algorithms against the state-of-the-art deterministic SVD algorithms.

6. We evaluate the robustness of our proposed model under high noise levels. We demonstrate uncertainty that increases significantly in the presence of high noise conditions.

CHAPTER II

LITERATURE REVIEW

2.1 Vulnerability Detection with Deep Neural Networks

Since the beginning of software development, manual code review has been employed as a technique for identifying software vulnerabilities. This guaranteed the quality and security of software systems. Manual code review, however, lost its value as a stand-alone vulnerability detection technique as software became more complicated over time. Two techniques for identifying software vulnerabilities have grown in popularity: static code analysis and dynamic code analysis. Static code analysis examines a piece of software source code without even running them [45, 51]. This strategy relies on the notion that specific coding patterns are linked to frequent vulnerabilities, such as buffer overflows or injection attacks. Static analysis tools can check the code for these patterns and flag any potential vulnerabilities. One drawback of static analysis is its dependency on a predefined set of rules and patterns to find vulnerabilities, which means it may fail to recognize new vulnerabilities. Moreover, using static code analysis to find vulnerabilities can be time-consuming and need specialized expertise [8]. On the other hand, Dynamic code analysis requires running a software program in a controlled setting while tracking the way it works to detect potential vulnerabilities [10, 12]. This approach relies on the idea that vulnerabilities can be discovered by looking at how software interacts with the environment, including input/output data or network relations. Dynamic analysis has the disadvantage that it can be time-consuming and expensive to conduct, and it may overlook some vulnerabilities that only exist in certain circumstances. The hybrid analysis combines the static and dynamic approaches and makes use of both of their strengths: while dynamic analysis can identify more complicated vulnerabilities during run-time, the static analysis can discover common vulnerabilities early in the software development process [44, 48]. Another automated software testing technique called "fuzz testing" involves sending random inputs to a program in order to locate vulnerabilities and flaws [34, 65]. Although these methods were successful in finding vulnerabilities in software in the early days, they can be labor-intensive, error-prone, and produce higher false positives, making it difficult for them to keep up with the rate at which software is developed. Machine learning, on the other hand, can automate and improve the productivity of software vulnerability detection by observing features in the code and behavior of the software. Large volumes of data, including code, logs, and network traffic, can be analyzed using ML-based algorithms to find patterns and abnormalities which can refer to a vulnerability. It is possible to reduce the time and skill necessary for vulnerability identification while increasing the accuracy and effectiveness of the whole process by applying ML-based techniques.

Modern machine learning (ML) and data-driven algorithms for automated vulnerability detection have become more convenient to implement due to the expanding availability of open-source software and repositories. An SVM classification method named VCCFinder, proposed by Perl *et al.*, identified potential vulnerabilities using codes taken from GitHub repositories [53]. Hovsepyan *et al.* used a bag-of-words (BoW) representation of Java source code to build an SVM in 2012 [30]. SVM is brittle to noisy data and performs poorly with large datasets [7]. Other conventional ML techniques have also been applied, including RF, KNN, Decision tree (DT), and neural network (NN). These approaches have been compared in terms of performance using single and numerous types of code vulnerabilities gathered from five distinct projects [9]. However, conventional ML approaches fail to recognize the semantic nature of the code components [58]. Codes with similar complexity matrices often have different semantic interpretations and, consequently, different vulnerability labels.

Deep neural networks have recently been utilized for processing sequential data, including software source codes. Using word embedding, Han *et al.* established a one-layer CNN for multi-class vulnerability identification. The model takes the syntactic and semantic characteristics of the code attributes into consideration [25]. Using a recurrent neural network (RNN), Zou *et al.* presented the deep learning-based μVulDeePecker system for multi-class vulnerability detection [76]. Vulnerabilities brought on by C/C++ library/API function calls can be detected with μVulDeePecker. For the automatic patching of software flaws, Harer *et al.* used adversarial learning in the encoder-decoder architecture of neural machine translation systems [26]. A long short-term memory (LSTM) model was developed by Dam *et al.* to autonomously acquire both semantic and syntactic features of source codes [19]. Russell *et al.* used both the CNN and RNN architectures independently to compare their performance metrics [58]. Li *et al.* suggested a hybrid neural network consisting of a CNN and an RNN for learning global as well as local vulnerability features [39].

Unlike feed-forward networks such as CNNs or fully connected networks (FCNs), RNNs are specifically adapted to processing sequential inputs of software source codes [42]. Two noteworthy RNN variants are long short-term memory (LSTM) and gated recurrent unit (GRU) [17,66]. Internal gate structures in both LSTM and GRU networks assist in preserving relevant details and in pruning out unnecessary data from earlier time steps, according to [17]. An LSTM-based method for capturing long contextual relationships and automatically learning features for predicting vulnerabilities in software source code has been presented by Dam *et al.* in [19]. Tang *et al.* compared the performance of bidirectional LSTM utilizing two separate data preprocessing approaches [62]. Xiao *et al.* deployed a GRU structure to reduce losing the syntax and semantic details in software code components [71]. On the other hand, Wang *et al.* applied both CNN and GRU to detect vulnerabilities in the source code files [69].

2.2 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are designed for processing and analyzing sequential data. RNNs are often utilized in speech recognition, language modeling, text generation, image description, and video tagging [60]. RNNs differ from feed-forward neural networks, often referred to as multi-layer perceptrons (MLPs), in how information passes through the network. RNNs retain a memory of previous inputs and use it to drive their current outputs, unlike feed-forward neural networks, which only transmit information in a single pass, as illustrated in Fig. 2.1. Conventional

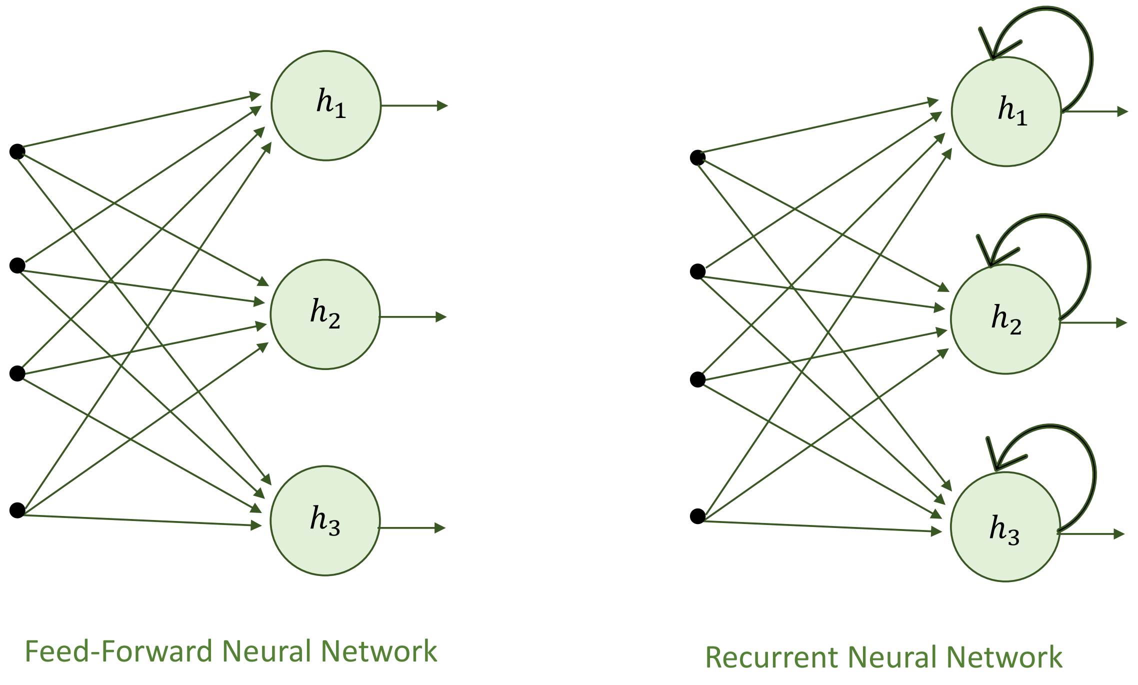


Figure 2.1: Basic Structure of Feed-Forward vs Recurrent neural network

neural networks assume input features are independent of each other. RNNs, on the other hand, process the input vector as a sequence considering the correlation between its entries. For instance, it is necessary to remember and store the previous tokens or blocks in some type of memory to predict the next token of any source code snippet or the next block of the source code. In RNNs, the hidden layers preserve the sequential information about the input.

At time step t, we consider having an input with size K, i.e., $\mathbf{x}^{(t)} \in \mathbb{R}^{K\times 1}$. The hidden variable (or hidden state) at time step t can be represented by $\mathbf{h}^{(t)} \in \mathbb{R}^{H\times 1}$. Here, the number of nodes in the hidden layer is denoted by H. In contrast to the MLP, we compute the weighted combination of features saved in the hidden state $\mathbf{h}^{(t-1)}$ from the previous time step by multiplying it with the weight parameter $\mathbf{W}^{(\mathbf{hh})} \in \mathbb{R}^{H\times H}$, that determines the important features from the previous time step and uses those features in the current time step. The hidden state also computes the weighted combination of input features at the current time step, $\mathbf{x}^{(t)}$ by multiplying it with the weight matrix $\mathbf{W}^{(\mathbf{xh})} \in \mathbb{R}^{H\times K}$. Fig. 2.2 shows the structure of the RNN network. The single RNN cell takes an input feature vector, $\mathbf{x}^{(t)}$, at time t and a hidden state $\mathbf{h}^{(t-1)}$ from the prior time step as

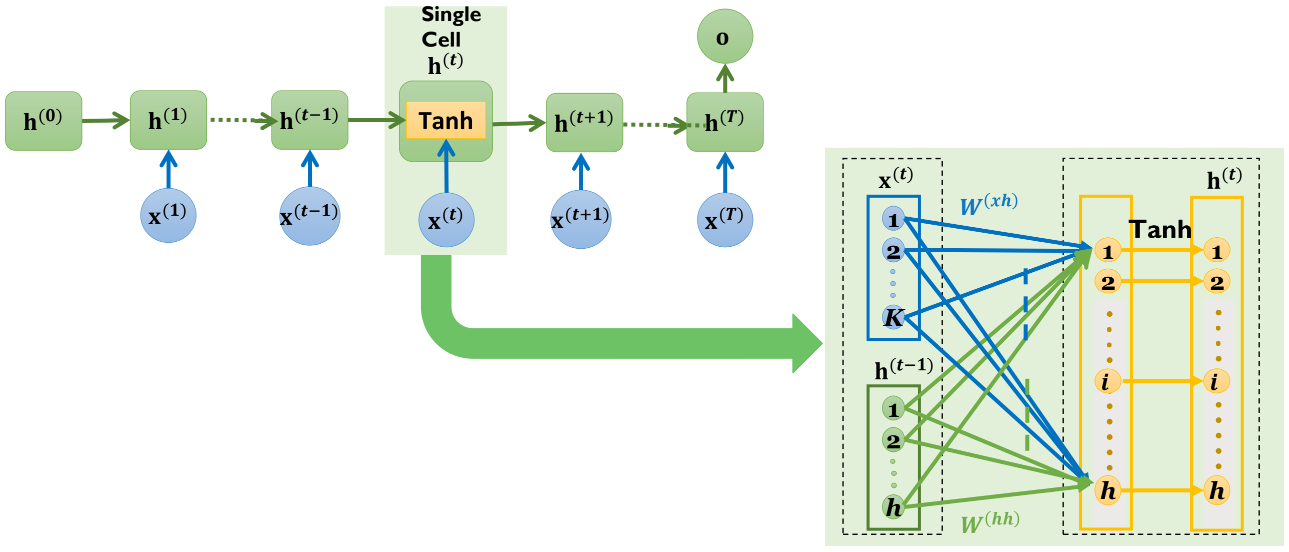


Figure 2.2: Internal structure of RNN

inputs. After that, it generates a new hidden state, $\mathbf{h}^{(t)}$, following the Equation (2.1) below:

$$\mathbf{h}^{(t)} = f(\mathbf{W}^{(\mathbf{xh})}\mathbf{x}^{(t)} + \mathbf{W}^{(\mathbf{hh})}\mathbf{h}^{(t-1)} + \mathbf{b}^{(\mathbf{h})}). \tag{2.1}$$

In Equation (2.1), f represents a non-linear activation function, such as the sigmoid, hyperbolic tangent (Tanh), or rectified linear unit (ReLU) function. The bias term is denoted by $\mathbf{b}^{(\mathbf{h})} \in \mathbb{R}^H$ in the Equation (2.1). The variables $\mathbf{h}^{(t)}$ and $\mathbf{h}^{(t-1)}$ store and record the sequence's previous information up to the current time step. Following the hidden layer, at the output layer, the output for time step t is given by:

$$\mathbf{o}^{(t)} = sigmoid(\mathbf{W}^{(\mathbf{ho})}\mathbf{h}^{(t)} + \mathbf{b}^{(\mathbf{o})}). \tag{2.2}$$

Here, the activation function, σ, is also a non-linear activation that is the softmax function if we have a multi-class classification problem or the sigmoid function if we have a binary classification problem. The parameters of the output layer are the weight matrix $\mathbf{W}^{(\mathbf{ho})}$ and the bias term $\mathbf{b}^{(\mathbf{o})}$.

In general, the back-propagation through time (BPTT) algorithm is used to optimize the RNN during training. This involves calculating the gradient of the loss with respect to the network parameters at each time step while spreading the RNN over a certain number of time steps. The network parameters are then updated using optimization techniques like stochastic gradient descent (SGD), adaptive moment estimation (Adam), and gradient clipping.

RNNs are ideal for processing sequential data due to their recurrent structure. They are effective for applications like sentiment analysis, machine translation, and speech recognition because they can detect temporal correlations between inputs. RNNs can mimic complicated non-linear dynamics in the input data because they can keep track of previous inputs and use that memory to influence their current outputs. Finally, RNNs can be improved by back-propagation through time (BPTT), which enables them to alter their internal state according to feedback from the output and learn from previous inputs.

However, RNNs are susceptible to the vanishing gradient and exploding gradient problems. The vanishing gradient problem arises when the gradient of the loss with respect to the network's parameters gets extremely small values, making it challenging for the network to update the parameters effectively. This can occur in RNNs when the gradients of the output with respect to the network weights are multiplied repeatedly by the same weight matrix for very long input sequences, resulting in the gradients getting gradually smaller as they propagate over time. On the other hand, the exploding gradient problem arises when the gradient of the loss with respect to the network's parameters increases significantly, making it challenging for the network to converge to an appropriate solution. RNNs can also be subject to overfitting and may need a lot of data to train well. Because of these shortcomings, more complex variations of the fundamental RNN architecture have been created, like the Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks, which are intended to deal with these challenges.

2.2.1 Long Short-Term Memory (LSTM)

A form of Recurrent Neural Network (RNN) architecture called Long Short-Term Memory (LSTM) was introduced to address the problem of vanishing gradients in regular RNNs [73]. The vanishing gradient problem occurs when gradients become very small during back-propagation, making it difficult for the network to learn long-term dependencies in sequential data [29]. This problem is addressed by LSTM by including a memory cell that has the capacity to selectively remember or forget information over time. Three gates — the input gate, the forget gate, and the output gate are in charge of controlling the flow of information into and out of the memory cell. The fact that LSTMs offer gating of the hidden state is the main distinction between LSTMs and plain vanilla RNNs. As a result, the gate structure of the LSTM network determines when a hidden state needs to be updated as well as when it needs to be reset. For example, the network will learn not to update the hidden state after the first observation if the first token happens to be significant. The network will also learn to ignore observations that are insignificant.

2.2.1.1 ***Input Gate, Forget Gate, and Output Gate.*** As shown in Fig. (2.3), the three LSTM gates require the input at the present time step t as well as the hidden state from the previous time step $t-1$ as inputs. The input, forget, and output gates' values are generated using sigmoid activation functions across three fully connected layers. Due to having the sigmoid activation, all the gates have values between $(0,1)$. At time step t, for a given input, $\mathbf{x}^{(t)} \in \mathbb{R}^{K\times 1}$ and $\mathbf{h}^{(t-1)} \in \mathbb{R}^{H\times 1}$ as the hidden state of the previous time step, we can compute the input gate $\mathbf{i}^{(t)} \in \mathbb{R}^{H\times 1}$, forget gate $\mathbf{f}^{(t)} \in \mathbb{R}^{H\times 1}$ and output gate $\mathbf{o}^{(t)} \in \mathbb{R}^{H\times 1}$ using the following equations:

$$\mathbf{i}^{(t)} = sigmoid(\mathbf{U^{(i)}}\mathbf{x}^{(t)} + \mathbf{W^{(i)}}\mathbf{h}^{(t-1)} + \mathbf{b^{(i)}}). \tag{2.3}$$

$$\mathbf{f}^{(t)} = sigmoid(\mathbf{U^{(f)}}\mathbf{x}^{(t)} + \mathbf{W^{(f)}}\mathbf{h}^{(t-1)} + \mathbf{b^{(f)}}). \tag{2.4}$$

$$\mathbf{o}^{(t)} = sigmoid(\mathbf{U^{(o)}}\mathbf{x}^{(t)} + \mathbf{W^{(o)}}\mathbf{h}^{(t-1)} + \mathbf{b^{(o)}}). \tag{2.5}$$

Here, $\mathbf{U^{(i)}}$, $\mathbf{U^{(f)}}$, $\mathbf{U^{(o)}} \in \mathbb{R}^{H\times K}$ and $\mathbf{W^{(i)}}$,$\mathbf{W^{(f)}}$, $\mathbf{W^{(o)}} \in \mathbb{R}^{H\times H}$ represent the weight parameter for the input, forget and output gates, and $\mathbf{b^{(i)}}$,$\mathbf{b^{(f)}}$, $\mathbf{b^{(o)}} \in \mathbb{R}^{H\times 1}$ denote the bias parameter for the respective gates. The input and forget gates work together to determine which data should be stored in the cell state. The forget gate regulates how much old data should be discarded, while the input gate

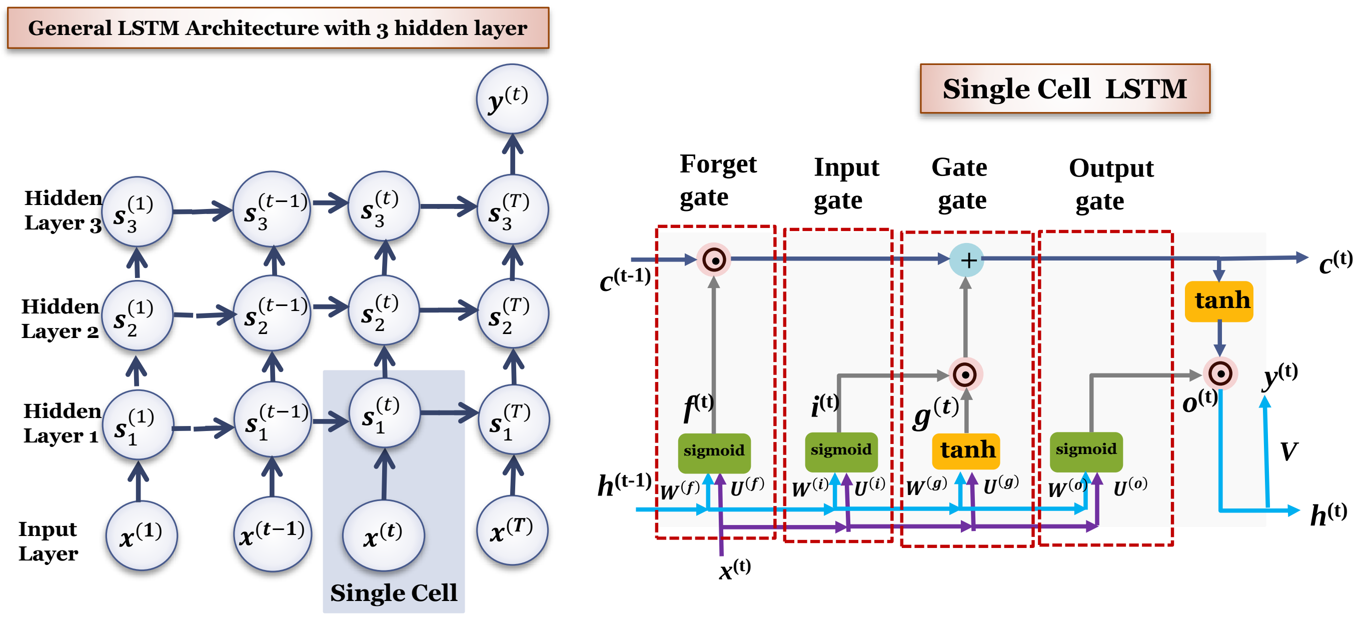


Figure 2.3: Internal structure of LSTM

controls how much new data should be added. The output gate decides how much data from the cell state should be obtained from the hidden state at the time step t.

2.2.1.2 *Gate gate.* Its operation resembles that of the first three gates discussed before, with the addition that the activation function is a Tanh function with a value range of $(-1,1)$. Thus, at time step t, the gate gate $\mathbf{g}^{(t)} \in \mathbb{R}^{H\times 1}$ is given in Equation (2.6).:

$$\mathbf{g}^{(t)} = Tanh(\mathbf{U}^{(\mathbf{g})}\mathbf{x}^{(t)} + \mathbf{W}^{(\mathbf{g})}\mathbf{h}^{(t-1)} + \mathbf{b}^{(\mathbf{g})}). \tag{2.6}$$

The weight parameters, in this case, are $\mathbf{U}^{(\mathbf{g})} \in \mathbb{R}^{H\times K}$ and $\mathbf{W}^{(\mathbf{g})} \in \mathbb{R}^{H\times H}$, whereas the bias parameter is denoted by $\mathbf{b}^{(\mathbf{g})} \in \mathbb{R}^{H\times 1}$.

2.2.1.3 *Internal state.* The new information added to the memory cell state can be controlled by the internal state, also known as the candidate memory cell state. Considering both the input gate $\mathbf{i}^{(t)}$ and the forget gate $\mathbf{f}^{(t)}$, the internal state $\mathbf{c}^{(t)}$ can be formulated in Equation (2.7) as follows:

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \odot \mathbf{g}^{(t)}. \tag{2.7}$$

Here, $\odot$ represents the Hadamard product operation which is an element-wise multiplication. The internal state of the memory cell, $\mathbf{c}^{(t-1)}$, from the previous time step, remains unchanged and moves unaltered to the current time step if the forget gate is always set to 1 and the input gate is set to 0. However, because of input gates and forget gates, the model has the flexibility to determine if it should maintain the current value as is or modify it in response to new inputs.

2.2.1.4 ***Current state.*** The next step is to compute the output of the memory cell, or the hidden state $\mathbf{h}^{(t)}$, as it appears to other layers. The internal state of the memory cell first goes through Tanh activation, then an element-wise multiplication with the output gate is performed. It assures that the values are always between $(-1,1)$. We can formulate the operation as follows:

$$\mathbf{h}^{(t)} = Tanh(\mathbf{c}^{(t)}) \odot \mathbf{o}^{(t)}. \tag{2.8}$$

When the output gate is close to 1, we let the internal state of the memory cell have an influence on the preceding layers; however, when the output gate is close to 0, we prevent the current memory from having an impact on the network's subsequent layers at present step. A memory cell can accumulate data over several time steps without affecting the rest of the network (as long as the output gate accepts values close to 0). Still, the network will be influenced when the output gate switches to values close to 1.

2.2.2 Gated Recurrent Unit (GRU)

A Gated Recurrent Unit (GRU) is a form of RNN designed to overcome the vanishing gradient problem in regular RNNs. GRU was developed as a simpler version of LSTM that uses fewer parameters to achieve comparable performance. With fewer parameters and less computation complexity, GRU incorporates the gating mechanisms of LSTM and, at the same time, speeds up training and reduces the risk of overfitting. The GRU architecture is also more straightforward to develop and less prone to overfitting on small datasets than the LSTM because it has fewer gates. Using the gating mechanism, GRUs allow the network to more effectively recognize long-term dependencies by selectively adding and removing information from the previous step. Here, we

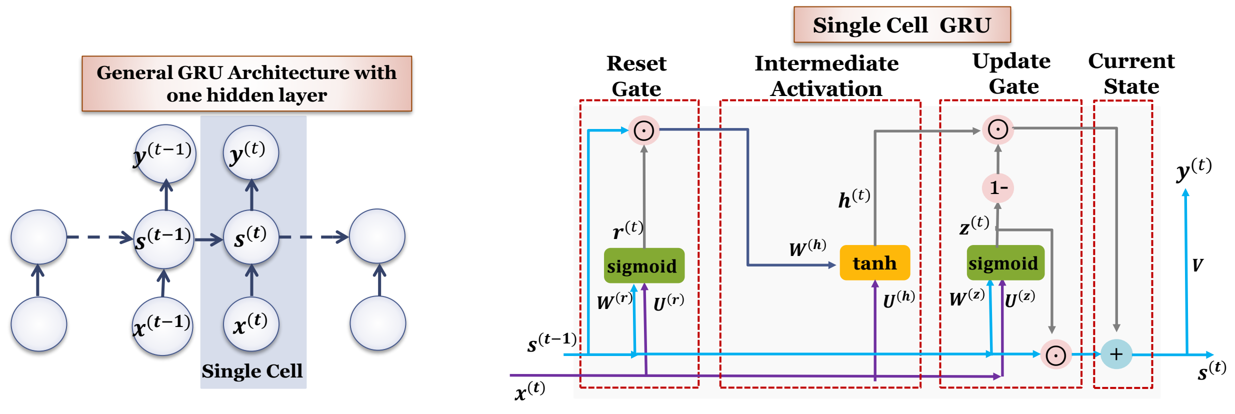


Figure 2.4: Internal structure of GRU

have the reset gate and the update gate, as compared to the three gates we had in the LSTM.

2.2.2.1 *Reset Gate and Update Gate.* The reset gate controls the information generated by the GRU cell and the update gate in the gating mechanism. Fig. (2.4) shows the inputs of the current time step and the hidden state of the previous time step for both the reset and update gates in the GRU. Similar to LSTMs, sigmoid activations are applied to these gates to ensure that the values are always between $(0,1)$. The reset gate, intuitively, regulates how much of the prior state we might still want to keep. Similarly, an update gate would allow us to identify how much of the new state is just an exact copy of the old state. Two fully connected layers with a sigmoid activation function provide the outputs of the two gates. At the time step t, given input $\mathbf{x}^{(t)} \in \mathbb{R}^{K\times 1}$ and the hidden state of the previous time step $\mathbf{s}^{(t-1)}$, we can compute the reset gate $\mathbf{r}^{(t)} \in \mathbb{R}^{H\times 1}$ and update gate $\mathbf{z}^{(t)} \in \mathbb{R}^{H\times 1}$ using the following equations:

$$\mathbf{r}^{(t)} = sigmoid(\mathbf{U}^{(\mathbf{r})}\mathbf{x}^{(t)} + \mathbf{W}^{(\mathbf{r})}\mathbf{s}^{(t-1)} + \mathbf{b}^{(\mathbf{r})}). \tag{2.9}$$

$$\mathbf{z}^{(t)} = sigmoid(\mathbf{U}^{(\mathbf{z})}\mathbf{x}^{(t)} + \mathbf{W}^{(\mathbf{z})}\mathbf{s}^{(t-1)} + \mathbf{b}^{(\mathbf{z})}). \tag{2.10}$$

Here, $\mathbf{U}^{(\mathbf{r})}, \mathbf{U}^{(\mathbf{z})} \in \mathbb{R}^{H\times K}$ and $\mathbf{W}^{(\mathbf{r})}, \mathbf{W}^{(\mathbf{z})} \in \mathbb{R}^{H\times H}$, represent the weight parameter of the reset and update gate, and $\mathbf{b}^{(\mathbf{r})}$, $\mathbf{b}^{(\mathbf{z})} \in \mathbb{R}^{H\times 1}$ denote the bias parameters for the respective gates.

The update gate enables the model to decide how much old data from earlier time steps should be transmitted to the future. This is particularly useful because the model has the capability of copying all previous data and removing the possibility of vanishing gradient challenges. The Update gate helps identify long-term dependencies, whereas the reset gate captures short-term dependencies in the input sequences.

2.2.2.2 *Intermediate hidden state.* To get the intermediate hidden state, $\mathbf{h}^{(t)}$, we incorporate the reset gate, $\mathbf{r}^{(t)}$ with the previous hidden state, $\mathbf{s}^{(t-1)}$ given by the following equation:

$$\mathbf{h}^{(t)} = Tanh(\mathbf{U}^{(\mathbf{h})}\mathbf{x}^{(t)} + \mathbf{W}^{(\mathbf{h})}(\mathbf{s}^{(t-1)} \odot \mathbf{r}^{(t)}) + \mathbf{b}^{(\mathbf{h})}). \tag{2.11}$$

The weight parameters, in this case, are $\mathbf{U}^{(\mathbf{h})} \in \mathbb{R}^{H \times K}$ and $\mathbf{W}^{(\mathbf{h})} \in \mathbb{R}^{H \times H}$, whereas the bias parameter for the intermediate hidden state is denoted by $\mathbf{b}^{(\mathbf{h})} \in \mathbb{R}^{H \times 1}$. Note that the *Tanh* activation function is used in this state. Whenever the output of the reset gate is close to 1, the GRU cell fully incorporates the previous hidden state into the candidate activation without any changes. In this instance, the candidate activation will be calculated using a weighted sum of the input at the current time step and the previous hidden state, with the weights set by the reset gate. The previous hidden state's weights will be set to their maximum value, which indicates that the candidate activation will be calculated using the entire prior hidden state.

When the reset gate's output is 0, the GRU cell discards the previous hidden state entirely and computes the candidate activation only using the received input. In this scenario, the candidate activation will only be calculated using the input at the current time step without considering the previous hidden state. The update gate will then decide how much of this new information needs to be included in the current hidden state.

2.2.2.3 *Current state.* Lastly, we consider the role of update gate $\mathbf{z}^{(t)}$. By taking element-wise convex combinations of $\mathbf{s}^{(t-1)}$ and $\mathbf{h}^{(t)}$ with update gate $\mathbf{z}^{(t)}$, we acquire the current state given in the Equation (2.12) as follows:

$$\mathbf{s}^{(t)} = \mathbf{z}^{(t)} \odot \mathbf{s}^{(t-1)} + (1 - \mathbf{z}^{(t)}) \odot \mathbf{h}^{(t)}. \tag{2.12}$$

The extent to which the new candidate hidden state $\mathbf{h}^{(t)}$ and the old hidden state $\mathbf{s}^{(t-1)}$ are combined to generate the new hidden state $\mathbf{s}^{(t)}$ is determined by the update gate $\mathbf{z}^{(t)}$. As we observe in the Equation (2.12), any time $\mathbf{z}^{(t)}$ becomes 0, the information from the previous hidden state is completely ignored. $\mathbf{s}^{(t)}$ gets the value of the new candidate hidden state. If $\mathbf{z}^{(t)}$ is 1, information from the previous hidden state is retained. Ultimately, the most important information gets passed from one state to the next.

Both LSTM and GRU are variants of RNNs that deal with the vanishing gradient problem and handle long-term dependencies by utilizing the gating mechanisms. When used for applications like speech recognition and language modeling, LSTM has the advantage of being able to simulate longer-term dependencies through input, output, and forget gates. While GRU still achieves comparable performance to LSTM with fewer parameters and less computational complexity, it has the advantage of being simpler and less prone to overfitting on small datasets. The decision between LSTM and GRU comes down to the precise specifications of the task assignment.

2.3 Bayesian Inference in RNN

One of the early approaches to incorporating Bayesian inference into neural networks was Hamiltonian Monte Carlo (HMC), a Markov chain Monte Carlo (MCMC)-based approach for generating samples from the posterior distribution. However, there were several computational complexities that made this approach impractical [50]. In [27], Harrison *et al.* suggested a technique for automated story-making for an LSTM network utilizing MCMC sampling. Although the technique demonstrated effectiveness, the resulting stories lack semantic interpretability in the present state. Traditional MCMC techniques estimate the posterior distribution using the complete dataset, which can be computationally expensive and not feasible for big datasets. This problem is addressed by stochastic gradient MCMC, which is more scalable because it uses subsets of the data. The stochastic gradient MCMC method combined MCMC sampling techniques with stochastic gradient descent (SGD) by adding a perturbation to the gradient during training [15, 49, 70].

Laplace approximation assumed that the posterior is a Gaussian distribution and was adopted in earlier attempts to incorporate Bayesian inference into neural networks [47]. Using Laplace approximation Chien *et al.* introduced a Bayesian approach to regularize a recurrent neural network language model for continuous speech recognition [16]. The maximum a posteriori (MAP) estimate was employed to calculate the posterior distribution's mean. Given the observed data, it generated a point estimate reflecting the most likely parameter values. The covariance can be approximated by the inverse of the Hessian matrix of the negative log-likelihood function calculated at the MAP estimation. The negative log-likelihood function and inverting the Hessian matrix for large-scale models like DNN is computationally expensive, making the solution intractable [57]. The posterior approximation techniques, expected propagation (EP), and assumed density filtering (ADF) repeatedly perform simple local computations for each data point for estimating the posterior distribution [23, 28, 40]. In order to improve the Gaussian posterior approximation for regression problems, Hernandez-Lobato and Adams introduced the probabilistic back-propagation (PBP) method in [28]. Later, PBP was further developed by Ghosh et al. to include multi-class classification problems [23]. By doing numerous ADF passes over the data, Hernandez-Lobato and Adams suggested ADF approximation eliminated dependence on order; nevertheless, the complete EP implementation was not feasible for DNNs because of its extensive computational and storage costs [40]. Variational inference (VI), a conventional approach for estimating posterior densities, has recently been effectively adapted for various forms of RNN [18, 22, 55] for time series data.

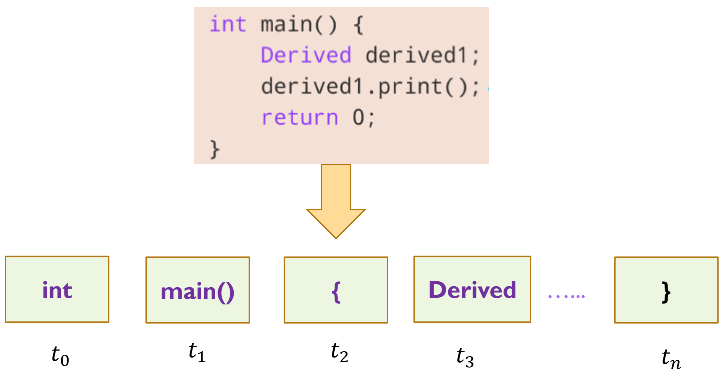
CHAPTER III

METHODOLOGY

3.1 Source code Preprocessing

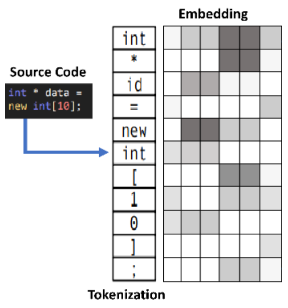
Software source codes are usually developed in high-level programming languages such as C/C++, Java, and Python. These languages are text-based and consist of words and phrases that were adapted from natural languages. Here, we approach the source codes as sequential data, analogous to the way sentences in natural languages are constructed, where each word in a phrase has some relationship to the adjacent words. This makes it possible to employ methods and models from natural language processing (NLP) to analyze the code, comprehend it, and extract valuable information. Sentences in natural language are made up of words that are arranged in a particular order and contain semantic and syntactic links. Similarly, statements and expressions are written in a certain order in source code, and each line or section of code usually has some relationship to the lines that came before it. For instance, function calls may rely on functions created earlier in the code, and variable declarations frequently depend on variables that have already been defined. As shown in 3.1, the code function can be represented as a sequence of variables and operators starting from time, t_o upto t_n. We can use sequential modeling approaches from NLP to capture the underlying patterns and dependencies contained in the source code by seeing it as sequential data. As a result, the semantic and syntactic characteristics of the source code will be preserved.

Because neural networks only accept real-valued inputs, the source code must be preprocessed in order to be converted into numerical real-valued input given the sequential nature of the original code. Two preprocessing steps are performed: 1) lexical analysis or tokenization, i.e., converting a code into a sequence of tokens and assigning indices to those tokens in the sequence



```
int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

Figure 3.1



```
int * data =
new int[10];
```

Figure 3.2: Feed-Forward vs Recurrent neural network

and 2) code token embedding, i.e., mapping the indices into real-values input matrix using a structure-preserving map [74]. Fig. 3.2 illustrates the data preprocessing of source codes before being fed into the neural network.

3.1.1 Tokenization

Tokenization is a crucial technique in Natural Language Processing (NLP) that includes decomposing a text into smaller pieces known as tokens. Depending on the level of granularity that is desired (character, word, subword level), these tokens can be single words, subwords, or even individual characters in their own right. In the field of natural language processing (NLP), tokenization is of the utmost importance since it enables efficient text preprocessing, vocabulary building, feature extraction, text representation, and numerous tasks involving language analysis. It lays the groundwork for subsequent processing and modeling of textual data, making it possible for algorithms to comprehend and operate productively with natural language. The process separates a large continuous code into individual tokens (small code segments). Analyzing sequences of tokens helps interpret source codes. The literal representations of strings, characters, and floats are referred to as type-specific placeholder tokens in source codes. [58, 74]. The tokenizer collects tokens from the original source codes used as training data to build a constrained vocabulary. As a result, each entry in the vocabulary will correspond to each token collected from the training examples. Each token in the dictionary will have a unique ID (or index, i.e., integer values). Any token or phrase on the test set that is unavailable in the vocabulary will be considered Out-Of-Vocabulary (OOV) and will be allocated a default ID. After building the vocabulary, we map the indices to the original code sequences. Tokenization is the process that takes a lengthy, complex source code and generates a collection of tokens from it, as shown in the example in Fig. 3.3 (a). In this illustration, we extracted three tokens from a single line of code. The concept of tokenization has advantages in text preparation because it eliminates noise and makes the input data more consistent. It also plays an important part in the process of creating new languages by producing an exhaustive list of tokens that are all unique. In order for machine learning algorithms to extract valuable features from the text, tokens are used as attributes in these algorithms. In addition, tokenization makes it possible to convert written text into numerical representations and simplifies the process of analyzing and comprehending text data. Thus, tokenization is essential for properly structuring source code, developing features, and enabling computers to analyze and comprehend software source code.

3.1.2 Code Token Embedding

The technique of encoding words or textual data as dense vectors in a high-dimensional space is referred to as embedding in the context of NLP. Models will be better able to comprehend and make sense of textual material if word embedding techniques are successful in capturing the semantic and contextual links between words. Word embeddings are generated by training a neural network model on a vast amount of text as part of the process of establishing the embeddings. The model learns to make predictions about the contexts in which words will appear based on the words that immediately surround them. The model will modify the weights of its internal layers as part of

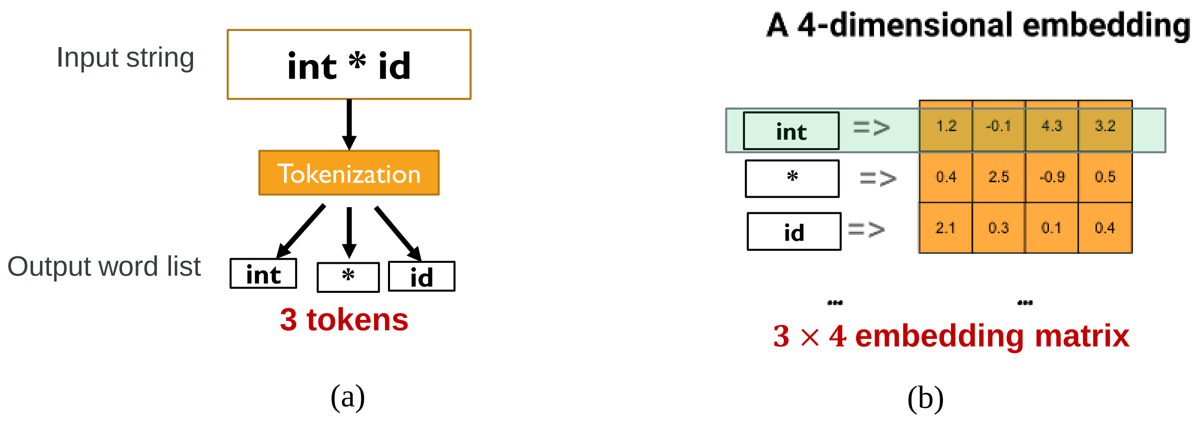


Figure 3.3: Tokenization and embedding of input source code function

this training process in order to reduce the error in its predictions. The hidden layer's weights, which are also referred to as the embedding layer, are what "capture" the semantic associations between words. After that, the word embeddings are constructed using the learned weights. Word2Vec, also known as "Global Vectors for Word Representation," and fastText are both examples of popular word embedding techniques. Because these word embeddings preserve the underlying semantic and syntactic qualities of words, they allow textual data to be represented and analyzed more effectively. The proposed models take sequences of code tokens as inputs. Before feeding the indices of code tokens into our models, we convert them into K-dimensional, continuous and real-valued vectors, referred to as *embedding*. Thus, the tokens' indices are converted into one-hot encoded vectors after tokenization. The length of the one-hot encoded vectors is equal to the vocabulary size. Then, the embedding layer linearly transforms the one-hot encoded vectors into vector representations of K dimensions.

The result is an embedded matrix of size $\tau \times K$, i.e., $\mathbf{X}^{(n)} \in \mathbb{R}^{\tau \times K}$, where τ is the tokens sequence length, which is the number of sequential tokens that are considered together, K is the size of the embedding vectors, and $n = 1, \cdots, N$ refers to the number of input token sequences or equivalently embedding matrices. The embedding matrix acts like a look-up table, where the i^{th} row in the embedding matrix is an embedded vector for the i^{th} token.

As an illustration, each of the three tokens in Fig. 3.3(b) has a four-dimensional vector representation that ultimately forms a 3×4 matrix. In our experiment, we adjust the embedding dimension, K, based on the complexity of the dataset (original source code). A higher dimensional embedding incorporates fine-grained token-to-token correlations but requires more training examples and, hence, higher computational complexity. The embedding matrix is learned during training via back-propagation [74].

3.2 Software Vulnerability Detection using Bayesian Sequence Models

We apply the tokenization and embedding to obtain the training dataset, $\mathscr{D}$, that consists of N token sequences. Each token sequence is represented by the embedding matrix $\mathbf{X}^{(n)}$, while the output, i.e., vulnerability labels $\mathbf{y}^{(n)} \in \mathbb{R}^C$ with C denotes the number of vulnerability classes, i.e., $\mathscr{D} = \{\mathbf{X}^{(n)}, \mathbf{y}^{(n)}\}_{n=1}^{N}$. Let $\mathscr{W} = \{\mathbf{W}^{(l)}\}_{l=1}^{L}$ be the sequence model's parameters (weights and biases) with L stacked layers.

3.2.1 Bayesian Formulation

Bayesian neural networks are a form of neural network that can be used to determine the degree of prediction uncertainty. We treat the network weights in Bayesian neural networks as random variables with a prior probability distribution. Before we observe any data, the prior distribution represents our initial weight assumptions. After observing the data, we establish our opinions about the weights defined by the posterior distribution. The likelihood distribution of the data given the weights multiplied by the prior distribution is proportional to the posterior distribution. The data and weights influence the likelihood distribution. The only factor affecting the prior distribution is the initial weight values.

To perform the Bayesian estimation, in the proposed Bayesian software vulnerability detection models, i.e., Bayesian SVD-LSTM and Bayesian SVD-GRU, the model's parameters $\mathscr{W}$ are assumed to be random variables with a Gaussian prior distribution $p(\mathscr{W})$. We consider all these parameters independent within and across the network layers. The independent assumption allows 1) to extract independent features across various network layers and 2) to establish a tractable optimization problem, as estimating the joint distribution of all layers is computationally challenging.

Now true posterior distribution, $p(\mathscr{W}|\mathscr{D})$, which captures the total knowledge about the network parameters, after observing the training dataset, $\mathscr{D}$ can be computed by employing Bayes' rule in Equation (3.1).

$$p(\mathscr{W}|\mathscr{D}) = \frac{p(\mathscr{D}|\mathscr{W})p(\mathscr{W})}{\int p(\mathscr{D}|\mathscr{W})p(\mathscr{W})d\mathscr{W}} \tag{3.1}$$

On the right-hand side, we have the likelihood distribution, $p(\mathscr{D}|\mathscr{W})$ of the data given the weights $\mathscr{W}$, multiplied by prior $p(\mathscr{W})$ divided by the marginal likelihood distribution of the data. However, the integration term in the denominator requires integrating every potential value for the model's parameters. The model parameters in DNNs are usually high-dimensional, and the integration is made even more difficult by the existence of non-linearities. As a result, utilizing the Bayesian technique to directly estimate the posterior distribution is no longer practical due to the prohibitively expensive calculation needed to calculate this integral. So exact Bayesian estimation of the true distribution can not be performed. In order to address the complexity of DNNs and make Bayesian inference more manageable, approximation Bayesian inference techniques, like variational inference and MCMC, have been developed.

3.2.2 Variational inference

Variational inference (VI) is an approximation algorithm for performing Bayesian inference by estimating the posterior distribution over the latent variables in a latent variable model when the actual true posterior is inaccessible. It is a commonly used machine learning methodology that estimates complex probability distributions through optimization methods. Due to this characteristic, VI converges faster than traditional techniques like MCMC sampling. VI tries to approximate the posterior using a "well-behaved" distribution. This implies that integrals are calculated so that the more accurate the estimate, the more precise the approximation.

VI involves taking into consideration a parameterized variational posterior distribution $q_\phi(\mathscr{W})$ and then estimating the true posterior $p(\mathscr{W}|\mathscr{D})$. This optimization is done by minimizing the Kullback-Leibler (KL) divergence between the variational posterior distribution $q_\phi(\mathcal{W})$ and the true unknown posterior distribution $p(\mathcal{W}|\mathcal{D})$ [68].

$$\mathrm{KL}\left[q_\phi(\mathcal{W})\|p(\mathcal{W}|\mathcal{D})\right] = \int q_\phi(\mathcal{W}) \log \frac{q_\phi(\mathcal{W})}{p(\mathcal{W})p(\mathcal{D}|\mathcal{W})} d\mathcal{W}, \tag{3.2}$$

The right-hand side of Equation 3.2 is referred to as the evidence lower bound (ELBO) loss function $\mathfrak{L}(\phi;\mathcal{D})$, and it is minimized with respect to the variational parameters ϕ when training the Bayesian SVD models by applying the gradient descent update method.

$$\mathfrak{L}(\phi;\mathcal{D}) = -E_{q_\phi(\mathcal{W})}\{\log p(\mathcal{D}|\mathcal{W})\} + \mathrm{KL}\left[q_\phi(\mathcal{W})\|p(\mathcal{W})\right]. \tag{3.3}$$

The two elements in the ELBO loss function, represented by Equation 3.3, are the expected log-likelihood of the training data given the model parameters and a regularization term, which is defined as the KL-divergence between the proposed variational distribution $q_\phi(\mathcal{W})$ and the prior distribution $p(\mathcal{W})$. By marginalizing the model parameters $\mathcal{W}$, we estimate the predictive distribution of newly tokenized source code transformed into an embedding matrix $\hat{\mathbf{X}}$ as follows.

$$p(\hat{\mathbf{y}}|\hat{\mathbf{X}},\mathcal{D}) = \int p(\hat{\mathbf{y}}|\hat{\mathbf{X}},\mathcal{W})\; p(\mathcal{W}|\mathcal{D})\; d\mathcal{W}. \tag{3.4}$$

In the Bayesian SVD structure, we propagate the mean and covariance matrix of the variational distribution, $q_\phi(\mathcal{W})$, and then get the mean and covariance matrix of the predictive distribution, $p(\hat{\mathbf{y}}|\hat{\mathbf{X}},\mathcal{D})$, at the output of the model.

The mean of the predictive distribution indicates the predicted vulnerability class, whereas the covariance matrix represents the uncertainty corresponding to the predicted vulnerability. The aim of the proposed models is to obtain uncertainty information about the output decision to facilitate trustworthy vulnerability prediction in software source codes to provide a secure implementation of sequence machine learning models in practical applications.

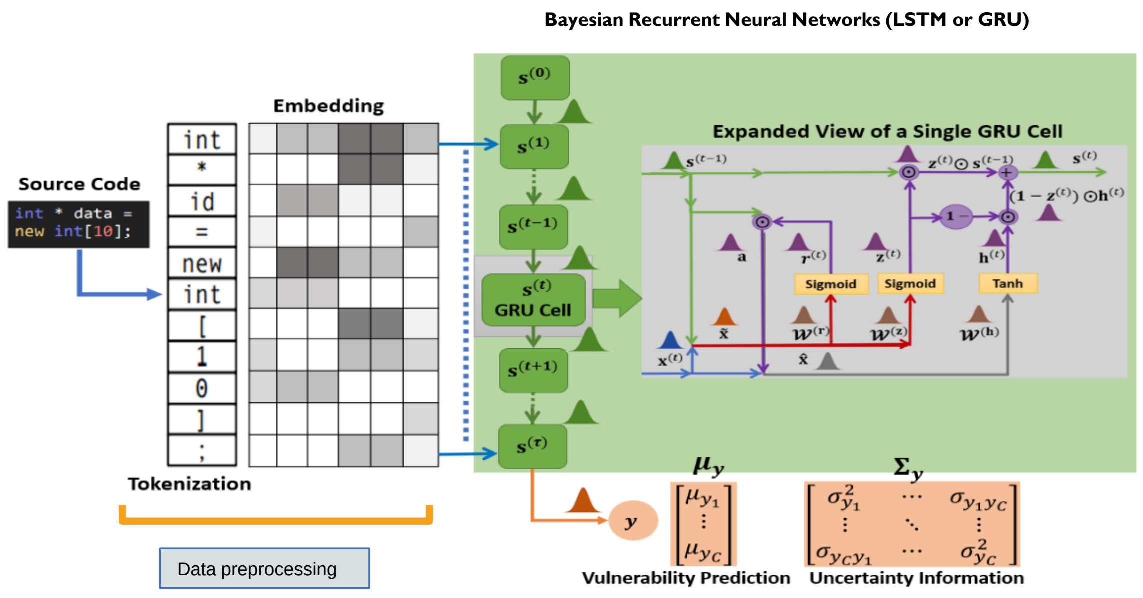


Figure 3.4: Illustration of the proposed software vulnerability detection approach based on Bayesian gated recurrent unit.

3.3 Mathematical Basis of the Software Vulnerability Detection Methods

This section conveys the mathematical basis of the proposed SVD models, which include SVD-LSTM and SVD-GRU as an extension of TRUST models in [20] . In Fig. 3.4, we show the schematic layout when considering a GRU network, i.e., SVD-GRU. We present a mathematical derivation for estimating uncertainty in SVD models for a single hidden state, $\mathbf{s}^{(t)} \in \mathbb{R}^{H\times 1}$, ($t^{\text{th}}$ state), where H is the number of hidden units. A similar derivation can then be applied to all states. The SVD models include several gates in each hidden state: SVD-GRU has three gates and SVD-LSTM has four gates. The gate structure regulates the flow of data from one concealed state to the next.

3.3.1 Software Vulnerability Detection with Gated Recurrent Unit, SVD-GRU

As covered in Chapter 2, the proposed software vulnerability detection SVD-GRU model includes a reset gate, $\mathbf{r}^{(t)}$, an update gate, $\mathbf{z}^{(t)}$, and a candidate hidden state, $\mathbf{h}^{(t)}$. While the update gate controls the information transmitted to the following hidden state, $\mathbf{s}^{(t+1)}$, the reset gate eliminates unimportant information from the prior hidden state, $\mathbf{s}^{(t-1)}$. Consider a row vector

$\mathbf{x}^{(t)} \in \mathbb{R}^{K\times 1}$, from the embedding matrix $\mathbf{X}^{(n)}$ corresponding to the t^{th} token sequence. We combine the preceding hidden state with $\mathbf{x}^{(t)}$, resulting in $\tilde{\mathbf{x}} = \left[\mathbf{x}^{(t)} \;\; \mathbf{s}^{(t-1)}\right]^T$ after transposing them. In addition, we combine the recurrent weight matrix $\mathbf{U}^{(\mathbf{r})} \in \mathbb{R}^{H\times K}$ and the input-hidden weight matrix $\mathbf{W}^{(\mathbf{r})} \in \mathbb{R}^{H\times H}$ to create a single, major weight matrix $\mathscr{W}^{(\mathbf{r})} = \left[\mathbf{U}^{(\mathbf{r})} \;\; \mathbf{W}^{(\mathbf{r})}\right]$. In a similar manner, we combine the weight matrices of the update gate into a single matrix by using the formula $\mathscr{W}^{(\mathbf{z})} = \left[\mathbf{U}^{(\mathbf{z})} \;\; \mathbf{W}^{(\mathbf{z})}\right]$. To formulate the candidate hidden state $\mathbf{h}^{(t)}$, we concatenate the input $\mathbf{x}^{(t)}$ with the element-wise multiplication $\mathbf{s}^{(t-1)} \odot \mathbf{r}^{(t)}$, as in $\hat{\mathbf{x}} = \left[\mathbf{x}^{(t)} \quad \mathbf{s}^{(t-1)} \odot \mathbf{r}^{(t)}\right]^T$ where $\odot$ stands for the element-wise multiplication. Therefore, we can rewrite 2.9, 2.10 and 2.11 for reset gate, update gate, and candidate hidden state as the following:

$$\mathbf{r}^{(t)} = f_s(\mathscr{W}^{(\mathbf{r})}\,\tilde{\mathbf{x}}) \tag{3.5}$$

$$\mathbf{z}^{(t)} = f_s(\mathscr{W}^{(\mathbf{z})}\,\tilde{\mathbf{x}}), \tag{3.6}$$

$$\mathbf{h}^{(t)} = f(\mathscr{W}^{(\mathbf{h})}\,\hat{\mathbf{x}}), \tag{3.7}$$

Here, we have weight matrices $\mathscr{W}^{(\mathbf{r})}$, $\mathscr{W}^{(\mathbf{z})}$ and $\mathscr{W}^{(\mathbf{h})}$ corresponding to the reset gate, update gate and the candidate hidden state, respectively. Additionally, we use sigmoid f and hyperbolic Tangent (Tanh) f_s activation functions. The hidden state $\mathbf{s}^{(t)}$ is then determined as follows:

$$\mathbf{s}^{(t)} = \mathbf{z}^{(t)} \odot \mathbf{s}^{(t-1)} + (1 - \mathbf{z}^{(t)}) \odot \mathbf{h}^{(t)}. \tag{3.8}$$

3.3.2 Software Vulnerability Detection with Long Short-Term Memory, SVD-LSTM

As demonstrated in Chapter 2, the SVD-LSTM model for software vulnerability detection comprises various components to regulate the flow of information and address the vanishing and exploding gradient problems. These components include an input gate $\mathbf{i}^{(t)}$, a forget gate $\mathbf{f}^{(t)}$, an output gate $\mathbf{o}^{(t)}$, a gate gate $\mathbf{g}^{(t)}$, and an additional state known as the cell state or memory cell $\mathbf{c}^{(t)}$. There is another state $\mathbf{s}^{(t)}$ in addition to these. The flow of information within the SVD-LSTM hidden state is collectively governed by the four gates and two states. In particular, the input gate controls the data introduced into the cell state $\mathbf{c}^{(t)}$ after receiving it from the gate gate. The output gate takes the output from the cell state, while the forget gate gets rid of the contents of the cell state. The input and hidden state vectors are merged together in a similar way as the SVD-GRU, $\tilde{\mathbf{x}} = \left[\mathbf{x}^{(t)} \;\; \mathbf{s}^{(t-1)}\right]^T$. Therefore, we can rewrite 2.3, 2.4, 2.5 and 2.6 for input, forget, output, and gate gates as the following:

$$\mathbf{i}^{(t)} = f_s(\mathscr{W}^{(\mathbf{i})}\,\tilde{\mathbf{x}}), \tag{3.9}$$

$$\mathbf{f}^{(t)} = f_s(\mathscr{W}^{(\mathbf{f})}\,\tilde{\mathbf{x}}), \tag{3.10}$$

$$\mathbf{o}^{(t)} = f_s(\mathscr{W}^{(\mathbf{o})}\,\tilde{\mathbf{x}}), \tag{3.11}$$

$$\mathbf{g}^{(t)} = f(\mathscr{W}^{(\mathbf{g})}\,\tilde{\mathbf{x}}), \tag{3.12}$$

Here, we have weight matrices $\mathscr{W}^{(\mathbf{i})}$, $\mathscr{W}^{(\mathbf{f})}$, $\mathscr{W}^{(\mathbf{o})}$ and $\mathscr{W}^{(\mathbf{g})}$ for the input, forget, output and gate gates, respectively. The hidden state $\mathbf{s}^{(t)}$ and the cell state $\mathbf{c}^{(t)}$ are modified as follows:

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \odot \mathbf{g}^{(t)}, \tag{3.13}$$

$$\mathbf{s}^{(t)} = \mathbf{o}^{(t)} \odot f(\mathbf{c}^{(t)}). \tag{3.14}$$

The SVD-GRU and SVD-LSTM models are equipped with gate structures that involve a linear operation (matrix-vector multiplication) followed by a nonlinear activation function (sigmoid or Tanh). By propagating the variational posterior distribution's mean and covariance matrix across a single gate, we demonstrate the theoretical basis for uncertainty estimation. The same mathematical derivation applies to all gates within SVD-GRU and SVD-LSTM. For the reset gate that is defined in the Equation 3.5 for the SVD-GRU model, we can write i^{th} row of the matrix $\mathscr{W}^{(\mathbf{r})}$ as $(\mathbf{w}_i^{\mathbf{r}})^T \in \mathbb{R}^{1\times(K+H)}$, where $i = 1,\ldots,K+H$. The weight vector $\mathbf{w}_i^{\mathbf{r}}$ is assumed to have a Gaussian prior distribution. The variational distribution is then given by $\mathbf{w}_i^{\mathbf{r}} \sim \mathscr{N}(\boldsymbol{\mu}_{\mathbf{w}_i^{\mathbf{r}}}, \boldsymbol{\Sigma}_{\mathbf{w}_i^{\mathbf{r}}})$. We make an assumption that the weight vectors are mutually independent and independent of the input vector

$\tilde{\mathbf{x}}$. This allows us to represent each element of the matrix-vector multiplication in Equation 3.5 as an inner product between two separate random vectors, leading to $\tilde{r}_i = (\mathbf{w}_i^{\mathbf{r}})^T\ \tilde{\mathbf{x}}$. Based on the inner product representation, we can determine the mean and covariance of $\tilde{r}_i$ using the following derivations.

$$\mu_{\tilde{r}_i} = \boldsymbol{\mu}_{\mathbf{w}_i^{\mathbf{r}}}^T\ \boldsymbol{\mu}_{\tilde{\mathbf{x}}}, \tag{3.15}$$
$$\boldsymbol{\Sigma}_{\tilde{\mathbf{r}}} = \begin{cases} \operatorname{tr}\left(\boldsymbol{\Sigma}_{\mathbf{w}_i^{\mathbf{r}}}\ \boldsymbol{\Sigma}_{\tilde{\mathbf{x}}}\right) + \boldsymbol{\mu}_{\mathbf{w}_i^{\mathbf{r}}}^T\ \boldsymbol{\Sigma}_{\tilde{\mathbf{x}}}\ \boldsymbol{\mu}_{\mathbf{w}_j^{\mathbf{r}}} + \boldsymbol{\mu}_{\tilde{\mathbf{x}}}^T\ \boldsymbol{\Sigma}_{\mathbf{w}_i^{\mathbf{r}}}\ \boldsymbol{\mu}_{\tilde{\mathbf{x}}}, & i = j \\ \boldsymbol{\mu}_{\mathbf{w}_i^{\mathbf{r}}}^T\ \boldsymbol{\Sigma}_{\tilde{\mathbf{x}}}\ \boldsymbol{\mu}_{\mathbf{w}_j^{\mathbf{r}}}^T, & i \neq j \end{cases}$$

where $i, j = 1, \cdots, K + H$, and tr refers to the trace operator. When employing SVD-LSTM and SVD-GRU, it is necessary to carry out nonlinear activation, including Tanh or Sigmoid activation in all gates. We adopt first-order Taylor series approximation, enabling us to propagate the mean and covariance of the distribution through these nonlinear functions effectively. By representing the activation functions using the Taylor series, we can calculate the mean and covariance with precision, enhancing the model's performance in handling uncertainty. The mean and covariance at the outputs of the nonlinear activation function f_s in the reset gate can be obtained through the following derivations:

$$\boldsymbol{\mu}_{\mathbf{r}^{(t)}} \approx f_s(\boldsymbol{\mu}_{\tilde{\mathbf{r}}}), \quad \boldsymbol{\Sigma}_{\mathbf{r}^{(t)}} \approx \boldsymbol{\Sigma}_{\tilde{\mathbf{r}}} \odot \left(\nabla f_s(\boldsymbol{\mu}_{\tilde{\mathbf{r}}})\ \nabla f_s(\boldsymbol{\mu}_{\tilde{\mathbf{r}}})^T\right). \tag{3.16}$$

The process of propagating the mean and covariance matrix of the variational posterior distribution through the update gate and candidate hidden state in the SVD-GRU model (Equations 3.6 and 3.7), as well as through the input, forget, output and gate gates in the SVD-LSTM model (Equations 3.9 - 3.12) can be accomplished using the derivations presented in Equations 3.15 and 3.16. Additionally, element-wise multiplication between two random vectors is used in the gate structures of the SVD-GRU and SVD-LSTM models, as seen in 3.8. If $\mathbf{a} = \mathbf{z}^{(t)} \odot \mathbf{s}^{(t-1)}$, we derive the mean and the covariance of **a** as follows, then the mean and covariance of **a** are computed as follows:

$$\begin{aligned}
\boldsymbol{\mu}_{\mathbf{a}} &= \boldsymbol{\mu}_{\mathbf{z}^{(t)}} \odot \boldsymbol{\mu}_{\mathbf{s}^{(t-1)}}, \\
\boldsymbol{\Sigma}_{\mathbf{a}} &= \boldsymbol{\Sigma}_{\mathbf{z}^{(t)}} \odot \boldsymbol{\Sigma}_{\mathbf{s}^{(t-1)}} + \mathbf{diag}(\boldsymbol{\mu}_{\mathbf{s}^{(t-1)}})\, \boldsymbol{\Sigma}_{\mathbf{z}^{(t)}}\, \mathbf{diag}(\boldsymbol{\mu}_{\mathbf{s}^{(t-1)}}) \\
&+ \mathbf{diag}(\boldsymbol{\mu}_{\mathbf{z}^{(t)}})\, \boldsymbol{\Sigma}_{\mathbf{s}^{(t-1)}}\, \mathbf{diag}(\boldsymbol{\mu}_{\mathbf{z}^{(t)}}),
\end{aligned} \tag{3.17}$$

The column vector $\boldsymbol{\mu}_{\mathbf{z}^{(t)}}$ is used to build the diagonal matrix $\mathbf{diag}(\boldsymbol{\mu}_{\mathbf{z}^{(t)}})$, ensuring that the elements of the vector occupy the main diagonal. We can efficiently compute the element-wise multiplications provided by Equations 3.13 and 3.14 in the SVD-LSTM model by using the deduction from Equation 3.17. We can establish the mean and covariance matrix of the concatenation operation between two independent random vectors, that is $\tilde{\mathbf{x}} = \begin{bmatrix}\mathbf{x}^{(t)} & \mathbf{s}^{(t-1)}\end{bmatrix}^T$, as follows:

$$\boldsymbol{\mu}_{\tilde{\mathbf{x}}} = \begin{bmatrix}\boldsymbol{\mu}_{\mathbf{x}^{(t)}} & \boldsymbol{\mu}_{\mathbf{s}^{(t-1)}}\end{bmatrix}^T, \qquad \boldsymbol{\Sigma}_{\tilde{\mathbf{x}}} = \begin{bmatrix}\boldsymbol{\Sigma}_{\mathbf{x}^{(t)}} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Sigma}_{\mathbf{s}^{(t-1)}}\end{bmatrix}. \tag{3.18}$$

There is no correlation between the two terms of the summation in the SVD-LSTM cell state $\mathbf{c}^{(t)}$ in 3.13). We rewrite Equation 3.13 as $\mathbf{c}^{(t)} = \mathbf{c}_1 + \mathbf{c}_2$, with $\mathbf{c}_1 = \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)}$ and $\mathbf{c}_2 = \mathbf{i}^{(t)} \odot \mathbf{g}^{(t)}$. In order to make the mathematical notation more straightforward, we strip away the superscript (t) from $\mathbf{c}_1$ and $\mathbf{c}_2$. Equation 3.17 is used to derive the mean and covariance matrices for $\mathbf{c}_1$ and $\mathbf{c}_2$. The following equations are used to obtain the mean and covariance matrix of the cell state $\mathbf{c}^{(t)}$:

$$\boldsymbol{\mu}_{\mathbf{c}^{(t)}} = \boldsymbol{\mu}_{\mathbf{c}_1} + \boldsymbol{\mu}_{\mathbf{c}_2}, \qquad \boldsymbol{\Sigma}_{\mathbf{c}^{(t)}} = \boldsymbol{\Sigma}_{\mathbf{c}_1} + \boldsymbol{\Sigma}_{\mathbf{c}_2}. \tag{3.19}$$

Equation 3.14 incorporates an element-wise multiplication between two independent random vectors; hence, the mean and covariance matrix of the hidden state of the SVD-LSTM model, $\mathbf{s}^{(t)}$, are also calculated according to Equation 3.17.

The SVD models use a fully-connected layer at the output, $\tilde{\mathbf{y}} = \mathbf{W}^{(y)}\, \mathbf{s}^{(T)}$, where the weight matrix is denoted by $\mathbf{W}^{(y)}$ and we apply a softmax function $f_{\max}$, to get $\hat{\mathbf{y}} = f_{\max}(\tilde{\mathbf{y}})$. The mean and covariance matrix of $\tilde{\mathbf{y}}$ are obtained using Equation (3.15) as the fully-connected layer can also be expressed as the inner product of two independent random variables. The first-order Taylor series approximation is used to derive the mean $\boldsymbol{\mu}_{\hat{\mathbf{y}}}$ and covariance matrix $\boldsymbol{\Sigma}_{\hat{\mathbf{y}}}$ at the output of the softmax function as,

$$\boldsymbol{\mu}_{\hat{\mathbf{y}}} \approx f_{\max}(\boldsymbol{\mu}_{\tilde{\mathbf{y}}}), \qquad \boldsymbol{\Sigma}_{\hat{\mathbf{y}}} \approx \mathbf{J}\boldsymbol{\Sigma}_{\tilde{\mathbf{y}}}\mathbf{J}^T, \tag{3.20}$$

where $\mathbf{J}$ refers to the Jacobian matrix of $\hat{\mathbf{y}}$ with respect to $\tilde{\mathbf{y}}$ evaluated at $\boldsymbol{\mu}_{\tilde{\mathbf{y}}}$.

We rewrite the expected log-likelihood in the ELBO loss function in Equation 3.3 as in Equation (3.21).

$$E_{q_\phi(\mathscr{W})}\{\log p(\mathscr{D}|\mathscr{W})\} \approx -\frac{1}{2N}\sum_{i=1}^{N}\Big[\log(|\boldsymbol{\Sigma}_{\hat{\mathbf{y}}}^{(i)}|) + (\mathbf{y}^{(i)} - \boldsymbol{\mu}_{\hat{\mathbf{y}}}^{(i)})^T(\boldsymbol{\Sigma}_{\hat{\mathbf{y}}}^{(i)})^{-1}(\mathbf{y}^{(i)} - \boldsymbol{\mu}_{\hat{\mathbf{y}}}^{(i)})\Big]. \tag{3.21}$$

The KL-divergence between two multivariate Gaussian distributions, which is the variational posterior distribution and the prior distribution defined over the network parameters, serves as the regularization term in the ELBO loss in the second term in Equation 3.3. There is a closed-form solution for this regularization term in [54]. Using gradient descent-based optimization, we calculate the gradient of the ELBO loss function with respect to the variational parameters ϕ in the back-propagation.

CHAPTER IV

EXPERIMENTAL ANALYSIS AND OUTCOMES

In this chapter, we compare the proposed SVD models with their deterministic equivalents. The Bayesian SVD structure consists of three distinct sections: (1) the source code preprocessing (tokenization and embedding); (2) the gating structure (Bayesian LSTM or GRU); and (3) the output fully connected layer. In the gate structure of the recurrent layers, we employ the sigmoid and hyperbolic tangent (Tanh) activation functions, and at the output of the fully connected layer, we use the softmax function. With a decaying learning rate and polynomial schedule, [3], we train Bayesian SVD models using the Adam optimization algorithm [33]. The Bayesian SVD models and the deterministic LSTM and GRU models are trained and fine-tuned for three different vulnerability detection datasets. The hyperparameters for the optimized models for each dataset are provided in table 4.1. In both deterministic and Bayesian contexts, we employ the identical input size, sequence length, number of layers, and hidden layer node count to maintain consistency. The sequence length is set to 50 for each dataset, while the input size is set at 13 for each dataset. The remaining hyperparameters, such as the number of epochs, batch size, initial and final learning rates, and the KL weighting factor, are provided only for the Bayesian models to give each model its best performance. Table 4.1 lists these hyperparameters for the proposed models.

4.1 Experimental Setup

We conduct three separate experiments or prediction tasks for recognizing vulnerabilities with the goal of evaluating the performance of the suggested Bayesian SVD models.

Table 4.1: Hyperparameter

		VDISC							BigVul	SeVC			
		C1	C2	C3	C4	C5	Combined	Multi-head		PU	FC	AU	AE
	Input Dimension	13	13	13	13	13	13	13	13	13	13	13	13
	time step	50	50	50	50	50	50	50	50	50	50	50	50
	Hidden Nodes	32	32	32	32	32	32	32	32	32	32	48	32
Deterministic	No of epochs	5	5	5	5	5	5	5	10	10	15	30	30
GRU	batch size	12	12	12	12	12	12	12	12	32	32	32	32
	Initial lr	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}
	Ending lr	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-5}	10^{-5}	10^{-5}	10^{-5}
	Input Dimension	13	13	13	13	13	13	13	13	13	13	13	13
	time step	50	50	50	50	50	50	50	50	50	50	50	50
	Hidden Nodes	32	32	32	32	32	32	32	32	32	32	48	32
Deterministic	No of epochs	5	5	5	5	5	5	5	10	10	10	30	30
LSTM	batch size	12	12	12	12	12	12	12	12	32	32	32	12
	Initial lr	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}
	Ending lr	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}
	Input Dimension	13	13	13	13	13	13	13	13	13	13	13	13
	time step	50	50	50	50	50	50	50	50	50	50	50	50
	Hidden Nodes	32	32	32	32	32	32	32	32	32	32	48	32
Bayesian	No of epochs	5	5	5	5	5	5	5	10	30	45	45	50
GRU	batch size	100	100	100	100	100	100	100	100	12	12	12	12
	Initial lr	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-5}	10^{-4}	10^{-4}	2×10^{-3}	2×10^{-3}	2×10^{-3}	2×10^{-3}
	Ending lr	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-6}	10^{-6}	10^{-5}	10^{-4}	2×10^{-4}	2×10^{-4}	10^{-4}
	KL term factor	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-5}	10^{-6}	10^{-5}	10^{-5}
	Input Dimension	13	13	13	13	13	13	13	13	13	13	13	13
	time step	50	50	50	50	50	50	50	50	50	50	50	50
	Hidden Nodes	32	32	32	32	32	32	32	32	32	32	48	32
Bayesian	No of epochs	5	5	5	5	5	5	5	10	15	45	35	25
LSTM	batch size	12	12	12	12	12	12	12	12	12	12	12	12
	Initial lr	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-5}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-3}
	Ending lr	10^{-6}	10^{-6}	10^{-5}	10^{-5}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-5}	10^{-5}	10^{-5}	10^{-4}
	KL term factor	10^{-4}	10^{-4}	10^{-3}	10^{-3}	10^{-4}	10^{-4}	10^{-5}	10^{-4}	10^{-6}	10^{-5}	10^{-5}	10^{-6}

4.1.1 Class-wise prediction

Our strategy involves addressing specific binary classification issues for each unique form of vulnerability in the given situation. This requires determining whether a specific vulnerability exists within each section of the source code. It's worth mentioning that only one element of source code can have numerous vulnerabilities at the same time. Because of this, we carry out the training, validation, and testing procedures separately for every individual type of vulnerability. The model's learning and evaluation are adapted to the particular behaviors and trends connected to each vulnerability category, thanks to this division.

4.1.2 Combined-class prediction

When examining a sequence of source code, it is likely that the code exhibit sensitivity to one or more types of vulnerabilities. To deal with this situation, we collect all potential vulnerabilities and classify source code as vulnerable if it has one or more vulnerabilities. In this approach, the source code is categorized as either vulnerable or immune based on how vulnerable it is to various combinations of vulnerabilities. This enables us to thoroughly evaluate the source code's vulnerability status, taking into account the presence of any or all potential vulnerabilities.

4.1.3 Multi-head class prediction

This experiment redefines the vulnerability discovery task as a multi-head classification problem. A fully connected layer with the softmax function and binary classification of vulnerability is implemented for each head. In order to simultaneously address each of the vulnerabilities, the network divides the model decision under multiple discrete output layers using a single input source code. A single branch of the early SVD layer can learn features from the whole dataset using the multi-head structure, which can then diversify into other vulnerabilities.

4.2 Dataset Selection for Model Development

Three publicly accessible source code datasets, namely the Draper VDISC [58], Big-Vul [21], and Semantics-based Vulnerability Candidate (SeVC) [41] datasets, consisting of labeled C/C++

source code functions, were utilized in our analysis.

4.2.1 VDISC dataset

A total of 1.27 million C/C++ coded functions make up the Draper VDISC dataset. The codes have been obtained from sources including the Debian Linux operating system, the SATE IV Juliet Test Suite, and publicly accessible GitHub repositories. There are 1,019,471 samples in the training set (with 65,904 susceptible codes). The validation and test sets comprise 127,476 and 127,419 samples, respectively. We recovered 1,094,129 tokens which are converted to embedding vectors of size K. The token sequences have a length of (tau). As a result, the embedding matrix has a dimension of $Ktimestau$. The five separate Common Weakness Enumeration (CWE) vulnerability classes are listed in table 4.2 along with the number of vulnerable sequences present in the training, validation, and test sets [2]. The training, validation, and test sets have less number of vulnerable samples, and the dataset is somewhat unbalanced overall. Overall the CWE-120 vulnerability is the most prevalent, but CWE-469 seems to be the least likely source code vulnerability [2]. Additionally, the average frequency of vulnerable sequences across the entire dataset is shown in table 4.2. We test vulnerability detection in the VDISC dataset using all three strategies (class-wise, combined-class, and multi-head class prediction).

Table 4.2: Statistics of the five different types of Common Weakness Enumeration (CWE) vulnerabilities in the training, validation and testing sets of the Draper VDISC dataset with 1.27 million C/C++ functions.

Vulnerable Classes	Associated Flaws	No. vulnerable seq. (training)	No. vulnerable seq. (validation)	No. vulnerable seq. (testing)	Average Frequency
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	19,286	2,419	2,452	1.9%
CWE-120	Classic Buffer Overflow	38,019	4,750	4,891	3.7%
CWE-469	Use of Pointer Subtraction to Determine Size	2,095	252	278	0.2%
CWE-476	NULL Pointer Dereference	9,694	1,208	1,192	0.94%
CWE-other	Buffer Access with Incorrect Length Value, Use of Uninitialized Variable, Improper Input Validation	27,959	3,579	3,490	2.7%

4.2.2 Big-Vul dataset

The Big-Vul dataset contains 3,754 vulnerable source codes dispersed across 91 distinct types of vulnerability, i.e., CWE IDs. These vulnerabilities were tracked down from 348 GitHub projects. The training set contains 150,908 samples with 8,659 vulnerable codes. The validation and test set each includes 37,728 samples. We extracted 236,024 tokens for this dataset. We only run the combined-class prediction experiment due to the large number of distinct kinds of vulnerabilities in this dataset. Optimizing and testing 91 different models or one model with 91 heads would be impractical. Particularly, there aren't enough data examples to train, validate, and test the models for each vulnerability separately.

4.2.3 SeVC dataset

Last but not least, the SeVC dataset includes a set of 126 vulnerabilities taken from the Software Assurance Reference Dataset (SARD) and the National Vulnerability Database (NVD). Based on four separate categories of syntax analysis, including Library/API Function Call (LFC), Array Usage (AU), Pointer Usage (PU), and Arithmetic Expression (AE), the vulnerabilities are categorized. The statistics on SeVC vulnerabilities are provided in table 4.3. Using each of the four vulnerability categories, we compare the proposed SVD models to their deterministic counterparts in order to validate them. Given the huge training dataset compared to the other three vulnerability categories, the PU class has the most tokens extracted, i.e., 86,104. There were 45,961, 31,451, and 38,402 extracted tokens for the library/API function call (LFC), array usage (AU), and arithmetic expression (AE), respectively. Table 4.3 reveals that LFC has the highest proportion of CWEs. PU seems to have the lowest likelihood of a vulnerability association based on average frequency, whereas AU has the highest likelihood. The SeVC dataset is evenly distributed overall. Since we have a separate dataset for each of the four vulnerability syntactic groups, we conduct a separate class-wise prediction experiment for each category.

Table 4.3: Statistics of the four different types of vulnerability syntax in the training, validation and testing sets of the SeVC dataset

Vulnerability Syntax Associate	No. of CWE IDs Associated	No. vulnerable seq. (training)	No. vulnerable seq. (validation)	No. vulnerable seq. (testing)	Average Frequency
Library/API Function Call (LFC)	106	12,214	1,388	1,388	21.5%
Array Usage (AU)	87	9,826	1,099	1,099	26%
Pointer Usage (PU)	103	22,760	5,630	5,630	9.6%
Arithmetic Expression (AE)	45	3,126	349	349	15.75%

4.3 Performance Evaluation

4.3.1 Robustness and Noise Analysis

We evaluate the robustness of the proposed SVD models by comparing their performance to the deterministic models under various noise situations. Each model undergoes initial training based on noise-free source code examples. Then, during testing, we introduce varying levels of Gaussian noise and adversarial attacks during the preprocessing phases. The standard deviation (SD) is a measurement of noise intensity for Gaussian noise. In order to sufficiently introduce random noise to the test data, we alter the SD value and employ three noise levels (Low, Medium, and High). The adversarial examples are constructed using two different approaches, namely the fast gradient sign method (FGSM) and the basic iterative method (BIM) [24,36]. For the FGSM adversarial attack, the noise is synthesized by adding the gradient of the loss function (ELBO loss in the Bayesian SVD models) to the test samples multiplied by ε, i.e., $\varepsilon \text{sign}\left[\nabla_{\mathbf{X}}\mathfrak{L}(\phi;\mathbf{X},\mathbf{y})\right]$, and $\nabla_{\mathbf{X}}$ represents the gradient with respect to the input embedding matrix [24]. For the deterministic GRU and LSTM, the FGSM attacks are generated by utilizing the gradient of the cross-entropy loss function of the deterministic network with respect to the input embedding matrix. With a small step size, α, the BIM is an iterative attack that repeatedly applies FGSM.

For our simulation, the step size, α, is set to 1, and the number of iterations is set to 20. The clipping operation makes sure that the adversarial cases are close to the original data (ε-neighborhood). By changing the ε value, the three levels of adversarial attacks (Low, Medium, and High) for both FGSM and BIM are selected. Along with the SD used to produce different levels of Gaussian noise, the ε values for each level of adversarial attack are given in Table 4.4. The SD and ε values may vary for each dataset based on the quantity of noise.

4.4 Results and Discussion

Tables 4.5, 4.6 and 4.7 show how effectively the proposed Bayesian SVD models performed compared to their deterministic analogs for a range of noise levels using VDISC, Big-Vul, and SeVC datasets. The accuracy of the proposed Bayesian SVD models stays noticeably higher than that of

Table 4.4: The level of Gaussian noise (standard deviation (SD)) and the strength of adversarial attacks (ϵ) applied for all the datasets in the experiments

Bayesian SVD models	Noise Type	Noise Level	VDISC	BigVul	SeVC			
					PU	LFC	AU	AE
Bayesian SVD-GRU	Gaussian	Low	0.05	0.05	0.1	0.001	0.001	0.001
		Medium	0.1	0.1	0.15	0.05	0.05	0.1
		High	0.5	0.5	0.2	0.1	0.1	0.25
	FGSM	Low	0.05	0.05	0.005	0.001	0.001	0.001
		Medium	0.1	0.1	0.01	0.005	0.005	0.005
		High	0.5	0.5	0.05	0.01	0.01	0.01
	BIM	Low	0.05	0.05	0.005	0.001	0.001	0.001
		Medium	0.1	0.1	0.01	0.005	0.005	0.005
		High	0.5	0.5	0.05	0.01	0.01	0.01
Bayesian SVD-LSTM	Gaussian	Low	0.05	0.05	0.05	0.01	0.01	0.1
		Medium	0.1	0.1	0.1	0.05	0.05	0.15
		High	0.5	0.5	0.3	0.3	0.2	0.2
	FGSM	Low	0.05	0.05	0.001	0.001	0.001	0.01
		Medium	0.1	0.1	0.005	0.005	0.005	0.2
		High	0.5	0.5	0.01	0.01	0.01	0.3
	BIM	Low	0.05	0.05	0.001	0.001	0.001	0.01
		Medium	0.1	0.1	0.005	0.005	0.005	0.2
		High	0.5	0.5	0.01	0.01	0.01	0.3

corresponding deterministic models under noisy settings for the VDISC dataset, as shown in Table 4.5. Even when source codes are distorted by random noise and especially adversarial perturbation, the Bayesian SVD models maintain their ability to find vulnerabilities efficiently. This tendency is maintained across all five vulnerability classes (CWE-119, CWE-120, CWE-469, CWE-476, and CWE-other) and all three vulnerability prediction tasks (class-wise prediction, combined-class prediction, and multi-head class prediction).

On the noise-free test data, the proposed Bayesian SVD models possess accuracy identical to the deterministic LSTM and GRU models. However, the accuracy of deterministic models drastically degrades when noise levels increase gradually, especially when adversarial attacks are present. A reasonable level of Gaussian noise appears to have no impact on either model, while a high level of Gaussian noise causes the deterministic model to decline in accuracy substantially. The proposed Bayesian SVD models, however, continue to be accurate even in the presence of very high noise levels. Similar behavior is seen in response to FGSM and BIM attacks, with the accuracy of the deterministic models degrading more rapidly.

The test accuracy of the Big-Vul dataset at different noise levels is displayed in Table 4.6.

Table 4.5: Draper VDISC test accuracy (in %) using SVD-GRU and deterministic GRU for different types of vulnerabilities (C1, C2, C3, C4, C5, Combined and Multiclass representing CWE-119, CWE-120, CWE-469, CWE-476, CWE-other, Combined Classes and Multi Head Classes respectively) under Gaussian noise, and FGSM and BIM adversarial attacks

		Deterministic GRU							Bayesian SVD-GRU						
Noise level		C1	C2	C3	C4	C5	Combined	Multi-head	C1	C2	C3	C4	C5	Combined	Multi-head
No Noise		98.06	96.08	99.78	99.09	97.25	93.47	98.06	98.08	96.16	99.78	99.06	97.26	93.52	98.07
Gaussian	Low	98.06	96.08	99.78	99.09	97.25	93.46	98.06	98.08	96.16	99.78	99.06	97.26	93.52	98.07
	Medium	98.06	96.06	99.78	99.09	97.25	93.45	98.06	98.08	96.16	99.78	99.06	97.26	93.52	98.07
	High	97.85	93.45	99.78	99.08	96.02	87.87	97.62	98.08	96.16	99.78	99.06	97.26	93.52	98.07
FGSM	Low	97.43	92.08	99.79	98.89	94.88	85.18	97.77	98.08	96.16	99.78	99.06	97.26	93.52	98.07
	Medium	91.11	50.64	99.78	95.46	64.94	33.80	92.40	98.08	96.16	99.78	99.06	97.26	93.52	98.07
	High	0.00	0.00	71.57	0.00	0.00	0.00	4.60	98.08	96.16	94.29	99.06	97.26	93.52	98.07
BIM	Low	97.43	92.08	99.78	98.89	94.88	85.18	97.77	98.08	96.16	99.78	99.06	97.26	93.52	98.07
	Medium	91.11	50.64	99.78	95.46	64.94	33.80	92.40	98.08	96.16	99.78	99.06	97.26	93.52	98.07
	High	0.00	0.00	71.57	0.00	0.00	0.00	4.60	98.08	96.16	94.29	99.06	97.26	93.52	98.07

		Deterministic LSTM							Bayesian SVD-LSTM						
Noise level		C1	C2	C3	C4	C5	Combined	Multi-head	C1	C2	C3	C4	C5	Combined	Multi-head
No Noise		98.07	96.16	99.78	99.06	97.26	93.52	98.06	98.08	95.92	99.78	99.06	97.26	93.52	98.07
Gaussian	Low	98.07	96.16	99.78	99.06	97.26	93.52	98.06	98.08	95.93	99.78	99.06	97.26	93.52	98.07
	Medium	98.01	95.89	99.78	99.06	97.26	93.47	98.05	98.08	96.00	99.78	99.06	97.26	93.52	98.07
	High	69.93	71.97	99.71	99.02	96.75	75.46	90.81	98.08	94.06	99.78	99.06	95.83	93.52	98.07
FGSM	Low	17.53	55.71	99.78	99.06	97.09	2.74	79.45	98.08	86.63	99.78	99.06	97.26	93.52	98.07
	Medium	14.73	55.69	99.72	99.00	56.97	0.01	10.58	98.08	50.14	99.78	99.06	97.26	93.52	98.07
	High	1.82	43.24	48.55	74.24	0.00	0.00	5.41	98.08	48.87	99.78	99.06	97.26	42.07	98.06
BIM	Low	17.53	55.70	99.78	99.06	97.09	2.74	79.45	98.08	86.63	99.78	99.06	99.06	99.06	98.07
	Medium	14.73	55.70	99.72	99.00	56.97	0.01	10.58	98.08	50.15	99.78	99.06	97.26	93.52	98.07
	High	1.82	43.24	48.55	74.24	0.00	0.00	5.41	98.08	48.87	99.78	99.06	97.26	42.07	98.06

When no noise is applied to the test data, we observe equivalent accuracy for the proposed Bayesian SVD and deterministic models. The accuracy of the deterministic models, however, declines when Gaussian noise or an adversarial attack is applied with significant levels of noise. With FGSM and BIM adversarial noise, the accuracy of the deterministic models drops dramatically. The accuracy of the Bayesian SVD models, however, is consistently accurate. Therefore, their performance is unaffected by intense noisy scenarios.

Table 4.7 illustrates the test accuracy for each of the four groups of syntactic vulnerabilities in the SeVC dataset. In the noise-free situation, we observe that the proposed models' accuracy is roughly comparable to that of their deterministic equivalents. In contrast, the proposed Bayesian SVD models exhibit higher accuracy when compared to the deterministic models. For PU, LFC,

Table 4.6: Big-Vul test accuracy (in %) using Bayesian and deterministic models under Gaussian noise, and FGSM and BIM adversarial attacks

Noise level		Deterministic GRU	Bayesian SVD-GRU	Deterministic LSTM	Bayesian SVD-LSTM
No Noise		94.22	94.34	94.21	94.36
Gaussian	Low	94.16	94.22	94.26	94.17
	Medium	94.21	94.25	94.29	94.15
	High	94.07	94.21	94.02	94.11
FGSM	Low	93.85	94.12	93.82	94.13
	Medium	93.15	94.34	91.16	94.05
	High	0.00	94.23	61.66	94.39
BIM	Low	94.24	94.17	94.06	94.31
	Medium	93.38	94.08	91.06	94.28
	High	0.00	94.15	62.62	94.40

Table 4.7: SeVC test accuracy (in %) using Bayesian and deterministic models for four different types of vulnerability syntax (PU, LFC, AU, and AE representing Pointer Usage, Library/API Function Call and Arithmetic Expression respectively) under Gaussian noise, and FGSM and BIM adversarial attacks

		Deterministic GRU				Bayesian SVD-GRU				Deterministic LSTM				Bayesian SVD-LSTM			
Noise level		PU	LFC	AU	AE	PU	LFC	AU	AE	PU	LFC	AU	AE	PU	LFC	AU	AE
No Noise		95.65	90.38	90.94	95.70	94.32	92.15	90.79	95.38	95.70	90.58	91.84	95.74	95.96	91.73	91.24	94.66
Gaussian	Low	93.93	90.39	90.86	95.60	93.56	92.12	90.79	95.40	95.17	90.41	91.79	92.39	95.53	91.71	91.03	92.57
	Medium	92.80	90.24	90.72	94.00	92.86	91.70	90.79	94.16	93.62	89.37	89.74	89.36	94.71	89.79	90.31	90.63
	High	91.92	89.61	88.02	91.53	91.61	90.73	88.77	92.53	86.88	72.70	80.99	83.74	91.11	78.50	82.69	88.09
FGSM	Low	89.60	89.65	87.26	95.34	93.59	91.34	90.27	94.52	95.06	88.81	88.76	83.20	95.61	89.72	89.60	89.63
	Medium	82.64	85.81	78.29	93.00	92.71	88.88	87.46	91.85	91.38	78.31	78.41	33.11	93.71	79.74	81.74	83.38
	High	59.86	80.38	62.60	87.00	78.36	86.43	80.65	89.00	85.80	62.00	66.48	17.66	90.80	68.63	72.60	83.33
BIM	Low	89.60	89.65	87.26	95.34	93.59	91.34	90.27	94.52	95.06	88.81	88.76	83.20	95.61	89.72	89.60	89.63
	Medium	82.64	85.81	78.29	93.00	92.71	88.88	87.46	91.80	91.38	78.31	78.41	33.11	93.71	79.74	81.74	83.38
	High	59.86	80.38	62.60	87.00	78.36	86.43	80.65	89.00	85.80	62.00	66.48	17.66	90.80	68.63	72.60	83.33

AU, and AE, the deterministic LSTM model generates accuracy values of 85.8%, 62.0%, 66.48%, and 17.66%, respectively, under the high level of FGSM and BIM attack, as shown in Table 4.7. For PU, LFC, AU, and AE, the Bayesian SVD-LSTM generates 90.8%, 68.63%, 72.6%, and 83.33% accuracy, respectively. In a similar manner, when compared to deterministic models, the Bayesian SVD-GRU maintains superior accuracy despite intense adversarial attacks.

4.5 Self-Assessment through Uncertainty Analysis

This section explores the proposed Bayesian SVD models' uncertainty in the context of noisy environments, such as Gaussian noise, FGSM, and BIM adversarial attacks. The signal-to-noise ratio (SNR) at each noise level is used to quantify the noise level. We compute the predicted variance of the Bayesian SVD models versus SNR (i.e., variance-vs-SNR curve) under noisy conditions for each of the three datasets with various prediction tasks since the predicted variance serves as a measure of uncertainty.

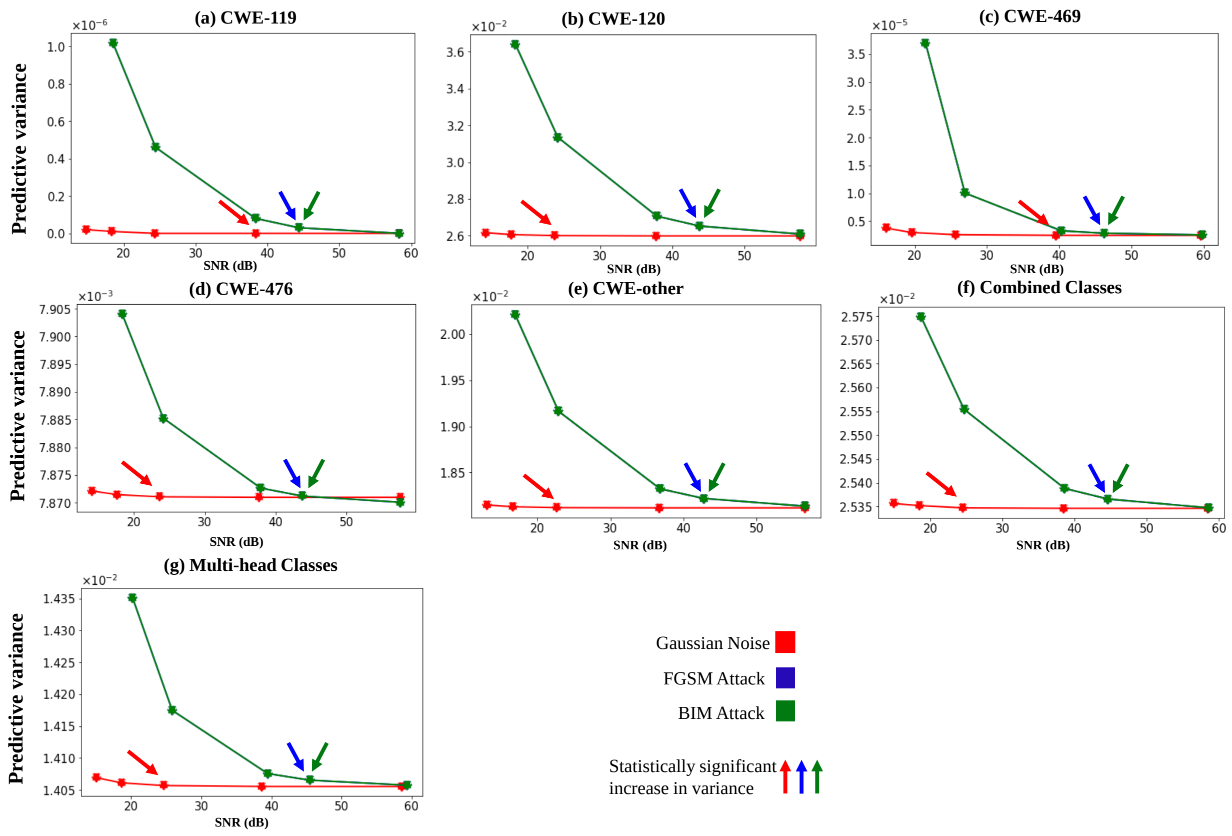


Figure 4.1: Average predictive variance of different classes plotted against SNR under Gaussian noise, FGSM and BIM adversarial attack for Bayesian SVD-GRU for VDISC dataset. The statistical increase in the variance is indicated by arrows pointing to the respective points. A significant increase in the variance can serve as a “red flag” and initiate the process of manual review of the input.

For all classes and all prediction tasks, the findings clearly indicate an increase in predictive variance (the x-axis is read from right to left) with decreasing SNR values. Therefore, the suggested Bayesian SVD models result in increasing uncertainty as the noise level increases (or, equivalently, as SNR drops) and the accuracy of the models degrades. We call this behavior “*self-assessment*" because when the noise level rises significantly, the model evaluates its own performance and identifies its failure mode.

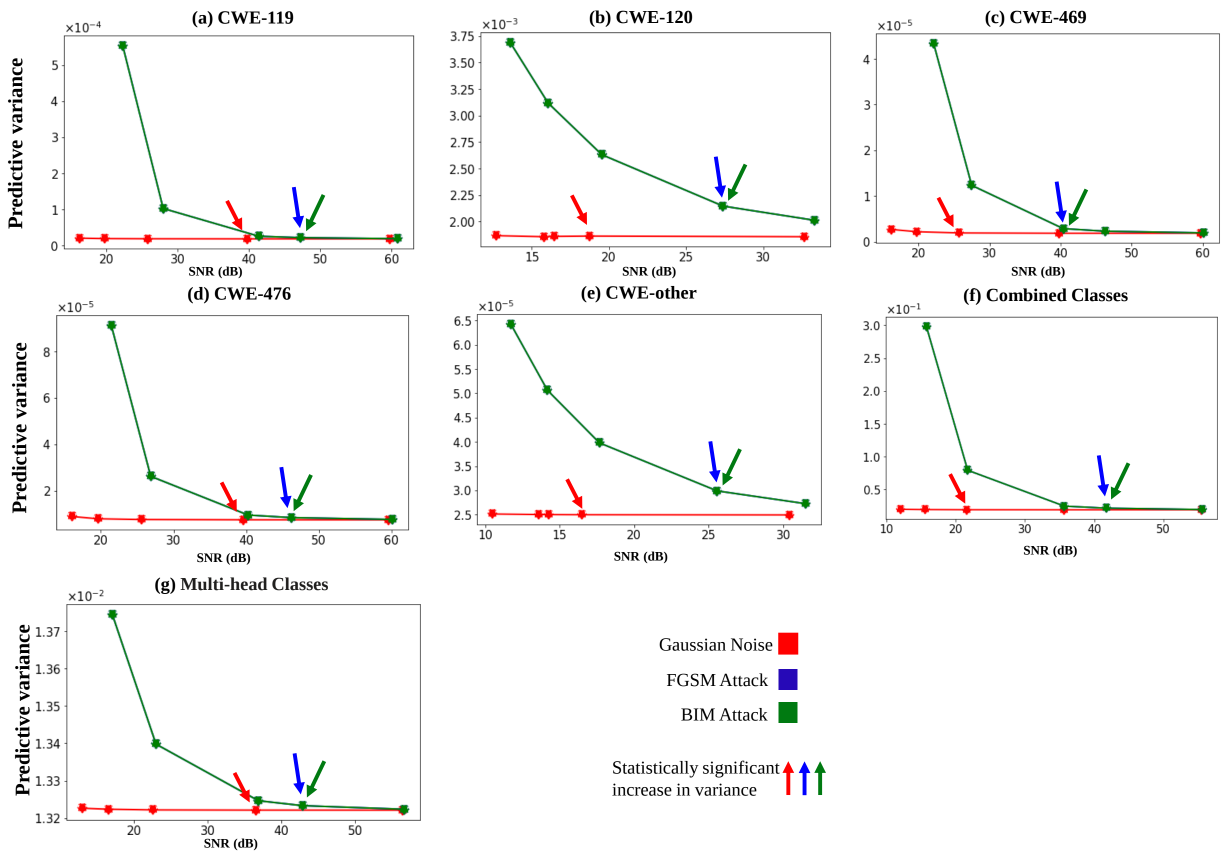


Figure 4.2: Average predictive variance of different classes plotted against SNR under Gaussian noise, FGSM and BIM adversarial attack for Bayesian SVD-LSTM for VDISC dataset. The statistical increase in the variance is indicated by arrows pointing to the respective points. A significant increase in the variance can serve as a "red flag" and initiate the process of manual review of the input.

The Wilcoxon signed-rank test has been utilized to make a pair-wise statistical comparison between the average predictive variance at each noise level and the average variance of the clean test data in order to validate the increase in the predictive variance. In Figures 4.1 and 4.2, an arrow signifies the noise level that causes a substantial increase in the prediction variance ($p < 0.01$). For all vulnerability classes across all tests, we notice the Bayesian SVD models exhibit a substantial increase in variance (or uncertainty). When the system input is distorted by noise or under attack by an adversary, the models' self-evaluation may assist in alerting the user of uncertain decisions. Due to the fact that BIM provides adversarial perturbation by including the gradient of the model output with respect to the input, the variance curves for the FGSM attack (blue curve) and BIM attack (green curve) occasionally overlap. Unlike FGSM, the BIM method generates noise repeatedly. We observe that the increase in model uncertainty under FGSM and BIM adversarial noise appears to be significantly more rapid than that associated with Gaussian noise by observing the uncertainty of the proposed Bayesian SVD models under noisy conditions for all classes. This is based on the fact that adversarial attacks are deliberately designed to harm model performance.

In Figure 4.3, the variance-versus-SNR curves for the Big-Vul dataset are presented. Notably, both Bayesian SVD models, specifically SVD-LSTM (depicted in Figure 4.3 (a)) and SVD-GRU (illustrated in Figure 4.3 (b)), exhibit a noticeable amplification of model uncertainty as the noise level increases. This elevation in uncertainty occurs more rapidly in the context of adversarial attacks compared to the presence of Gaussian noise.

Figures 4.4 and 4.5 demonstrate the noise analysis for the Bayesian SVD-GRU and SVD-LSTM models, respectively. We observe how different kinds of noise affect the model's uncertainty for each of the four vulnerability syntaxes in the SeVC dataset. We detect the same pattern in the model uncertainty as observed in the first two datasets. In contrast, SVD-LSTM exhibits a considerable increase in uncertainty under Gaussian noise as compared to adversarial attacks for the vulnerability associated with Array Usage and Library/API Function Call 4.5 (b and c)). We justify this behavior by the uniqueness of each dataset in the ways in which noise and adversarial attacks can influence trained models.

As a result, the SVD models provide a potential method for discovering vulnerabilities in software source codes, regardless of whether those codes have been compromised by various types of noise or are the target of malicious attacks. These Bayesian SVD models also have the capacity to distinguish between noisy and attacked source codes due to the clear increase in uncertainty that is seen in the predicted variance as the level of noise or the severity of the attack increases.

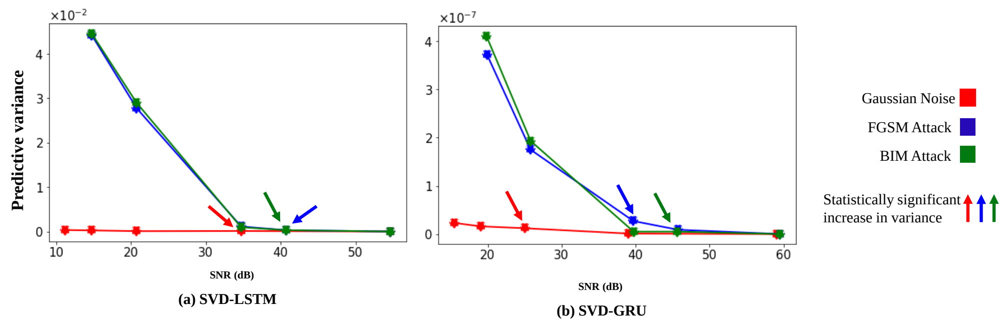


Figure 4.3: Average predictive variance of different classes plotted against SNR under Gaussian noise, FGSM and BIM adversarial attack. The statistical increase in the variance is indicated by arrows pointing to the respective points for (a) Bayesian SVD-LSTM and (b) Bayesian SVD-LSTM for Big-Vul dataset. A significant increase in the variance can serve as a “red flag” and initiate the process of manual review of the input.

Figures 4.1 and 4.2 depict the variance-versus-SNR (Signal-to-Noise Ratio) curves for the Bayesian SVD GRU and LSTM models. These curves illustrate the class-wise predictions, combined-class prediction, and multi-head class prediction using the VDISC dataset. Specifically, Figures 4.1 (a-e) and 4.2 (a-e) represent the class-wise predictions, while Figures 4.1 (f) and 4.2 (f) show the combined-class prediction. Moreover, Figures 4.1 (g) and 4.2 (g) illustrate the multi-head class prediction.

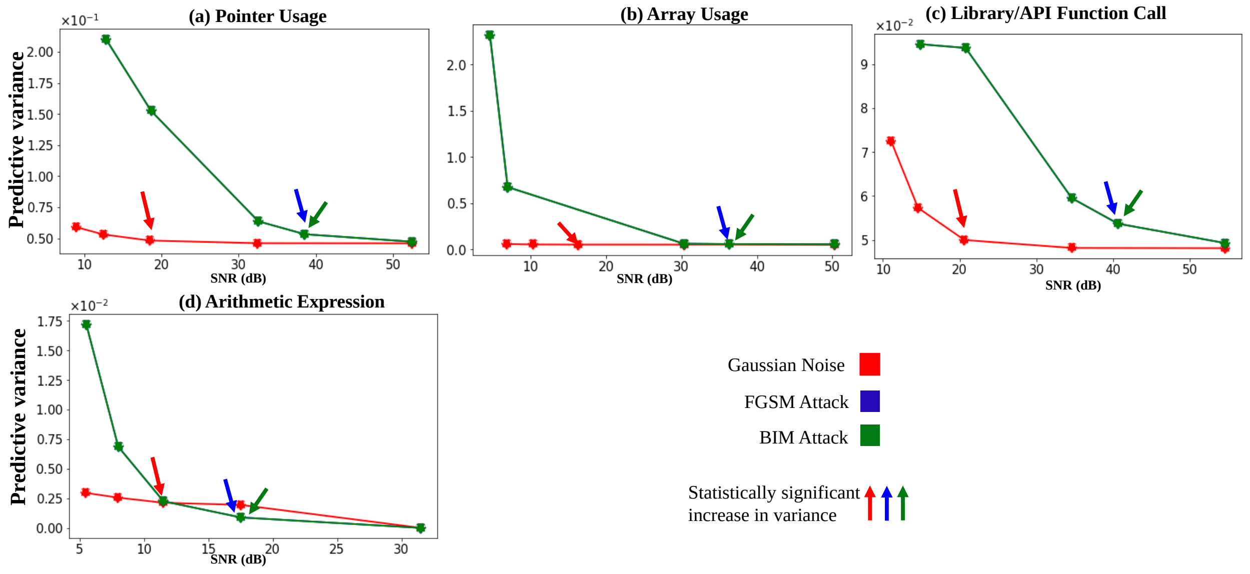


Figure 4.4: Average predictive variance of different classes plotted against SNR under Gaussian noise, FGSM and BIM adversarial attack for Bayesian SVD-GRU for SeVC dataset. The statistical increase in the variance is indicated by arrows pointing to the respective points. A significant increase in the variance can serve as a “red flag” and initiate the process of manual review of the input.

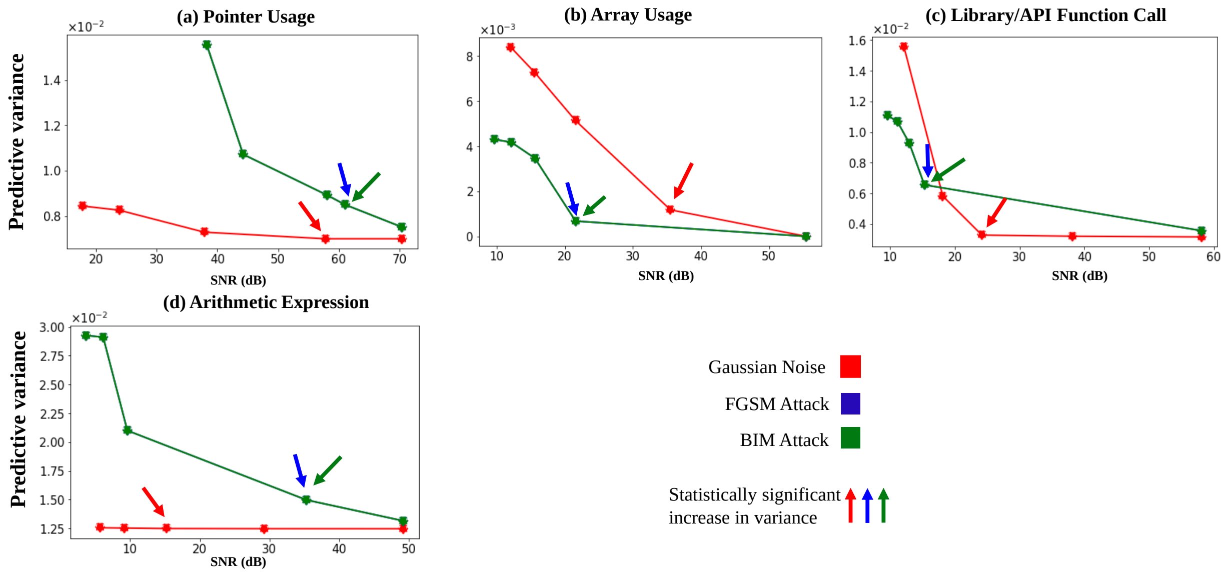


Figure 4.5: Average predictive variance of different classes plotted against SNR under Gaussian noise, FGSM and BIM adversarial attack for Bayesian SVD-LSTM for SeVC dataset. The statistical increase in the variance is indicated by arrows pointing to the respective points. A significant increase in the variance can serve as a “red flag” and initiate the process of manual review of the input.

CHAPTER V

FUTURE WORK

To guarantee thorough coverage and robust security protocols, it is vital to combine manual code reviews, security audits, and intrusion detection with automated vulnerability identification. It provides businesses the capability to spot security flaws and take appropriate action, lowering the chance of exploitation and its related risks. Early in development, companies can prioritize vulnerability identification to avert security incidents, safeguard user data, and foster user and stakeholder confidence. In addition, it assists in complying with regulations, reduces costs by proactively fixing vulnerabilities, and raises the general standard and maintainability of the codebase. Here are a few instances of software vulnerability detection being used in various domains:

- Mobile Applications: Vulnerability scanners and dynamic analysis tools are used in mobile applications to find security flaws particular to mobile platforms. These can include storing data insecurely, using weak authentication methods, communicating between processes insecurely, or using outdated APIs.

- Web Applications: Web application vulnerability scanners may identify frequent cyberattacks like SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF) or insecure direct object references. Uncertain cryptographic techniques, data leaks, or vulnerabilities in server-side scripting languages like PHP, Ruby, or ASP.NET can be found using our vulnerability analysis framework to check the source code or web application binaries.

- Operating Systems: To find flaws like buffer overflows, privilege escalation vulnerabilities, race circumstances, or improper system call usage, operating systems can be examined using vulnerability analysis techniques. Using dynamic analysis tools, vulnerabilities in the run-time environment can be found by simulating attacks and tracking system behavior.

- Embedded systems: Utilizing vulnerability detection technologies, embedded systems like IoT devices, medical devices, or industrial control systems can be tested. Such tools are capable of discovering flaws like faulty authentication, vulnerable communication protocols, buffer overflows, or hazardous default configurations. To find security flaws and vulnerabilities, static analysis tools are capable of reviewing the source code or binaries of embedded systems.

Our source code vulnerability detection algorithm can be customized to serve as a software testing tool or Software as a Service (SaaS) solution. The algorithm can be modified to meet the specifications of a testing tool or SaaS platform to produce a specialized solution for locating and addressing security issues during software testing. This could involve implementing additional vulnerability detection approaches, enhancing the algorithm's rules and strategies, and integrating with the tool's user interface and reporting capabilities. The algorithm can be incorporated into a software testing tool's scanning and analysis sections. It has the ability to automatically scan the source code, run static or dynamic analysis, and find potential security flaws, including unsafe coding practices, poor input validation, or weak authentication. The application can then deliver comprehensive findings, rank vulnerabilities, and support developers and testers in resolving the discovered problems. The modified algorithm can be installed on a cloud-based infrastructure in the case of a SaaS offering. Users can connect their version control repositories to the site or upload their source code. Utilizing the improved technique, the SaaS service scans and analyzes the codebase for vulnerabilities.

Moreover, the source code vulnerability detection framework can be adapted from C/C++ to other programming languages, including PHP, Ruby, JavaScript, Go, Python, and Java. Insecure deserialization in Java, code injection in Python, DOM-based XSS in JavaScript, remote code execution in PHP, mass assignment vulnerabilities in Ruby, and race conditions in Go are just a few examples of language-specific security problems that can be found by adapting the framework to these languages. This requires understanding the existing algorithm, modifying the algorithm's logic, taking language-specific faults into account, and documenting the process. A comprehensive analysis is required to address security threats and programming language changes.

CHAPTER VI

CONCLUSION

We build a novel source code vulnerability detection framework based on the Bayesian gated recurrent unit (SVD-GRU) and Bayesian long short-term memory (SVD-LSTM). The Bayesian inference allows approximation of the variational distribution defined over the model's parameters. At the output of the SVD models, the mean of the predictive distribution provides the estimate of the vulnerability, and the covariance matrix captures the uncertainty in the predicted vulnerability. The proposed SVD models are validated on three datasets containing over one million C/C++ source codes containing different types of CWE vulnerabilities. The experiments have demonstrated superior robustness against Gaussian noise and adversarial attacks compared to the deterministic models. The SVD models show a significant increase in the prediction uncertainty (or predictive variance) under high noise levels or stronger adversarial attacks. The model can use such behavior to assess its own performance and alert the user about performance degradation linked to noise or malicious attacks. This self-assessment mechanism is especially useful in high-stakes applications where accurate and trustworthy predictions are crucial. By continuously monitoring their own levels of uncertainty, these models can timely detect any performance degradation resulting from excessive noise or malicious attacks. This feature allows them to notify users of potential dangers or compromised data. Overall, the ability of the Bayesian SVD models to evaluate their performance under adverse conditions makes them appropriate for applications where noise or adversarial attacks are prevalent, establishing confidence in their predictions and empowering users to make informed decisions despite the presence of interruption factors.

REFERENCES

[1] *CVSS Severity Distribution Over Time*, Jan 2022.

[2] *CWE Common Weakness Enumeration*, 2022. Accessed: March, 2022.

[3] M. Abadi, , et al., *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. Software available from tensorflow.org.

[4] A. M. Algarni and Y. K. Malaiya, *Software vulnerability markets: Discoverers and buyers*, International Journal of Computer and Information Engineering, 8 (2014), pp. 480 – 490.

[5] O. H. Alhazmi and Y. K. Malaiya, *Application of vulnerability discovery models to major operating systems*, IEEE Transactions on Reliability, 57 (2008), pp. 14–22.

[6] A. Arora and R. Telang, *Economics of software vulnerability disclosure*, IEEE security & privacy, 3 (2005), pp. 20–25.

[7] A. Atla, R. Tada, V. Sheng, and N. Singireddy, *Sensitivity of different machine learning algorithms to noise*, J. Comput. Sci. Coll., 26 (2011), p. 96–103.

[8] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, *Static code analysis to detect software security vulnerabilities-does experience matter?*, in 2009 International Conference on Availability, Reliability and Security, IEEE, 2009, pp. 804–810.

[9] X. Ban, S. Liu, C. Chen, and C. Chua, *A performance evaluation of deep-learnt features for software vulnerability detection*, Concurrency and Computation: Practice and Experience, 31 (2019), p. e5103.

[10] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, *Dynamic analysis of malicious code*, Journal in Computer Virology, 2 (2006), pp. 67–77.

[11] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay, *Vulnerability prediction from source code using machine learning*, IEEE Access, 8 (2020), pp. 150672–150684.

[12] J. Cai, P. Zou, J. Ma, and J. He, *Sworddta: A dynamic taint analysis tool for software vulnerability detection*, Wuhan University Journal of Natural Sciences, 21 (2016), pp. 10–20.

[13] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, *Deep learning based vulnerability detection: Are we there yet*, IEEE Transactions on Software Engineering, (2021).

[14] S. E. CHANDY, A. RASEKH, Z. A. BARKER, AND M. E. SHAFIEE, *Cyberattack detection using deep generative models with variational inference*, Journal of Water Resources Planning and Management, 145 (2019), p. 04018093.

[15] T. CHEN, E. FOX, AND C. GUESTRIN, *Stochastic gradient hamiltonian monte carlo*, in International conference on machine learning, PMLR, 2014, pp. 1683–1691.

[16] J.-T. CHIEN AND Y.-C. KU, *Bayesian recurrent neural network for language modeling*, IEEE transactions on neural networks and learning systems, 27 (2015), pp. 361–374.

[17] K. CHO, B. VAN MERRIENBOER, D. BAHDANAU, AND Y. BENGIO, *On the properties of neural machine translation: Encoder-decoder approaches*, in Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST-8), 2014.

[18] A. DAIRI, F. HARROU, AND Y. SUN, *A deep attention-driven model to forecast solar irradiance*, in 2021 IEEE 19th International Conference on Industrial Informatics (INDIN), IEEE, 2021, pp. 1–6.

[19] H. DAM, T. TRAN, T. PHAM, S. NG, J. GRUNDY, AND A. GHOSE, *Automatic feature learning for predicting vulnerable software components*, IEEE Transactions on Software Engineering, 47 (2021), pp. 67–85.

[20] D. DERA, S. AHMED, N. C. BOUAYNAYA, AND G. RASOOL, *Trustworthy uncertainty propagation for sequential time-series analysis in rnns*, IEEE Transactions on Knowledge and Data Engineering, (2023), pp. 1–13.

[21] J. FAN, Y. LI, S. WANG, AND T. N. NGUYEN, *Ac/c++ code vulnerability dataset with code changes and cve summaries*, in Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 508–512.

[22] Y. GAL AND Z. GHAHRAMANI, *A theoretically grounded application of dropout in recurrent neural networks*, Advances in neural information processing systems, 29 (2016).

[23] S. GHOSH, F. DELLE FAVE, AND J. YEDIDIA, *Assumed density filtering methods for learning bayesian neural networks*, in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 30, 2016.

[24] I. J. GOODFELLOW, J. SHLENS, AND C. SZEGEDY, *Explaining and harnessing adversarial examples*, in Proceedings of 3rd International Conference on Learning Representations, (ICLR), 2015.

[25] Z. HAN, X. LI, Z. XING, H. LIU, AND Z. FENG, *Learning to predict severity of software vulnerability using only vulnerability description*, in 2017 IEEE International conference on software maintenance and evolution (ICSME), IEEE, 2017, pp. 125–136.

[26] J. HARER, O. OZDEMIR, T. LAZOVICH, C. REALE, R. RUSSELL, L. KIM, ET AL., *Learning to repair software vulnerabilities with generative adversarial networks*, in Proceedings of the 31st International Conference on Neural Information Processing Systems, (NIPS), vol. 31, 2018.

[27] B. HARRISON, C. PURDY, AND M. RIEDL, *Toward automated story generation with markov chain monte carlo methods and deep neural networks*, in Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, vol. 13, 2017, pp. 191–197.

[28] J. M. HERNÁNDEZ-LOBATO AND R. ADAMS, *Probabilistic backpropagation for scalable learning of bayesian neural networks*, in International conference on machine learning, PMLR, 2015, pp. 1861–1869.

[29] S. HOCHREITER, *The vanishing gradient problem during learning recurrent neural nets and problem solutions*, International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 6 (1998), pp. 107–116.

[30] A. HOVSEPYAN, R. SCANDARIATO, W. JOOSEN, AND J. WALDEN, *Software vulnerability prediction using text analysis techniques*, in Proceedings of the 4th international workshop on Security measurements and metrics, 2012, pp. 7–10.

[31] S. KAMIYA, J.-K. KANG, J. KIM, A. MILIDONIS, AND R. M. STULZ, *Risk management, firm reputation, and the impact of successful cyberattacks on target firms*, Journal of Financial Economics, 139 (2021), pp. 719–749.

[32] S. KIM, S. WOO, H. LEE, AND H. OH, *Vuddy: A scalable approach for vulnerable code clone discovery*, in IEEE Symposium on Security and Privacy (SP), 2017, pp. 595–614.

[33] D. P. KINGMA AND J. BA, *Adam: A method for stochastic optimization*, in Proceedings of 3th International Conference on Learning Representations, (ICLR), 2015.

[34] G. KLEES, A. RUEF, B. COOPER, S. WEI, AND M. HICKS, *Evaluating fuzz testing*, in Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, 2018, pp. 2123–2138.

[35] M. KOMISAREK, M. PAWLICKI, R. KOZIK, AND M. CHORAS, *Machine learning based approach to anomaly and cyberattack detection in streamed network traffic data.*, J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl., 12 (2021), pp. 3–19.

[36] A. KURAKIN, I. GOODFELLOW, AND S. BENGIO, *Adversarial examples in the physical world*, in Proceedings of 5th International Conference on Learning Representations, (ICLR) workshop track, 2017.

[37] D. LAST, *Using historical software vulnerability data to forecast future vulnerabilities*, in 2015 Resilience Week (RWS), IEEE, 2015, pp. 1–7.

[38] A. LAZARENKO AND S. AVDOSHIN, *Financial risks of the blockchain industry: A survey of cyberattacks*, in Proceedings of the Future Technologies Conference (FTC) 2018: Volume 2, Springer, 2019, pp. 368–384.

[39] X. LI, L. WANG, Y. XIN, Y. YANG, Q. TANG, AND Y. CHEN, *Automated software vulnerability detection based on hybrid neural network*, Applied Sciences, 11 (2021), p. 3201.

[40] Y. Li, J. M. Hernández-Lobato, and R. E. Turner, *Stochastic expectation propagation*, Advances in neural information processing systems, 28 (2015).

[41] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, *Sysevr: A framework for using deep learning to detect software vulnerabilities*, IEEE Transactions on Dependable and Secure Computing, 19 (2021), pp. 2244–2258.

[42] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, *Software vulnerability detection using deep neural networks: A survey*, Proceedings of the IEEE, 108 (2020), pp. 1825–1848.

[43] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, *TBar: revisiting template-based automated program repair*, in Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019.

[44] L. Liu and S. Vasudevan, *Scaling input stimulus generation through hybrid static and dynamic analysis of rtl*, ACM Transactions on Design Automation of Electronic Systems (TODAES), 20 (2014), pp. 1–33.

[45] P. Louridas, *Static code analysis*, Ieee Software, 23 (2006), pp. 58–61.

[46] G. Lykou, D. Moustakas, and D. Gritzalis, *Defending airports from uas: A survey on cyber-attacks and counter-drone sensing technologies*, Sensors, 20 (2020), p. 3537.

[47] D. J. MacKay, *A practical bayesian framework for backpropagation networks*, Neural computation, 4 (1992), pp. 448–472.

[48] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere, *Hybrid static-dynamic attacks against software protection mechanisms*, in Proceedings of the 5th ACM workshop on Digital rights management, 2005, pp. 75–82.

[49] T. Nagapetyan, A. B. Duncan, L. Hasenclever, S. J. Vollmer, L. Szpruch, and K. Zygalakis, *The true cost of stochastic gradient langevin dynamics*, arXiv preprint arXiv:1706.02692, (2017).

[50] R. M. Neal, *Bayesian learning for neural networks*, vol. 118, Springer Science & Business Media, 2012.

[51] J. Novak, A. Krajnc, et al., *Taxonomy of static code analysis tools*, in The 33rd international convention MIPRO, IEEE, 2010, pp. 418–422.

[52] A. Pechenkin and R. Demidov, *Application of deep neural networks for security analysis of digital infrastructure components*, in SHS Web of Conferences, vol. 44, EDP Sciences, 2018, p. 00068.

[53] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, *VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits*, in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 426–437.

[54] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*, The MIT Press, 2005.

[55] M. Rezaee and F. Ferraro, *A discrete variational recurrent topic model without the reparametrization trick*, Advances in neural information processing systems, 33 (2020), pp. 13831–13843.

[56] M. Riek and R. Böhme, *The costs of consumer-facing cybercrime: an empirical exploration of measurement issues and estimates†*, Journal of Cybersecurity, 4 (2018).

[57] H. Ritter, A. Botev, and D. Barber, *A scalable laplace approximation for neural networks*, in 6th International Conference on Learning Representations, ICLR 2018-Conference Track Proceedings, vol. 6, International Conference on Representation Learning, 2018.

[58] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, *Automated vulnerability detection in source code using deep representation learning*, in Proceedings of the 17th IEEE international conference on machine learning and applications (ICMLA), 2018, pp. 757–762.

[59] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, *Predicting vulnerable software components via text mining*, IEEE Transactions on Software Engineering, 40 (2014), pp. 993–1006.

[60] R. M. Schmidt, *Recurrent neural networks (rnns): A gentle introduction and overview*, arXiv preprint arXiv:1912.05911, (2019).

[61] G. Suciu, A. Scheianu, A. Vulpe, I. Petre, and V. Suciu, *Cyber-attacks–the impact over airports security and prevention modalities*, in Trends and Advances in Information Systems and Technologies: Volume 3 6, Springer, 2018, pp. 154–162.

[62] G. Tang, L. Meng, H. Wang, S. Ren, Q. Wang, L. Yang, and W. Cao, *A comparative study of neural network techniques for automatic software vulnerability detection*, in 2020 International Symposium on Theoretical Aspects of Software Engineering (TASE), IEEE, 2020, pp. 1–8.

[63] R. Telang and S. Wattal, *An empirical analysis of the impact of software vulnerability announcements on firm stock price*, IEEE Transactions on Software engineering, 33 (2007), pp. 544–557.

[64] M. M. Tikir and J. K. Hollingsworth, *Efficient instrumentation for code coverage testing*, ACM SIGSOFT Software Engineering Notes, 27 (2002), pp. 86–96.

[65] P. Tsankov, M. T. Dashti, and D. Basin, *Secfuzz: Fuzz-testing security protocols*, in 2012 7th International Workshop on Automation of Software Test (AST), IEEE, 2012, pp. 1–7.

[66] G. Van Houdt, C. Mosquera, and G. Nápoles, *A review on the long short-term memory model*, Artificial Intelligence Review, 53 (2020), pp. 5929–5955.

[67] J. Wan, S. Tang, Z. Shu, D. Li, S. Wang, M. Imran, and A. V. Vasilakos, *Software-defined industrial internet of things in the context of industry 4.0*, IEEE Sensors Journal, 16 (2016), pp. 7373–7380.

[68] B. Wang, J. Lu, Z. Yan, H. Luo, T. Li, Y. Zheng, and G. Zhang, *Deep uncertainty quantification: A machine learning approach for weather forecasting*, in Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2019, pp. 2087–2095.

[69] Q. Wang, Y. Li, Y. Wang, and J. Ren, *An automatic algorithm for software vulnerability classification based on cnn and gru*, Multimedia Tools and Applications, (2022), pp. 1–22.

[70] M. Welling and Y. W. Teh, *Bayesian learning via stochastic gradient langevin dynamics*, in Proceedings of the 28th international conference on machine learning (ICML-11), 2011, pp. 681–688.

[71] T. Xiao, J. Guan, S. Jian, Y. Ren, J. Zhang, and B. Li, *Software vulnerability detection method based on code property graph and bi-gru*, Journal of Computer Research and Development, 58 (2021), p. 1668.

[72] Z. Xu, T. Kremenek, and J. Zhang, *A memory model for static analysis of c programs*, in Leveraging Applications of Formal Methods, Verification, and Validation, T. Margaria and B. Steffen, eds., Springer Berlin Heidelberg, 2010, pp. 535–548.

[73] S. Yang, X. Yu, and Y. Zhou, *Lstm and gru neural network performance comparison study: Taking yelp review dataset as an example*, in 2020 International workshop on electronic communication and artificial intelligence (IWECAI), IEEE, 2020, pp. 98–101.

[74] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into deep learning*, arXiv preprint arXiv:2106.11342, (2021).

[75] Y. Zhang, M. Qiu, C.-W. Tsai, M. M. Hassan, and A. Alamri, *Health-cps: Healthcare cyber-physical system assisted by cloud and big data*, IEEE Systems Journal, 11 (2017), pp. 88–95.

[76] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, *VulDeePecker: A deep learning-based system for multiclass vulnerability detection*, IEEE Transactions on Dependable and Secure Computing, 18 (2021), pp. 2224–2236.

BIOGRAPHICAL SKETCH

Orune Aminul is a dedicated and passionate individual who has overcome numerous challenges to pursue a career in engineering and data science. Hailing from a low-developed county like Bangladesh, she defied societal expectations and nurtured her fervor for science and engineering.

Her journey led her to earn a bachelor's degree in Electrical and Electronics Engineering in 2019. She further pursued her academic aspirations by enrolling in a Master's program at the University of Texas Rio Grande Valley and earned her degree in August 2023. She was honored with the prestigious Presidential Research Fellowship award. During her academic tenure, she demonstrated exceptional skills and knowledge in research, leading to her acceptance into the esteemed University of Texas Rio Grande Valley.

Under the guidance of Dr. Dimah Dera, an accomplished researcher in machine learning and data science, she was engaged in pursuing her Master's thesis. Her research revolves around the innovation of Bayesian deep learning networks for sequential datasets, with practical applications spanning healthcare, industry, cybersecurity, and optimization. As she delves into her work, she envisions the transformative potential of data science in various sectors, from finance to national security.

Orune Aminul's aspiration to contribute to the data-driven world drives her to play a pivotal role in advancing the field of artificial intelligence and data science. With a firm belief in the power of knowledge and innovation, she aims to leave a lasting impact on the intersection of technology and society.

For inquiries or further communication, Orune Aminul can be reached via email at orune.aminul@gmail.com.