MASTER'S THESIS

Explaining and improving vulnerability detection Using Layer-wise Relevance Propagation

Foeken, H.

Award date:
2022

Link to publication

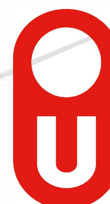
General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain.
- You may freely distribute the URL identifying the publication in the public portal.

Take down policy
If you believe that this document breaches copyright please contact us at:
pure-support@ou.nl
providing details and we will investigate your claim.

Downloaded from https://research.ou.nl/ on date: 02. Jul. 2022

Open Universiteit
www.ou.nl



Explaining and improving vulnerability detection

Using Layer-wise Relevance Propagation

H. Foeken

Student: Date: 07/04/2022

Open Universiteit
www.ou.nl

Explaining and improving vulnerability detection

Using Layer-wise Relevance Propagation

by

H. Foeken

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University of the Netherlands, Faculty of Science,
Master's Programme in Software Engineering
to be defended publicly on 7 April, 2022 at 3 PM.

Student number:		
Course code:	IM9906	
Thesis committee:	dr. ir. H.P.E. Vranken (supervisor),	Open University
	dr. A.J. Hommersom (supervisor),	Open University

Open Universiteit
de beste! www.ou.nl

Acknowledgements

I want to express my sincere gratitude to the Leiden University Medical Hospital for supporting my academic ambitions alongside my work and family. Also, I want to offer my special thanks to my supervisors Arjen Hommersom en Harald Vranken for their insightful comments and suggestions in our meetings which improved the quality of this thesis.

Furthermore, I want to thank Evert Verduijn for reviewing my draft thesis and the persistent motivational talks. I am also grateful to Daniel Ostkamp for reading my draft thesis and providing me with valueable feedback.

On a more personal note, I want to extend my sincere thanks to my friends Wouter van Drecht and Wybo Houkes for our discussions regarding my thesis which motivated me to continue working.

Last but not least, Renate, words cannot express how grateful I am for the time you gave me to complete this thesis, for taking care of our family, and for the support in difficult times. I could not have done it without you.

Hylke Foeken

Contents

Acknowledgements i

Summary vii

Samenvatting viii

1 Introduction 1
- 1.1 Software security . . . 1
- 1.2 Software vulnerabilities . . . 1
- 1.3 Software vulnerability detection . . . 1
- 1.4 Goal . . . 3
- 1.5 Overview report structure . . . 3

2 Background 5
- 2.1 Machine learning . . . 5
- 2.2 Deep learning . . . 5
- 2.3 Vulnerability detection using deep learning . . . 9
- 2.4 Explaining classifications . . . 12

3 Research design 14
- 3.1 Research motivation . . . 14
- 3.2 Research objective . . . 15
- 3.3 Research questions . . . 15
- 3.4 Research method . . . 16
- 3.5 SySeVR metrics. . . . 18
- 3.6 Intersection over Union . . . 20

4 Model and dataset preparation 21
- 4.1 Model preparation . . . 21
  - 4.1.1 SySeVR vulnerability detection . . . 21
  - 4.1.2 SySeVR dataset and implementation . . . 22
  - 4.1.3 Deep learning model creation . . . 22
- 4.2 Vulnerabilities dataset . . . 25
  - 4.2.1 Sample origins. . . . 25
  - 4.2.2 Vulnerable line labels. . . . 27
  - 4.2.3 Dataset validation. . . . 27
- 4.3 Model baseline performance . . . 28

5 Explaining software vulnerability classifications 30
- 5.1 Explaining classifications . . . 30
  - 5.1.1 Layer-wise relevance propagation . . . 30
  - 5.1.2 Visualizing explanations . . . 32

5.2 Explaining software vulnerability classifications 32
5.2.1 LRP implementation 33
5.2.2 Line comparison statistic 34
5.2.3 Vulnerable line detection precision 35
5.2.4 Vulnerability localization precision 37
5.3 Conclusion 38

6 Improving the SySeVR vulnerability detection system **39**
6.1 Introduction 39
6.2 Improve the classification performance during inference 39
6.2.1 Remove tokens with high relevance to prevent FP classifications 39
6.2.2 Remove tokens with low relevance to decrease FN classifications 41
6.2.3 Combining FP/FN improvements 44
6.3 Improve classification performance during training 47
6.4 Conclusion 48

7 Related work **50**

8 Discussion **52**
8.1 Explaining software vulnerability classifications 52
8.2 Improving the SySeVR vulnerability detection system 54
8.3 Limitations 56

9 Conclusion & recommendations **58**

References **61**
Literature 61
Web links 64
Books 64
Theses 65

A IoU data **66**

B Token filters **68**

C Filter results **71**

List of Figures

1.1 NVD report, published vulnerabilities per year . . . 2
1.2 Vulnerabilities per type by CVE details . . . 2
1.3 Explanation of image classification . . . 3

2.1 Deep learning concept learning . . . 6
2.2 Simplified neural network example . . . 7
2.3 Recurrent neural network example . . . 8
2.4 Convolution operation in a CNN . . . 9
2.5 Patterns learned by CNN from Krizhevsky et al. . . . 9
2.6 Granularity of SySeVR sample . . . 10
2.7 Russel et al. neural network architecture . . . 11
2.8 SySeVR neural network architecture . . . 11

3.1 Dataset preparation overview . . . 17
3.2 Research method RQ1 . . . 18
3.3 Research method RQ2 and RQ3 . . . 19

4.1 Region proposals in software vulnerability detection . . . 22
4.2 SARD testcase and SySeVR sample . . . 23
4.3 Architectures of SySeVR and Arras et al. . . . 24
4.4 SARD metadata example . . . 26
4.5 SySeVR dataset sample kind counts . . . 26
4.6 Vulnerabilities dataset summary . . . 27
4.7 Dataset split counts . . . 28
4.8 Validation of SySeVR and our model . . . 29

5.1 Example deep neural network . . . 31
5.2 Example classification . . . 31
5.3 Example LRP explanation result . . . 31
5.4 Sample explanation . . . 33
5.5 Mean, maximum and median line statistics for thesis example . . . 34
5.6 Mean, maximum, and median selection criteria . . . 34
5.7 Precision-recall curves line precision . . . 36
5.8 Average localization precision . . . 37
5.9 Histogram localization precision . . . 38

6.1 Lowering FP . . . 40
6.2 LRP FP classification metrics . . . 42
6.3 LRP FN classification metrics . . . 43
6.4 LRP FP-FN classification metrics . . . 45
6.5 Lowering incorrect classifications . . . 47

6.6 LRP model classification metrics 49

8.1 Loss comparison 53

A.1 Localization precision histogram bins 66
A.2 Vulnerability localization precision counts 67

B.1 Top 50 unique tokens 69
B.2 Bottom 50 unique tokens 70

C.1 Classification results and metrics 71

List of Tables

3.1 SySeVR performance metrics 19
3.2 IoU metric sample 20

4.1 Comparision of SySeVR, initial, and final model 25

5.1 Top 5 localization precision histogram bins 38

6.1 LRP-analysis confusion matrix 40
6.2 LRP FP classification results 41
6.3 LRP FN classification results 42
6.4 Token count FP-FN filter-sets 44
6.5 LRP FP-FN classification results 44
6.6 LRP sorted classification metrics 46
6.7 LRP model classification results 48

8.1 Experiment results 54
8.2 Model training results 55

SUMMARY

Software security engineering is concerned with creating software that operates as required in a potentially hostile environment. Creating secure software is important because it is ubiquitously used and enables interactions between humans, governments, and companies. This engineering practice is essential to avoid and fix software defects related to security (i.e., software vulnerabilities). Towards this goal, methods exist to detect software vulnerabilities using deep learning.

Our research aims to improve such a vulnerability detection system by using explanations of its incorrect classifications. To reach this goal, we measured how precisely the layer-wise relevance propagation (LRP) method could detect vulnerable lines across 51,586 vulnerable samples and how accurately LRP located the vulnerable lines in these samples. Using the insights gained from these measurements, we performed experiments to improve the classification performance of our example vulnerability detection system by filtering tokens found in incorrect classifications before and after fitting the model.

We discovered that the LRP method is more than twice as good at detecting vulnerable lines in our dataset than a random guess. In spite of this improvement there are still many vulnerable lines which are not detected using our approach and this shows that our application of the LRP method is not well suited for this purpose. Furthermore, we measured how accurate the LRP method can detect vulnerable lines in a sample. We determined that, on average, a significant part of the vulnerable lines in a sample is not selected by the LRP method. Because the majority of our vulnerable samples has a single vulnerable line, our application of the LRP method will rarely locate it which limits its practical use in explaining these samples.

The results of our experiments show that the number of correct classifications and the average precision decreased by removing the most relevant parts from our samples after fitting the model. Conversely, removing them before fitting the model did not change the average precision significantly but did decrease the number of false positive classifications at the expense of an increase in the number of false negative classifications.

Our analysis of explanations improves our understanding of deep learning approaches in the context of software vulnerability detection. We show that the lines we labeled as being vulnerable do not play a significant role in the classifications of our model. This insight into the behavior of our model was unexpected and lowered our trust in its ability to detect vulnerabilities when applied in a different setting. Our adjustments to the model training procedure yielded a model in which, at the same classification threshold, a larger proportion of detected vulnerabilities are relevant. However, they also resulted in a model that selects a smaller proportion of the vulnerabilities. An improvement would require higher proportions on both terms and therefore, we did not improve our model's performance but rather shifted its focus in this classic trade-off in vulnerability detection systems.

SAMENVATTING

Software security engineering houdt zich bezig met het maken van software die werkt zoals verwacht in een potentieel vijandige omgeving. Dit is belangrijk omdat software alomtegenwoordig wordt gebruikt en de interactie tussen mensen, overheden en bedrijven mogelijk maakt. Het vermijden en oplossen van softwaredefecten met betrekking tot beveiliging (d.w.z. software kwetsbaarheden) is een essentieel aspect van deze praktijk. Om dit doel te bereiken bestaan er methoden die software kwetsbaarheden kunnen detecteren met behulp van deep learning.

Ons belangrijkste onderzoeksdoel was om een dergelijke methode voor het detecteren van kwetsbaarheden te verbeteren door gebruik te maken van verklaringen van onjuiste classificaties. Voor dit doel hebben we gemeten hoe nauwkeurig de Layerwise Relevance Propagation (LRP) methode kwetsbare regels kon detecteren die zich in 51.586 kwetsbare voorbeelden bevonden en hoe volledig LRP kwetsbare regels in de voorbeelden kon lokaliseren. Daarnaast hebben we vier experimenten uitgevoerd om de prestaties van het model te verbeteren. Deze experimenten filteren broncode elementen, gevonden in onjuiste classificaties, uit samples tijdens en na het trainen van het model.

We ontdekten dat de LRP-methode meer dan twee keer zo veel kwetsbare regels detecteert dan een willekeurige detectie. Ondanks deze verbetering zijn er nog veel kwetsbare regels die niet gedetecteerd worden wat aantoont dat onze toepassing van de LRP methode in mindere mate geschikt is voor dit doel. Daarnaast hebben we bepaald hoe nauwkeurig de LRP methode de kwetsbare regels kan bepalen in een sample. Hieruit bleek dat, gemiddeld gezien, een groot deel van de kwetsbare regels in samples niet geselecteerd worden door de LRP methode. In dit onderzoek was het praktisch nut van onze toepassing van de LRP methode beperkt omdat onze voorbeelden vaak een enkele kwetsbare regel bevatten die zelden als zodanig werd aangemerkt.

De resultaten van onze experimenten laten zien dat het aantal correcte classificaties en de gemiddelde precisie lager werd door de meest relevante onderdelen uit onze samples te verwijderen in een getraind model. Het verwijderen van deze onderdelen voordat het model had geleerd van de data veranderde de gemiddelde precisie niet significant maar verminderde wel het aantal fout-positieve classificaties met 29% wat ten koste ging van het aantal fout-negatieve classificaties, wat steeg met 23%.

Onze analyse van de verklaringen vergroot ons inzicht over de toepassing van deep learning in de context van software kwetsbaarheden detectie. We tonen aan dat de kwetsbaar gelabelde regels geen grote rol spelen in de classificaties van ons model. Dit inzicht in het gedrag van het model was onverwacht en verminderde ons vertrouwen in zijn vermogen om kwetsbaarheden te detecteren in een andere toepassing. Onze aanpassingen aan de trainings procedure van het model zorgen ervoor dat, bij een gelijkblijvende classificatie drempelwaarde, een groter gedeelte van de gedetecteerde kwetsbaarheden relevant zijn. Ze hebben echter ook tot gevolg dat een kleiner gedeelte van de kwetsbaarheden gedetecteerd wordt. Voor een verbetering zouden beide proporties groter moeten zijn geworden en daarom hebben we niet zozeer de prestaties van het model verbeterd als wel de focus van het model verlegd in deze klassieke afweging bij het detecteren van kwetsbaarheden.

1

INTRODUCTION

1.1. SOFTWARE SECURITY

Software engineering is seen as a structured, disciplined approach to create, implement, and maintain software [41]. This engineering discipline has a long history, starting halfway in the twentieth century. Software security engineering, creating software that operates as expected in a potentially hostile environment, only started during the last decade of the twentieth century.

Nowadays, software is ubiquitously used (e.g., in smartphones, hospitals, or vehicles) and enables interactions between humans, governments, and companies (e.g., mobile banking, digital identification, or online shopping). As society has become dependent on software, engineering its security has also become more critical.

1.2. SOFTWARE VULNERABILITIES

Avoiding and fixing software defects related to security (i.e., software vulnerabilities) is an essential aspect of engineering secure software [23]. Software vulnerabilities are design (i.e., flaws) or implementation (i.e., bugs) errors which pose a risk to the intended use of the application. For example, an online banking environment vulnerability could allow unauthorized access to this bank account.

Software vendors report vulnerabilities found in their products to the national vulnerability database (NVD) in the US [26]. Each vulnerability is assigned a specific common weakness enumeration (CWE) type when reported. The CWE is a list of common weakness types, such as buffer overflow[1] or cross-site scripting[2]. The NVD reports that the number of vulnerability registrations steeply increased in the last three years (see figure 1.1). As a side note, an analysis by Micro Focus[3] attributes this spike to the combination of a broader range of CWE types and a tripling of the number of products affected.

1.3. SOFTWARE VULNERABILITY DETECTION

The NVD vulnerability report includes vulnerabilities in commonly used software products, such as Microsoft Internet Explorer, Google Android, or Adobe Acrobat Reader. These

[1] https://cwe.mitre.org/data/definitions/120.html
[2] https://cwe.mitre.org/data/definitions/79.html
[3] https://content.microfocus.com/application-security-risk-tb/2019-appsec-risk-report

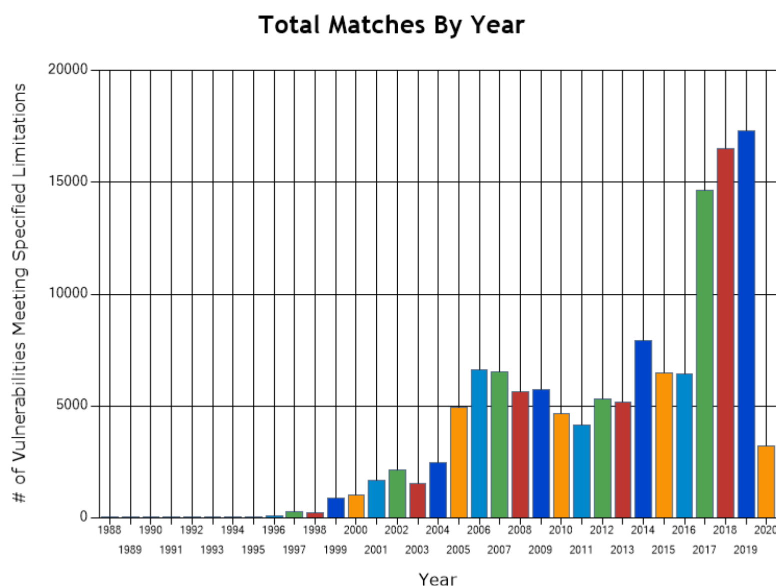


Figure 1.1: Yearly totals of published vulnerabilities. Source: National Vulnerability Database [26]

products are commonly built with the programming languages C or C++. These programming languages provide means for direct addressing of memory locations for efficiency and speed of computation. Direct addressing of memory requires diligent programming to avoid writing or reading unintended memory locations and is prone to bugs. These bugs are common root causes of vulnerabilities found in said products. For example, according to CVE details[4], the highest occurring vulnerability categories (see figure 1.2) for these products include *memory corruption*, *overflow* and *execute code*. These categories encompass CWE types which typically involve incorrect addressing of memory. For example, CWE-119: buffer overflow[5], CWE-416: Use After Free[6], or CWE - 787: Out-of-bounds Write[7].

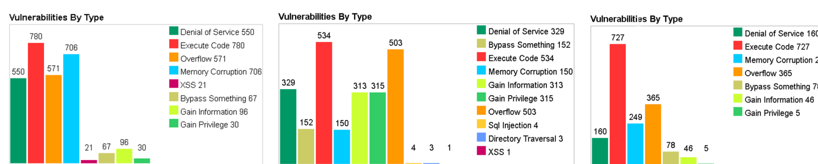


Figure 1.2: Vulnerability counts per type of Microsoft Internet Explorer, Google Android, and Adobe Acrobat. Source: CVE details [29].

[4]a site providing detailed information and statistics on the NVD database https://www.cvedetails.com
[5]https://cwe.mitre.org/data/definitions/119.html
[6]https://cwe.mitre.org/data/definitions/416.html
[7]https://cwe.mitre.org/data/definitions/787.html

Software vulnerability detection can be employed to find bugs related to the direct addressing of memory. Methods for vulnerability detection analyze source code or a running program. Analysis techniques such as string-pattern matching, integer range analysis, or tainted dataflow, suffer from false positives [44, 46, 14, 39]. Validating many false positives requires manual effort, but undetected vulnerabilities due to false negatives introduce a false sense of security. A software vulnerability detection tool that aims to keep the number of false positives low while eliminating false negatives is thus seen as useful [23].

Several studies show that machine learning and deep learning techniques can detect software vulnerabilities with good performance [22, 34]. For example, Li et al. compare their approach to state-of-the-art open-source and commercial tools. They report a decrease of 50% and 19.4% in false negative and false positive rates, respectively, compared to state-of-the-art static analysis tools [21]. To trust such an approach, we should be able to attain details on or validate decisions made by it. The complex models underlying their deep learning approach are not interpretable by humans. Explaining the evidence for a decision in an interpretable manner provides insight into its decisions which can build trust and allow for improvements of such an approach. Figure 1.3 shows an example of such an interpretable explanation. See section 2.4 for more details on explaining deep learning.

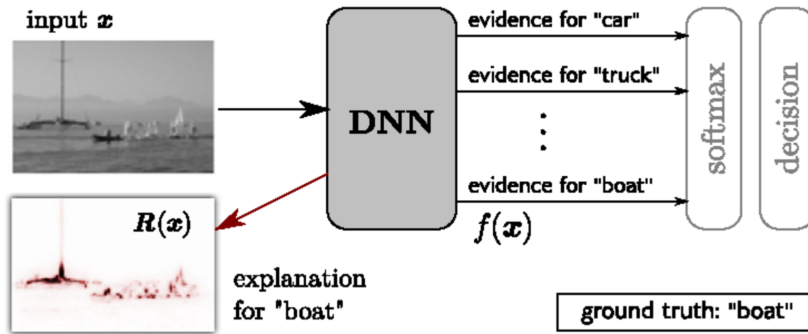


Figure 1.3: Explanation of image classification. The heatmap (lower left image) shows which parts of the image classify the image as a boat. Source Montavon et al. [25]

1.4. Goal

The main research goal was to improve the SySeVR vulnerability detection system of Li et al. [21] by explaining incorrect classifications by means of LRP. Towards this goal, we determined the vulnerable lines in samples and measure how many of these lines are used in classifications. Furthermore, we used these explanations to determine which parts are responsible for incorrect classifications and measured whether removing the influence of these parts during inference or model training lowered the number of incorrect classifications. Our contribution was the new application of an explanation method to a vulnerability detection approach.

1.5. Overview report structure

Chapter 2 provides the background for the research fields used in this thesis. Chapter 3 of this report shows the approach to reaching the research objective. Chapter 4 describes how samples in the NVD and SARD vulnerability datasets are transformed to the dataset of

Li et al. and how vulnerable lines of code are determined. Chapter 5 describes the explanation technique, which explains the classifications made by the deep learning approach of Li et al, and shows how we apply this technique to determine the relevant parts of vulnerabilities.[21, 3]. Chapter 6 describes how we use the obtained relevant parts to improve the vulnerability detection approach of Li et al. and reports the results of the experiments. The limitations of the research and the relevance of the results regarding the problem of vulnerability detection are discussed in chapter 8. Finally, chapter 9 concludes how the research objective is achieved and reiterates the supporting results. It also indicates potential further research opportunities.

2

BACKGROUND

2.1. MACHINE LEARNING

The research field of machine learning studies computer algorithms that can learn from data. Machine learning can be used to make predictions, e.g., an algorithm can learn property values from the location, lot size, or the number of rooms. Alternatively, to make classifications, e.g., classifying an e-mail as spam, an algorithm can analyze words in an e-mail. These relevant parts of information concerning the subject are known as features. Determining which features to extract from a subject to perform well on a machine learning task is commonly guided by domain knowledge. For example, algorithms can detect human faces in images with information about skin color or geometric relations between nose, eyes, or mouth. Manually determining and extracting these features requires a large amount of human effort, and it can be challenging in some tasks to determine the right features.

To reduce these shortcomings, research is conducted to automate the discovery of features of a specific machine learning task. Representation learning methods have been proposed which can discover relevant features in data [10]. For example, meaningful vector representations for words can be learned from text corpora. Mikolov et al. have shown that these representations, named word embeddings, capture similarities and relationships between words. These embeddings can improve the generalisation of word prediction tasks because previously unseen combinations of words roughly yields the same output as the output learned from example combinations. For example, sentence completion for the sentence: "violets are ?" can yield different types of colors even though they have not been seen during learning [24].

2.2. DEEP LEARNING

When confronted with natural input data (e.g., object detection from raw pixel values), representation learning methods can benefit from learning increasingly abstract representations[10]. Deep learning methods create layers of representations, with each layer expressed in terms of its predecessor. In an example of object detection in figure 2.1, the edges in the "1st hidden" layer are expressed in pixels from the input layer. The edges represent corners and contours in the "2nd hidden layer" that describe object representations in the final "3rd hidden layer". A machine learning classifier uses the abstract representa-

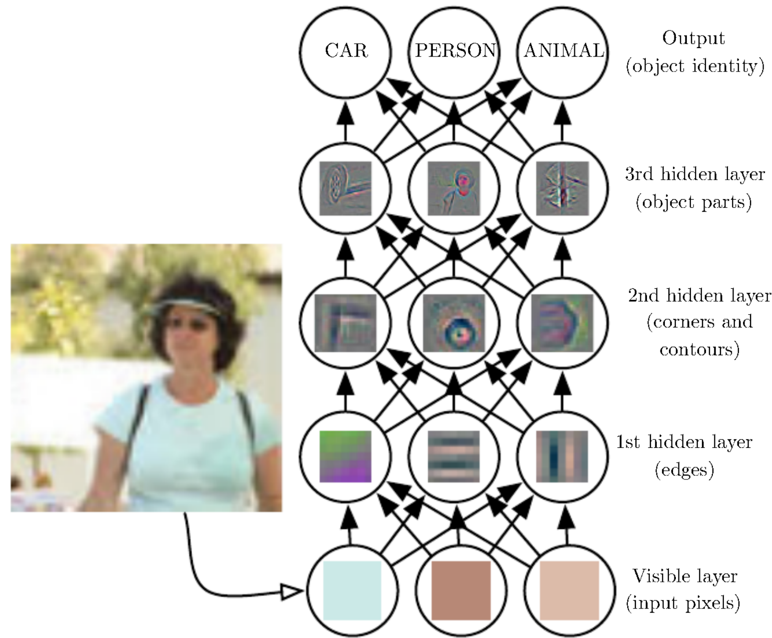


Figure 2.1: Deep learning allows computers to perceive high-level concepts by expressing them in terms of lower-level concepts. Image source: Deep learning by Goodfellow et al. [10].

tions in the final layer to determine what kind of object is most likely to be present in the example.

Deep learning methods train neural networks to learn the function $y = f'(x)$ of the machine learning task by forming layers of representations. These representations are called the parameters, or weights, w of the neural network. Training aims to yield a set of parameters to function $y = f(x, w)$ that approximates function f'. Neural networks are composed of neurons that are sending signals to related neurons in successive layers. The type and amount of layers and the loss function (which measures how much f approximates f') determine the architecture of a neural network. Each layer in the neural network can have its own parameters, called hyperparameters. For example, the layer dimensions or which activation function is used to compute the output of the layer. The fully-connected layer type is commonly applied in neural networks. All neurons in a fully-connected layer are connected to the neurons in the successive layer. Figure 2.2 shows an example neural network which consists of two fully-connected layers. Each node represents a neuron, and each arrow represents a weighted connection between two neurons. The neurons in the input layer receive real-valued features from the task. The output $y \in \mathbb{R}$ for a given neuron h after the input layer with n predecessor neurons $x_0, x_1, .., x_{n-1}, x_n$ and weights $w_0, w_1, .., w_{n-1}, w_n$ is

$$y = \sigma\left(b_h + \sum_{i=0}^{n} x_n w_n\right),$$

where σ is a function which determines how much this neuron is activated by its input and is commonly defined as $\sigma(x) = max(0, x)$. Term b_h is a real-valued scalar that can offset the weighted sum of the input. The network-output y is computed by the neuron o_0 in the final (output) layer. A loss function is applied to the network output during training in order to

measure the network's performance on the learning task.

The usual training process of neural networks is gradient descent using backpropagation (see [33] for details), and it encompasses three steps. The first step is computing the output of the neural network for a given example. The next step is computing the loss L of the network output with regard to the true class of this sample. Finally, an algorithm updates the neural network parameters in a layer-by-layer manner using the error gradient. This gradient is a vector containing partial-derivatives signifying how sensitive the loss L is to changes in the network's parameters. The update for a parameter p_0 is

$$p_0 = p_0 - \eta\left(\frac{\partial L}{\partial p_0}\right),$$

where η is the learning rate, which is a hyperparameter that determines the size of the update, and $\frac{\partial L}{\partial p_0}$ is the partial-derivative of L with respect to parameter p_0. The value of the parameter is decreased because the loss function is minimized. This process is repeated for the samples in the training data until the neural-network parameters are optimized for the machine learning task (i.e., good representations have been computed).

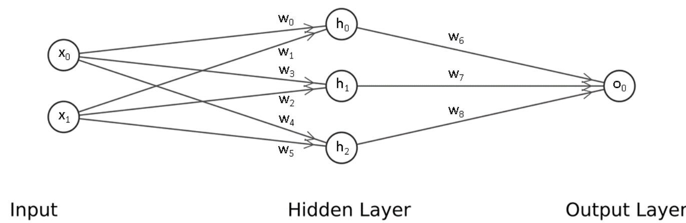


Figure 2.2: Neural network showing input neurons and their connections to the hidden and output layers. Image generated with nn-svg [20]

Deep learning tasks commonly apply two forms of neural networks, the recurrent neural network (RNN) and the convolutional neural network (CNN).

Recurrent neural network

Recurrent neural networks (RNN) have been used in the field of natural language processing for language analysis tasks, for example, machine translations, or describing images [4, 45]. RNNs store information from earlier seen input and use this to reason about the current input. See figure 2.3 for an example. RNN A processes input x_0 at timestep 0 to compute output h_0 and remembers its state (the hidden state). At time-step 1 RNN A uses this state together with input x_1 to compute output h_1 and remembers the new state. This process is repeated until the last input X_t at timestep t has been processed. White et al. and Gu et al. have shown that RNNs can be used to represent source code for software engineering tasks such as code clone detection and API usage proposals [49, 48, 11].

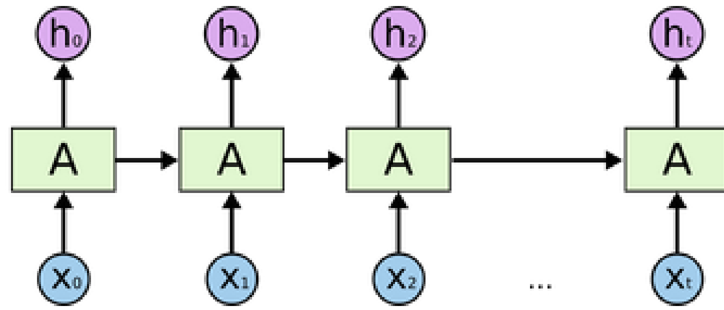


Figure 2.3: Recurrent neural networks use information from previously processed input for reasoning about current input. Image source: Understanding LSTM's by Olah [28].

Researchers show that training RNNs to learn from input over many timesteps is limited [5]. During backpropagation, the gradient of the network error tends to become either very small or very large. In the first case, the neural network learns very slow or not at all which is because of tiny parameter updates. In the latter case, parameter updates become very large, prohibiting the learning algorithm (e.g. gradient descent) from converging. Hochreiter and Schmidhuber showed that a RNN variant, the long short-term memory (LSTM) network, can overcome the problems mentioned above [15]. The nodes of a LSTM network contain, besides their hidden state, an additional cell state that functions as long-term memory. At each timestep, a node can add information (or remove information from) the long-term memory in a constrained manner, preventing the cell state from becoming either very small or very large.

Convolutional neural network

Convolutional neural networks (CNN) are often used in object- or handwriting-recognition in images [17, 19]). Designed to handle images in its learning task, the input layer of a CNN can accept a 3D volume of real-valued values (e.g., height, width, and color depth of an image). The CNN contains layers that employ filters to transform the volume into a new representation. Each filter is a small volume with trainable weights. See figure 2.4 for an example showing two filters. When a filter is moved across the input volume, it calculates a weighted sum at each step to detect patterns (see figure 2.5 for an example of detectable patterns), this is called the convolution operation. The number of weights required for this operation increases linearly with the depth of the input volume. This number remains constant even when the input volume's height and width are increased. In contrast, this would require a quadratic increase of weights in a fully connected layer. Therefore fewer calculations and memory are required for a CNN, which allow larger input dimensions [10].

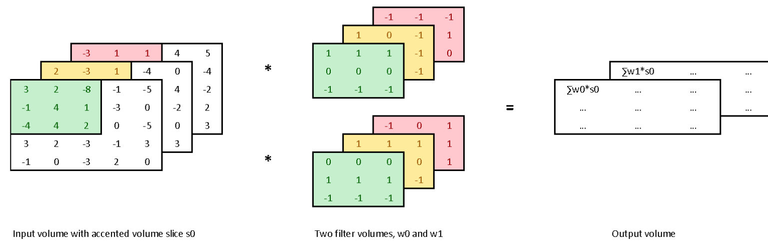


Figure 2.4: This images shows the first step of a convolution operation in a CNN. The input volume is transformed to an output volume using two filters. The weighted sum of the filter weights and the accented volume slice are stored in the output volume.

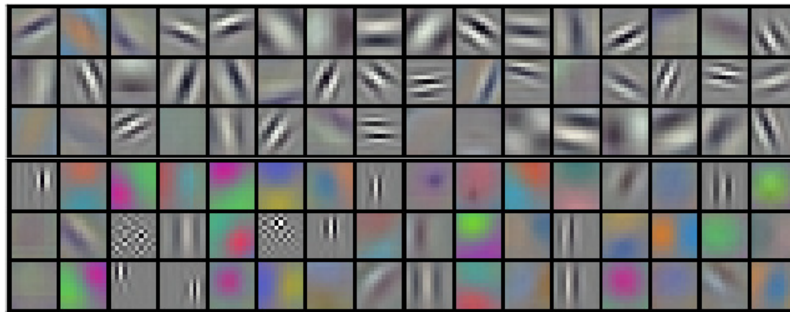


Figure 2.5: 96 Patterns learned by the first convolutional layer of the CNN by Krizhehvsky et al. Image Source: Imagenet classification with deep convolutional networks [17]

2.3. Vulnerability detection using deep learning

Detecting vulnerabilities using deep learning has roots in the software engineering practice of program analysis. Program analysis for software vulnerabilities is commonly performed using static or dynamic analysis. While static analysis detects vulnerabilities in the source (e.g., C source), bytecode (e.g., Java), or compiled (e.g., Intel x86) forms of a program [23, 7], dynamic analysis tries to detect vulnerabilities in programs that are executed by an operating system [12]. Deep learning methods detect vulnerabilities in source or compiled forms of a program and thus can be seen as a form of static program analysis.

Although software vulnerability detection using deep learning has been shown to detect vulnerabilities in source code with good performance, it also has limitations. Li et al. compare their approach (called SySeVR) to state of the art open-source and commercial tools and report a decrease of 50% and 19.4% in false negative and false positive rates respectively [21]. However, unlike traditional static software vulnerability analyzers, they do not report the evidence for each classification and report on a low level of granularity (i.e., many lines of code). Figure 2.6 shows an example of such a classification. This program slice of related statements is classified as vulnerable in its entirety, and the reason for this classification is not easily observed. As the granularity of reporting on vulnerabilities is an indication of the usefulness of a vulnerability analyzer this shows their limitations in this regard [7].

To detect vulnerabilities using deep learning, neural networks are trained to recognize patterns with supervised learning. Pattern recognition is performed on features created from source code with neural networks. Recent research employs convolutional and recurrent neural-networks [34, 21]. The former detects local correlations in the word-embeddings

```
1  283209 65521/CWE121_Stack_Based_Buffer_Overflow__CWE806_char_alloca_loop_10.c dest 41
2  void CWE121_Stack_Based_Buffer_Overflow__CWE806_char_alloca_loop_10_bad()
3  char * data ;
4  char * dataBuffer = ( char * ) ALLOCA ( 100 * sizeof ( char ) ) ;
5  data = dataBuffer;
6  if ( globalTrue )
7  memset ( data , 'A' , 100 - 1 );
8  data [ 100 - 1 ] = '\0';
9  char dest [ 50 ] = "" ;
10 size_t i , dataLen ;
11 dataLen = strlen ( data );
12 for (i = 0; i < dataLen; i++)
13 dest [ i ] = data [ i ];
14 dest [ 50 - 1 ] = '\0';
```

Figure 2.6: This image shows the granularity of a code gadget from the dataset by Li et al. [21]. A source buffer for 100 chars is created at line four and filled with 'A' at line seven. A destination buffer for 50 chars is declared and initialized at line nine. The number of iterations in the for-loop at line 12 depends on the length of the source buffer. Therefore, the assignment to the destination buffer at linenumber thirteen will be out of bounds after writing 50 characters and a stack-based buffer overflow occurs. This sample is classified as vulnerable.

for every source code token (i.e. a token produced by the lexical analysis of the source code). The latter neural network is chosen for its ability to process natural languages. These neural networks require real-valued input, and therefore the source code must be converted into a usable format. The word2vec technique can create word-embeddings from source code and is commonly employed as it can improve natural language processing tasks [24]. Figure 2.7 and 2.8 show how the neural network architectures used by Li et al. and Russel et al. process these word-embeddings as input.

These networks are then trained with supervised learning methods (i.e., learning from labeled examples). The training is performed on vulnerability datasets that contain the source code of both vulnerable and non-vulnerable programs. The studies of Russel et al. and Li et al. train the vulnerability detection classifiers on the SARD dataset, which contains a mix of synthetic, academic, and natural vulnerability test-cases written in the programming languages C or C++ [27, 34, 21]. Whereas Russel et al. add source code from a selection of GitHub projects and the Debian operating system, Li et al. add source code from software reported as vulnerable by the NVD such as Mozilla Firefox, Apache HTTP server, or OpenSSL [26]. Supervised learning methods require that the ground truth of each piece of source code (i.e., a sample) is known. Therefore a labeling procedure is performed to mark the samples as either vulnerable or non-vulnerable. While the SARD dataset provides these labels for each test case, the other source code is not labeled. Russel et al. and Li et al. use different labeling procedures. Russel et al. employ three static software analyzers (i.e., Clang, Cppcheck, and Flawfinder) to generate labels for samples created from the non SARD source code. Li et al. use patches (i.e. diffs) fixing the vulnerabilities to derive labels. A sample is marked vulnerable when it includes a removed or moved line described by such a patch. Both studies manually check all vulnerable classified samples for false positives.

After training the neural network, samples can be classified as vulnerable or non-vulnerable by using a classifier. Whereas Li et al. classify with a softmax classifier (which outputs the probability of a sample being of the vulnerable or non-vulnerable class), Russel et al. employ a random forest classifier which creates decision trees and chooses the most prevalent one.

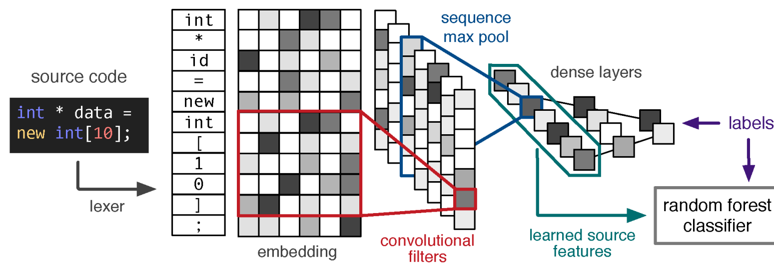


Figure 2.7: This figure shows the architecture of the convolutional neural network used in the paper by Russel et al. [34]. The source code sample is lexically analyzed to extract tokens. These tokens form the input for the learning task, and their word embeddings are stacked to form a matrix. Each convolutional filter combines information from sequential tokens. The output of each filter (a column vector containing the values for a group of tokens) is max-pooled to form a scalar feature value. Features are built with fully-connected layers, and classifications at test-time are performed with a random forest classifier

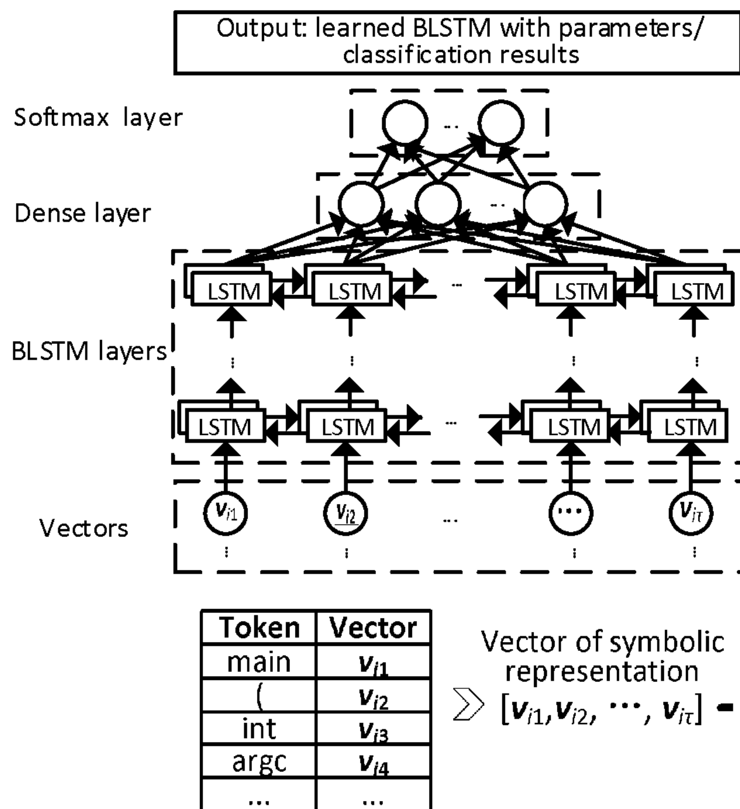


Figure 2.8: This figure describes the architecture of the neural network employed in the SySeVR paper by Li et al. [21]. The architecture is based on earlier work by Li et al. [22]. The input vectors are word-embeddings of the samples. Two LSTM layers extract the features from the vectors and forward their output to a (fully-connected) dense layer. The output of the dense layer is used in the Softmax layer, which calculates the probability of a sample being vulnerable or non-vulnerable.

There are two possible outcomes for the classification of a sample, vulnerable or non-vulnerable. As such, the detection of vulnerabilities can be seen as a binary classification problem and its performance can be expressed using standard metrics (e.g. precision, recall, or F1-score). In addition to the classification performance, we can also measure the time required by the system for training and classification since this is an indication of its practical usability.

Finally, the detection performance can be expressed with metrics and time-to-detect measurements. Also, detection performance can be compared to the performance of rule-based static analyzers. Compared to rule-based static analyzers such as tainted dataflow analysis or integer range overflow analysis, vulnerability detection using deep learning is limited in its applicability. Whereas rule-based analyzers can apply their rules on previously unseen source code, deep learning analyzers must be retrained if they are to be applied to such source code.

2.4. EXPLAINING CLASSIFICATIONS

There are various reasons for explaining classification decisions by machine learning algorithms [38]. An erroneous classification could incur high costs (for example, weather predictions involving critical infrastructure) or damage a person (for example, an unnecessary treatment due to an incorrect diagnosis). Such machine learning algorithms need to be explainable in order to provide details on or verify decisions. European law (i.e., GDPR) includes requirements on meaningful explanations of machine learning decisions, or profiling [8]. Another reason to explain a machine learning algorithm can be to identify biases or weaknesses to improve the algorithm. Finally, explaining an algorithm can provide humans with new knowledge concerning its prediction strategy [36].

Explanation methods have been created which provide insight into classifications. These methods have a certain perspective on a neural network. A mechanistic (i.e., white-box) perspective can be used to understand how a neural network solves a problem. For example, Karpathy et al. show how recurrent neural networks learn to represent data over time [16]. A functional perspective (i.e., black-box) can be used to understand how a neural network relates input to the output. For example, Montavon et al. describe which individual pixels contribute to a classification (see Figure 1.3) [25].

Samek and Müller categorized methods for explaining classification decisions in their introductory paper on explaining artificial intelligence [35]. They describe the following categories.

Explanation using surrogates These methods explain a prediction of a non-interpretable model by sampling variations of the input to create an interpretable surrogate function. The LIME research by Ribeiro et al. presents an example of such a method [31]. They demonstrate that a black-box model is explainable in the domains of text and image classification. They create a new model (a linear model in the paper) based on samples created by slightly altering a prediction and measuring the change in classification in the original model. The interpretable model explains a text classification example by showing which words are the most relevant towards its predicted class.

Explanation using local perturbations These methods explain a model's prediction by changing the input to and measuring the difference in the output. They delete parts of an

image and measure the effect of the deletion on a target classification. They explain a classification by highlighting the image region that has the most influence on the classification. An example of such methods applied in the domain of image recognition is a work by Fong and Vedaldi [9].

Explanation leveraging structure These methods provide explanations by leveraging the structure of the model. An example of these methods the layer-wise relevance propagation method by Bach et al. which demonstrates that explanations of image classifications in terms of their input (i.e. pixels) can be created by propagating the classification score backward through the layers of a neural network [3]. They visualize how much each pixel contributes towards the classification output by projecting a heatmap over the classified image.

Meta explanations These methods aim to explain how a classifier behaves in general or how to interpret learned representations. An example of such methods is the recent work by Lapuschkin et al. which poses a method to describe typical and atypical predictions of a model [18]. They create relevance heatmaps for a single classification. These heatmaps are clustered and analyzed to determine which clusters indicate a different classification pattern. The original images augmented with heatmaps explain the classification patterns.

3

RESEARCH DESIGN

3.1. RESEARCH MOTIVATION

Although software vulnerability detection methods with deep learning can detect vulnerabilities in source code with good performance, they do not report the evidence for each classification and classify on a low level of granularity (i.e., many lines of code). Determining if explanations of these classifications can precisely locate relevant parts of vulnerabilities has relevance for science, practical use, and society. First, we can improve existing research approaches using deep learning by finding reasons for their incorrect classifications. Secondly, as we cannot easily validate or attain details on classifications, such methods are limited in their practical use. Software security analysis benefits from precise explanations of vulnerabilities as this lowers the effort required to validate classifications. Consequently, society benefits when the number of vulnerabilities in commonly used software decreases due to improved detection performance and usability.

To decide which vulnerability detection approach to improve by analyzing incorrect classifications requires two choices. First of all, we choose a vulnerability detection system. To improve such a system, we require access to the analyzed data to reproduce experimental results in addition to having room for improvement and an available or reproducible implementation. Both Li et al. and Russel et al. showed room for improvement due to the high (but not perfect) F1 metrics (92.6 % and 84% respectively) [34, 21]. Even though both approaches have published their analyzed datasets, Li et al. include the original source code containing the vulnerabilities (i.e., the source code before the transformation to a sample). Additionally, the implementation by Li et al. is publicly available, whereas the implementation by Russel et al. is not.

Furthermore, we required a method to explain incorrect vulnerability classifications. Such an explanation method must be compatible with the employed deep learning technique, provide explanations in an interpretable manner, and in a practical time frame. The explanations themselves should be precise enough to support assessing the root cause for incorrect classifications. Two commonly applied methods are sensitivity analysis (SA) and layer-wise relevance propagation (LRP) [32, 3]. Both methods are compatible with the deep learning models employed by Li et al. and Russel et al. Sensitivity analysis is a local perturbation explanation method that uses the gradient with respect to the neural network function to relate the input of a model to the output. It can show which parts of a sample lead to an increase or decrease of the classification score when changed. On the other hand,

layer-wise relevance propagation (LRP) is a propagation-based method that leverages the neural network structure to relate the output to the input. It determines which parts of a sample processed by a model have contributed towards a classification (i.e., are relevant in the sample according to the model). While both methods can provide explanations in an interpretable form, recent studies show that LRP can identify relevant parts in image and text classification settings with higher precision than sensitivity analysis [37, 2].

In conclusion, the approach by Li et al. was a better candidate for improvement because we could recreate their experiments more accurately, and LRP is a better method to explain incorrect classifications due to its higher performance in a text-based classification task.

3.2. Research objective

The main research goal was to improve the SySeVR vulnerability detection system of Li et al. [21] by explaining incorrect classifications by means of LRP. Towards this goal, we determined the vulnerable lines in samples and measure how many of these lines are used in classifications. Furthermore, we used these explanations to determine which parts are responsible for incorrect classifications and measured whether removing the influence of these parts during inference or model training lowered the number of incorrect classifications. Our contribution was the new application of an explanation method to a vulnerability detection approach.

3.3. Research questions

Main research question

How can we use explanations of incorrect classifications by means of LRP to improve the SySeVR deep learning vulnerability detection system?

Contributing research questions

These sub-questions support the main research question and are described in the following sections.

- RQ1: How precise are vulnerable lines in samples detected using deep learning?
- RQ2: How can we improve the classification performance using the relevance of features contributing to incorrect classifications during inference?
- RQ2: How can we improve the classification performance using the relevance of features contributing to incorrect classifications during model training?

RQ1: How precise are vulnerable lines in samples detected using deep learning?

The layer-wise relevance propagation (LRP) technique described by Bach et al. can determine the relevance of individual pixels concerning an image classification [3]. This technique is also used by Arras et al. to determine the relevance of words in sentiment classification of movie reviews [2].

Samek et al. show that LRP is more suitable to identify relevant parts in classifications than sensitivity analysis [37]. Henceforth, we will use the LRP technique to determine the

vulnerable lines in samples detected using deep learning. Li et al. have published the implementation and dataset of their approach to detect vulnerabilities using deep learning [21]. We used this approach as an example to answer this research question. The precision is expressed in the similarity between vulnerable and relevant parts of detected vulnerabilities. For this similarity, the Intersection over Union (IoU) metric is proposed. The average precision for true positive and false negative samples will be reported.

We hypothesized that incorrect classified vulnerabilities using deep learning had a less precise explanation using the LRP method than correctly classified vulnerabilities. We verified this hypothesis by measuring this relation in the SySeVR vulnerability detection system.

RQ2: How can we improve the classification performance using the relevance of features contributing to incorrect classifications during inference?

We identified which features in both non-vulnerable and vulnerable samples are responsible for incorrect classifications. The research question are answered by measuring whether diminishing the influence of these features on the samples will lower the number of incorrect classifications in our model test data. To validate the improvements, we performed experiments and reported on the results using the SySeVR evaluation metrics (see table 3.1).

RQ3: How can we improve the classification performance using the relevance of features contributing to incorrect classifications during model training?

We used the features in samples which are responsible for incorrect classifications to change the training procedure. We determined whether removing the influence of these features on the model training samples would lower the number of incorrect classifications in our model test data. To validate the improvements, we performed an experiment and reported on the results using the SySeVR evaluation metrics.

3.4. Research method

This section describes the methods used in answering our research questions.

Research method RQ1

To answer RQ1 we required a dataset with vulnerable samples, the location of the vulnerability (i.e. the vulnerable lines in the sample), and the explanations of the classifications. Figure 3.1 describes how we obtained these artifacts. The orange accented parts contain research output from Li et al. and Arras et al., which we used to build our own research project. First, the dataset employed by the SySeVR system contains non-vulnerable and vulnerable samples created from source files obtained from the National Vulnerabilities Database (NVD) and the Software Assurance Reference Dataset (SARD) [21, 26, 27]. We extracted the SARD samples from the SySeVR dataset because the SARD metadata describes the location of the included vulnerabilities. We used these locations to determine the vulnerable lines of code in our subset of SySeVR dataset. We started our dataset by combining

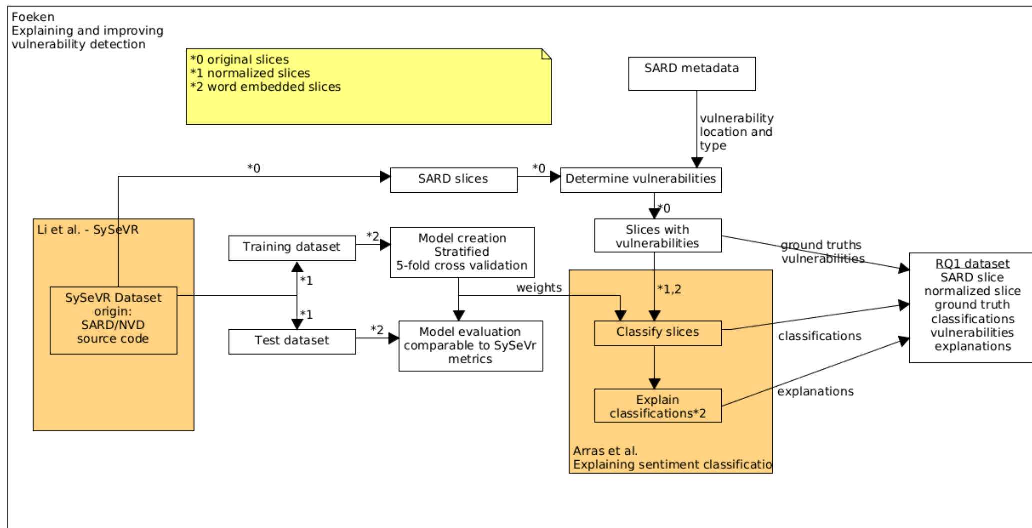


Figure 3.1: This figure describes the preparation of our dataset containing vulnerable samples, the precise locations of vulnerabilities, sample classifications and the explanations thereof. The orange accented parts show research output by Li et al. and Arras et al. which is used.

the extracted SARD samples and locations of the vulnerabilities thereof. Second, we required a deep learning model to classify this dataset. Since Li et al. have not published the SySeVR model, we thus needed to create a comparable model because this supported our findings and improvements. Therefore, we have split their dataset into a training and testing dataset. We created an equivalent model and trained this model on the training dataset using stratified cross-validation until we had comparable scoring performance metrics on the test dataset. This model and its metric scores served as a baseline for our findings and improvements. Finally, to complete our dataset, we explained these classifications using the LRP procedure described by Arras et al. in their research on explaining deep learning classifications [2]. The LRP procedure explains classifications of a recurrent (LSTM) model, the implementation of this model was adapted to overcome a difference in classification type (see paragraph 5.2.1 for more details). Whereas their model predicted five classes of sentiment in movie reviews, from (very) negative to neutral to (very) positive, our model predicted two classes, vulnerable or non-vulnerable. Therefore, we changed the activation function of the final layer in their model to make it consistent with our model. We compared the classification output of both models on a small number of samples to validate whether our adaptation to the model produced the same classifications.

We measured two precision values and determined whether our initial hypothesis is valid in RQ1. Figure 3.2 shows the necessary steps. We started with the prepared dataset and selected both vulnerable (step one) and relevant (step two) lines. The line detection precision expresses (step three) how precise LRP detects these vulnerable lines in our dataset. The detection is based on how much relevance the LRP procedure assigns to each line of code. We measured how many selected lines are vulnerable (precision metric) and how many vulnerable lines are selected (recall metric) at different relevance thresholds. We report these measurements with a precision-recall curve and average precision. The localization precision demonstrates how precise LRP can locate vulnerabilities in slices. Towards this end, we selected lines from slices having relevance above a fixed threshold.

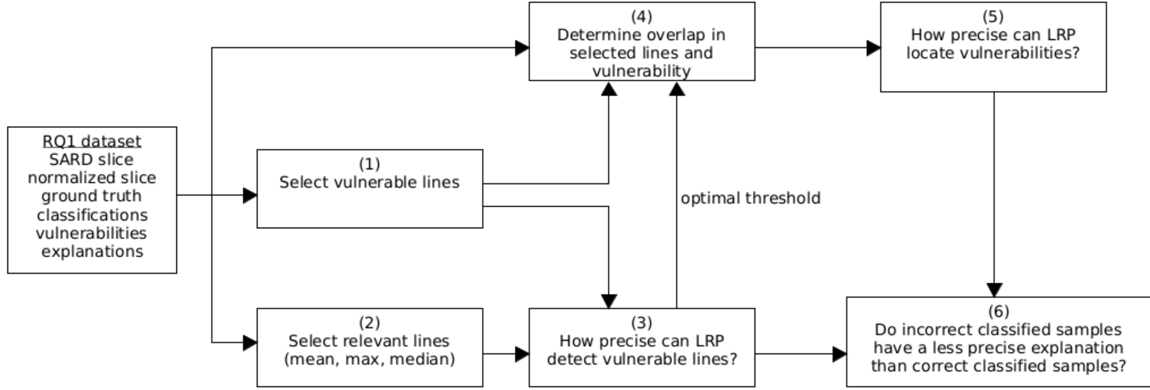


Figure 3.2: This figure shows which steps we have taken to answer RQ1.

We measured the ratio of overlapping selected and vulnerable lines per sample using the intersection over union metric described in 3.6 (step four). We reported the distribution of this measurement with a histogram and average precision (step five). Using the measurements obtained in these steps, we determined whether incorrect classified samples have less precise explanations than correct classified samples (step six) and answer RQ1. We reported the vulnerable line detection precision and vulnerability localization precision broken down into false negative and true positive slices.

RESEARCH METHOD RQ2

After reporting on the findings regarding the first research question, we conducted experiments to answer RQ2. Figure 3.3 shows the high-level approach. The goal of these experiments is to decrease incorrect classifications and thereby increase classification performance. The first two experiments follow the same steps. We selected incorrect classified samples (step one and four) from our dataset and retrieve the tokens whose relevance contribute strongly towards incorrect classifications (step two and five). We concluded these experiments by removing these tokens (steps three and six) during inference and measuring the change in classification performance using the SySeVR metrics. Section 3.5 describes these metrics in detail. In the third experiment (step seven), we combined and applied the changes to the dataset from the previous two experiments. We measured and reported the changes in model performance metrics using the SySeVR metrics.

RESEARCH METHOD RQ3

We answered RQ3 by improving the model training procedure using the most promising token filter found in our experiments towards answering RQ2. We applied this token filter in the model training procedure (step eight in figure 3.3) and tested whether this lowers the number of incorrect classifications in our test data. We measured and reported the changes in model performance metrics using the SySeVR metrics.

3.5. SYSEVR METRICS

This section describes the evaluation metrics that are used to measure the performance of the SySeVR system. Table 3.1 lists the formula for each metric.

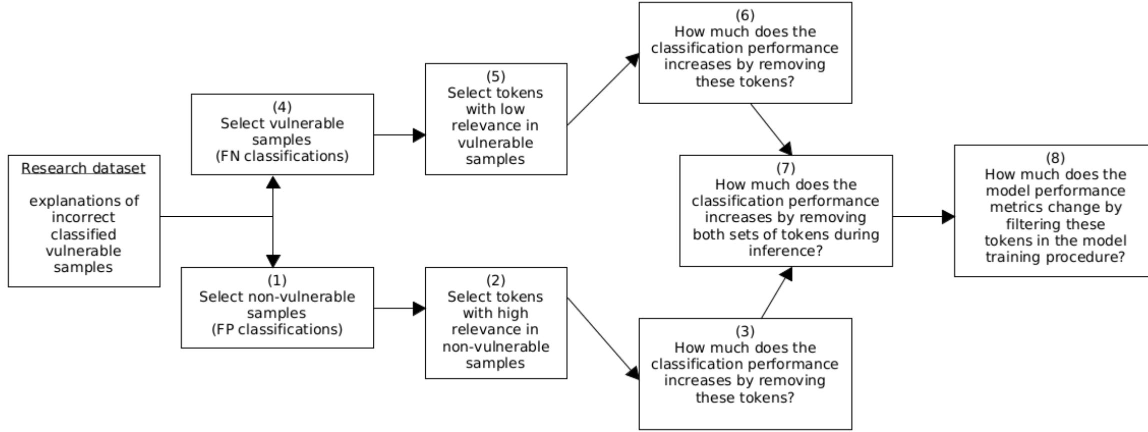


Figure 3.3: This figure shows proposed experiments which answer research questions RQ2 and RQ3.

Table 3.1: SySeVR performance metrics

Metric	Formula
False positive rate (FPR)	$\frac{FP}{FP+TN}$
False negative rate (FNR)	$\frac{FN}{TP+FN}$
Accuracy (A)	$\frac{TP+TN}{TP+TN+FP+FN}$
Precision (P)	$\frac{TP}{TP+FP}$
Recall (R)	$\frac{TP}{TP+FN}$
F-Score (F1)	$2 \times \frac{P \times R}{P+R}$
Matthews correlation coefficient (MCC)	$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP) \times (TP+FN) \times (TN+FP) \times (TN+FN)}}$

Where FP is defined as the number of non-vulnerable samples predicted vulnerable, TP is defined as the number of vulnerable samples predicted vulnerable. FN is defined as the number of vulnerable samples predicted non-vulnerable, and TN is defined as the number of non-vulnerable samples predicted non-vulnerable. The FPR metric shows the proportion of incorrect vulnerable predictions with regard to all non-vulnerable samples (e.g., false alarms). The FNR metric measures the proportion of incorrect non-vulnerable predictions (e.g., missed vulnerabilities) with regard to all vulnerable samples. The accuracy metric shows how often a correct prediction is made concerning vulnerable and non-vulnerable samples. The SySeVR dataset has an imbalanced class distribution (e.g., many more non-vulnerable samples than vulnerable samples). Consequently, the amount of correctly classified non-vulnerable samples can be very high. Because vulnerability detection systems should find vulnerable samples, such high accuracy is not a fair indicator of the model performance. The precision and recall metrics are defined without using the actual non-vulnerable samples (i.e., without the FN class) and thus are a better indicator of the model performance. Precision is defined as the proportion of vulnerable samples with regard to all vulnerable classified samples (e.g., how often is the model correct when a vulnerability is predicted), and the recall metric is defined as the proportion of vulnerable classified samples with regard to all vulnerable samples (e.g., how complete is the model in selecting the actual vulnerable samples). The F1 metric is the harmonic

mean of these two metrics and expresses the overall effectiveness of the model. Finally, the MCC measures the correlation between the predictions of the samples (for both vulnerable and non-vulnerable samples) and whether a sample is vulnerable. Unlike the F1 metric, the MCC metric calculates this correlation using the negative class. It, therefore, can indicate whether classes are correctly labeled (i.e., negative for the majority of samples non-vulnerable and positive for the minority vulnerable samples).

3.6. Intersection over Union

This section describes the intersection over union (IOU) metric that is used to measure the similarity between truly vulnerable and detected vulnerable lines. The IOU metric is calculated as:

$$IoU = \frac{|V \cap S|}{|V \cup S|},$$

where V = {l | l is a vulnerable linenumber} and S = {l | l is a detected vulnerable linenumber)}. The metric has a range of [0, 1] where zero and one IoU express no overlap and total overlap respectively. For example, table 3.2 shows a sample which has four vulnerable lines (e.g. a 1 in the label column) and two predicted vulnerable lines (i.e., a 1 in the detection column). For this sample the set V is defined as {16, 17, 18, 19} and S is defined as {17, 19}. The sample has $\frac{|V \cap S)|}{|V \cup S|} = \frac{2}{4} = 0.5$ IoU.

Table 3.2: This table shows lines in a vulnerable sample. The first column shows the line number, the "Label" column contains a 1 if a line is vulnerable and the "Detected" column contains a 1 if the line is predicted vulnerable. The vulnerable and detected lines are used in the calculation of the IOU metric.

#	Label	Detection	Line
1	0	0	**void** initlinedraw(**int** flag)
2	0	0	**int** stonesoup_i = 0 ;
3	0	0	**char** ∗ owlishly_ionospheres ;
4	0	0	**if** (__sync_bool_compare_and_swap (& trance_deforciant , 0 , 1))
5	0	0	**if** (mkdir ("/opt/stonesoup/workspace/lockDir", 509U) == 0)
6	0	0	**if** (owlishly_ionospheres != 0)
7	0	0	alphabetizers_lbl = ((**char** ∗) owlishly_ionospheres);
8	0	0	stonesoup_data = (**struct** stonesoup_struct ∗) malloc (**sizeof** (**struct** stonesoup_struct));
9	0	0	**if** (stonesoup_data != NULL)
10	0	0	memset (stonesoup_data −> before , 'A', 63);
11	0	0	stonesoup_data −> before [63] = '\0';
12	0	0	memset (stonesoup_data −> buffer , 'Q', 63);
13	0	0	stonesoup_data −> buffer [63] = '\0';
14	0	0	memset (stonesoup_data −> after , 'A', 63);
15	0	0	stonesoup_data −> after [63] = '\0';
16	1	0	stonesoup_buff_size = ((**int**) (strlen (alphabetizers_lbl)));
17	1	1	memcpy (stonesoup_data −> buffer , alphabetizers_lbl , 64);
18	1	0	**for** (; stonesoup_i < stonesoup_buff_size; ++stonesoup_i)
19	1	1	stonesoup_printf ("%x", stonesoup_data −> buffer [stonesoup_i]);

4

Model and dataset preparation

4.1. Model preparation

Li et al. created a vulnerability detection approach (the SySeVR system) and constructed a dataset to train it [21]. This section introduces the SySeVR system and describes our work in recreating their model.

4.1.1. SySeVR vulnerability detection

The SySeVR vulnerability detection approach by Li et al. is inspired by the concept of region proposal for object detection in images. Object detection with region proposals uses predictions of object bounds in images to train neural networks [30, 40]. Figure 4.1 shows (red-accented) region proposals (predictions of object regions) in the top part. These regions are extracted to form individual samples for a supervised deep learning task. Whereas the region proposal network by Ren et al. learns the regions from input data, the SySeVR framework employs fixed rules to create regions, these rules are derived from vulnerability detection rules found in the commercial static analysis tool Checkmarx [6]). The bottom part of the figure shows the transfer of the concept of the region to the software vulnerability detection context. The SySeVR system use the rules to locate potentially vulnerable parts (for example, statements including pointers) in source programs (the SyVCs block in figure 4.1). These potentially vulnerable parts and the related source code form the regions of interest (the SeVCs block in figure 4.1) for the software vulnerability detection task. The regions are related to library/API function calls, array or pointer usage, or arithmetic expressions and are called "kinds" in the SySeVR system. To summarize the approach, we describe the five high-level steps the SySeVR method consists of.

1. Vulnerable and non-vulnerable source code are collected to serve as training and testing data (programs for learning and target programs in figure 4.1).
2. The region detection rules are applied to the collected source code and yield the four kinds of potentially vulnerable program slices (the SyVCs and SeVCs blocks in figure 4.1).
3. A labeling method determines whether a vulnerability candidate contains a vulnerability and labels it accordingly (not shown in figure 4.1).

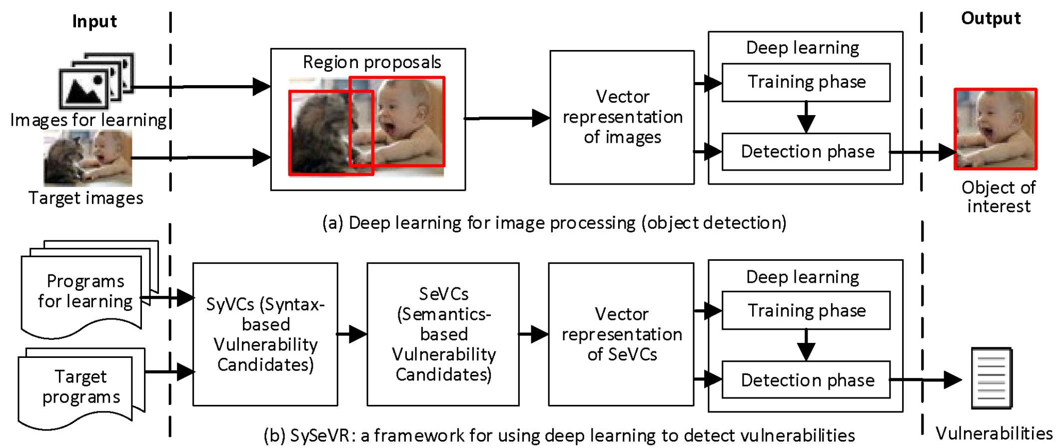


Figure 4.1: This images shows the application of region proposal for object detection to software vulnerability detection.

4. Vulnerability candidates are transformed into an effective representation for deep learning (block "vector representation of SeVCs" in figure 4.1).

5. A deep learning model is trained to predict vulnerabilities ("Deep learning" block in figure 4.1).

4.1.2. SySeVR dataset and implementation

The published SySeVR dataset contains 420,627 samples (56,395 vulnerable and 364,232 non-vulnerable) created using source code with natural and synthetic vulnerabilities obtained from the National Vulnerability Database (NVD) and Software Assurance Reference Dataset (SARD) respectively [21, 26, 27]. The dataset consists of four text files containing all samples of the same kind. Figure 4.2 (lower listing) shows a vulnerable SySeVR sample extracted from the SARD dataset. A region rule detected array usage in the source code (line 12). The stack-based buffer overflow occurs at line 13.

The software described in the paper of Li et al. which analyses the dataset is published at Github[1] [21]. The software is written in the Python[2] programming language and uses the Tensorflow[3] platform for machine learning. It comprises three modules, the first module parses the source code and extracts SeVCs, the second module transforms the SeVCs into samples, and the final module trains the deep learning model and outputs test results.

4.1.3. Deep learning model creation

To create a deep learning model comparable to the SySeVR model we followed the training procedure described in the SySeVR publication [21]. The SySeVR dataset was randomly split into 80% training and 20% testing samples. Because the in-memory dataset splitting led to out-of-memory errors, we changed the implementation to a dataset format (Tensorflow data[4]) that supports data streaming during processing. This prevented further out-of-memory errors during training.

[1] https://github.com/SySeVR/SySeVR
[2] https://www.python.org
[3] https://www.tensorflow.org
[4] https://www.tensorflow.org/guide/data

```
23  void CWE121_Stack_Based_Buffer_Overflow__CWE806_char_alloca_loop_10_bad()
24  {
25      char * data;
26      char * dataBuffer = (char *)ALLOCA(100*sizeof(char));
27      data = dataBuffer;
28      if(globalTrue)
29      {
30          /* FLAW: Initialize data as a large buffer that is larger than the small buffer used in the sink */
31          memset(data, 'A', 100-1); /* fill with 'A's */
32          data[100-1] = '\0'; /* null terminate */
33      }
34      {
35          char dest[50] = "";
36          size_t i, dataLen;
37          dataLen = strlen(data);
38          /* POTENTIAL FLAW: Possible buffer overflow if data is larger than dest */
39          for (i = 0; i < dataLen; i++)
40          {
41              dest[i] = data[i];
42          }
43          dest[50-1] = '\0'; /* Ensure the destination buffer is null terminated */
44          printLine(data);
45      }
46  }
```

```
1   283209 65521/CWE121_Stack_Based_Buffer_Overflow__CWE806_char_alloca_loop_10.c dest 41
2   void CWE121_Stack_Based_Buffer_Overflow__CWE806_char_alloca_loop_10_bad()
3   char * data ;
4   char * dataBuffer = ( char * ) ALLOCA ( 100 * sizeof ( char ) ) ;
5   data = dataBuffer;
6   if ( globalTrue )
7   memset ( data , 'A' , 100 - 1 );
8   data [ 100 - 1 ] = '\0';
9   char dest [ 50 ] = "" ;
10  size_t i , dataLen ;
11  dataLen = strlen ( data );
12  for (i = 0; i < dataLen; i++)
13  dest [ i ] = data [ i ];
14  dest [ 50 - 1 ] = '\0';
```

Figure 4.2: This figure contains two listings. The upper listing shows a SARD testcase and the lower listing shows a corresponding SySeVR sample. SySeVR sample extraction occured because of the dest array usage at line 41. This can be seen in the first (header) row in the SySeVR sample

We created a bi-directional LSTM model (Bi-LSTM) as employed in the SySeVR system with two modifications to allow for the explanation of its classifications by means of LRP.

The first modification concerned the number of Bi-LSTM layers used in the model. Figure 4.3 shows the architecture of the SySeVR and LRP models. In the upper part can be observed that the SySeVR system employs a model with two Bi-LSTM layers and in the lower part that the LRP procedure assumes a model with a single Bi-LSTM layer. This prevents the LRP procedure from explaining SySeVR classifications because it uses the structure of the model. This difference in number of layers could be solved by adding a Bi-LSTM layer to the LRP procedure or conversely, removing a Bi-LSTM layer from the SySeVR model.

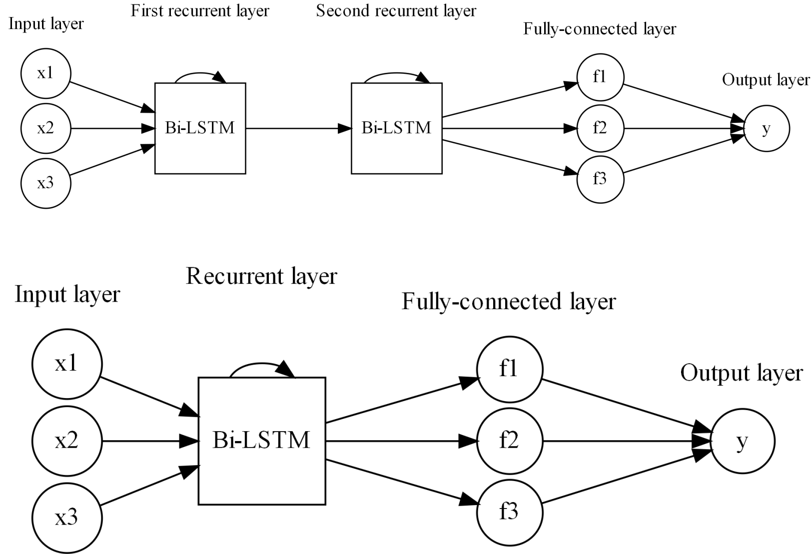


Figure 4.3: Architecture of SySeVR vulnerbility detection and the LRP sentiment detection models. The SySeVR model (top part) has two Bi-LSTM layers whereas the LRP model (bottom part) contains only one.

Adjusting the LRP procedure would incur the risk that the explanations could become incorrect. On the other hand, modifying the SySeVR model would introduce the risk that the classification performance would be lower than the original model.

The exploration of the first solution direction showed that complex changes to the LRP implementation were necessary to adjust the procedure. On the other hand, the exploration of the second solution showed minor differences in the performance metrics after retraining the model. Because of the uncertain efforts required to implement the complex changes to the LRP procedure, we accepted the small differences in the precision and recall metrics and exclude the second layer in our model.

The second modification to the SySeVR model was, on the recommendation of Arras et al., to remove the bias in the output layer in the TF model during training. This advantage is that all relevance from the output layer is distributed to the fully connected layer, and no relevance is lost in the biases.

Also, model training was conducted using 5-fold cross-validation on 30K samples randomly selected from the training dataset and evaluated with 7.5K samples randomly selected from the testing dataset. Stratification was used in these selections to preserve the ratio of vulnerable/non-vulnerable samples and the ratio of sample kinds.

Finally, we evaluated model performance using the SySeVR evaluation metrics. We compared models with the F1 metric since the SySeVR model hyperparameters were obtained by maximizing this metric. Table 4.1 shows the performance scores of the SySeVR model, our initial two-layer model, and our final, single-layer model. Even though our final and the SySeVR models show comparable F1 scores (84.0% and 84.4% respectively), there are some differences. Our final model has 4.7% lower precision and 3.8% higher recall than the SySeVR model. Our explanation for this difference is that, due to training and test samples being randomly selected, there are different samples included in the SySeVR dataset and ours. If the samples in our dataset require a lower threshold to classify as vulnerable this would decrease the number of FN samples and increase the number of FP samples leading to the higher recall but lower precision. Finally, as the final MCC metric value (81.8%) is near our final F1 metric value (84.0%), we deduce that our classes are correctly labeled.

Table 4.1: Evaluation metrics of SySeVR, initial, and final model.

	SySeVR	initial model	final model
Bi-LSTM layers	2	2	1
FPR	1.7%	2.6%	2.4%
FNR	19.0%	14.4%	15.2%
A	96.0%	96.0%	96.0%
P	88.0%	82.5%	83.3%
R	81.0%	85.6%	84.8%
F1	84.4%	84.0%	84.0%
MCC	82.2%	81.7%	81.8%

4.2. Vulnerabilities dataset

To measure how precise LRP can locate relevant parts of vulnerabilities, we select samples from the SySeVR dataset with known vulnerable lines of code. This section describes the samples in our dataset, shows how we located the vulnerable lines and reports how we validated our dataset samples.

4.2.1. Sample origins

To determine the relevant parts of vulnerabilities, we require that our dataset differentiates between non-vulnerable and vulnerable lines of code. The vulnerable lines of code in the SySeVR dataset are unknown. Therefore, we will use the SARD metadata, which describes vulnerability locations, to label the lines of code in our dataset. This limits our dataset to SySeVR samples originating from the SARD dataset. We show in figure 4.2 (upper listing) an example of a SARD source file. The source code contains a stack-based buffer overflow vulnerability. The for loop starts at line number 39 and copies characters using the length of the source buffer instead of the length of the destination buffer. Hence a buffer overflow occurs at line 41.

The SARD metadata on 67,984 test cases is described in an XML file. Figure 4.4 shows the metadata for the SARD source file in figure 4.2. This test case metadata includes the

```
<testcase id="65521" type="Source␣Code"
                status="Accepted" submissionDate="2013−05−20"
                language="C" author="NSA/Center␣for␣Assured␣Software"
                numberOfFiles="4" testsuiteid="86␣108">
        <description><![CDATA[CWE: 121 Stack Based Buffer Overflow<br/>
                         BadSource:  Initialize data as a large string<br/>
                         GoodSource: Initialize data as a small string<br/>
                         Sink: loop<br/>
                         BadSink : Copy data to string using a loop<br/>
                         Flow Variant: 10 Control flow: if(globalTrue) and if(globalFalse)]]>
    </description>
        <file path="000/065/521/CWE121_Stack_Based_Buffer_Overflow__CWE806_char_alloca_loop_10.c"
                language="C" size="4327"
                checksum="4c6319dda6f2e678081009e82cb0ec9b30da4de8">
                <mixed line="41" name="CWE−121:␣Stack−based␣Buffer␣Overflow"/>
        </file>
        <file path="shared/108/io.c"
                language="C" size="5429"
                checksum="bcf531cb1922c03347031698c1c72eddadbf0d88"/>
        <file path="shared/108/std_testcase.h"
                language="C" size="4004"
                checksum="d1801c64bc49d5d05fb3d55f7cc7e3a925c47e2f"/>
        <file path="shared/108/std_testcase_io.h"
                language="C" size="1457"
                checksum="56de16829d5ac9d8086670ee5098217aa6694d26"/>
</testcase>
```

Figure 4.4: This image shows the metadata for the SARD testcase. This testcase describes a vulnerability at line number 15 ("<mixed.../>" xml element. The stack-based buffer overflow vulnerability can be found at line number 41 in the file mentioned at line number 12

location and type of the vulnerability.

The SySeVR dataset contains 420,630 samples extracted from 1,591 NVD programs en 14,000 SARD test cases. After reducing the SySeVR dataset to samples originating from the SARD test cases, 313,030 samples remain (25,6% decrease). Figure 4.5 shows this broken down in sample kind.

SySeVR dataset count per kind:

API function call	:	64404
Array usage	:	42230
Pointer usage	:	291841
Arithmetic expression	:	22155
total	:	420630

SySeVR dataset count per kind after reducing to samples originating from SARD testcases

API function call	:	58047
Array usage	:	32416
Pointer usage	:	217951
Arithmetic expression	:	4616
total	:	313030

Figure 4.5: This figures shows the difference between the full SySeVR dataset and the SySeVR dataset reduced to samples originating from the SARD testcases.

We observe decreases per kind of 9.9% (API function call), 23.2% (array usage), 25.3% (pointer usage), and 79.2% (arithmetic expression). The large decrease in arithmetic expression samples was unexpected because the SySeVR paper stated that 23.9% of this kind originates from the NVD dataset. We examined the root cause for this discrepancy (>55% difference) and found 4,518 missing SARD test cases that caused it. These missing test cases are responsible for 12,244 missing SySeVR samples. To determine vulnerable lines in SySeVR samples, we need the source files from the corresponding SARD test cases. Therefore we excluded these samples from our dataset and thus, our findings on the arithmetic expression sample kind will be less reliable.

Sample kind	Non-vulnerable samples		Vulnerable samples	
	count	vulnerable lines	count	vulnerable lines
API function call	44797	33	13250	15483
Array usage	21941	6	10475	12007
Pointer usage	191000	77	26951	32701
Arithmetic expression	3706	0	910	1150
Total	261444	116	51586	61341

Figure 4.6: This figure shows the amount of matching vulnerable lines in non-vulnerable and vulnerable samples grouped by sample kind

4.2.2. VULNERABLE LINE LABELS

To label the lines of code in our samples we created a procedure that labels each line as either vulnerable or non-vulnerable. This procedure receives a SySeVR sample (the lower pane of figure 4.2), vulnerability descriptions (the metadata seen in figure 4.4), and the original SARD source file (the upper pane of figure 4.2).

The procedure parses the SySeVR sample to obtain the lines of code (lines two to fourteen in the example) and reads the vulnerable lines of code from the SARD source files (line 41 in the example). The SySeVR samples lines are compared to the vulnerable SARD source code lines and labeled accordingly (i.e., vulnerable when a match is found, non-vulnerable otherwise). In the SySeVR sample line thirteen matches SARD source file line 41 and is labeled as being vulnerable. To be able to compare SySeVR sample lines to vulnerable SARD source code lines we were required to remove white space as well as testing for four textual changes. These tests prevented mismatches in specific situations. The first test was the occurrence of block- and statement demarcations (i.e., curly brackets and semi-colons) in the lines of code from the SARD source files. We deleted these characters found at the end of lines and accepted a match resulting from this change. The second and third test checked for end-of-line and in-line (i.e // and /* ... */ respectively) comment in vulnerable lines. We deleted the comment from the vulnerable lines and accepted a match resulting from this change. The final test checked for functions spanning multiple lines (these are compressed to one line in SySeVR samples) by counting the opening and closing parenthesis in a line. When an unequal count occurred, we accepted the first line of the function call.

Figure 4.6 shows the amount of matching vulnerable lines grouped by sample kind. Altogether, we collect 51586 vulnerable samples containing 61341 vulnerable lines and 261444 non-vulnerable samples containing 116 vulnerable lines. We did not expect vulnerable lines in non-vulnerable samples, and therefore we inspected the samples for causes. Upon inspection, we saw these lines were marked as vulnerable because the SARD source file contained duplicate lines of code. Since we do not include non-vulnerable samples in our analysis, we did not further act on these samples.

4.2.3. DATASET VALIDATION

We validated our dataset after collecting the vulnerable and non-vulnerable samples. We observed 571 vulnerable slices which did not contain vulnerable lines of code. Upon inspection of the samples, we noticed two reasons for this absence. A large part of these samples originates in CWE-types that manifest without source code (for example, memory leaks). A smaller part of the samples did not include any vulnerable lines. We observed that these samples contained nested code blocks that did not contain data dependencies

Vulnerable origins	#samples	%	lrp-analysis	lrp-test	Parameters	
Pointer usage	26951	(52%)	21561	5390	lrp-analysis percentage	80%
API function call	13250	(26%)	10600	2650	lrp-test percentage	20%
Arithmetic expression	910	(2%)	728	182	vulnerable samples	51586
Array usage	10475	(20%)	8380	2095	non-vulnerable samples	261328
Non-vulnerable origins						
Pointer usage	190923	(73%)	152738	38185		
API function call	44765	(17%)	35812	8953		
Arithmetic expression	3706	(1%)	2965	741		
Array usage	21934	(8%)	17547	4387		
		Totals	250331	62583		

Figure 4.7: This figure shows the number of non-vulnerable and vulnerable samples divided between the analysis and the evaluation datasets.

and were skipped because the forward program slice procedure does not include control flow dependencies. Because the vulnerable lines were missing, it seemed like these samples were incorrectly classified. We did not further examine this and used the dataset as is in our analysis. We discuss this decision in chapter 8.1.

4.3. Model baseline performance

We establish a baseline model in two steps to compare and evaluate improvements. First, as our analysis in RQ2 and RQ3 is performed on our dataset, we required previously unseen data to support the validity of our findings. To obtain such unseen data, we have split our dataset into two parts. A larger part for the analysis and a smaller part for the evaluation. We observed some differences in the vulnerable and non-vulnerable origin proportions as shown in figure 4.7. For example, the "array usage" origin occurred more than twice as many relatively seen in the vulnerable samples as in the non-vulnerable. We preserved these proportions for validity in the analysis and evaluation datasets resulting in 250,331 analysis samples and 62,583 test samples. Second, we established the final baseline performance using the average precision metric created from pr-curves of these two datasets and the SySeVR test dataset. Figure 4.8 shows the computed pr-curves with their average precision and maximum F1 metric values. We observed two findings. First, the analysis and evaluation datasets showed higher average precision values than the SySeVR test dataset. This is expected as the training data for the model contains relatively more SARD samples than NVD samples, and therefore the classifier is more likely to correctly classify SARD samples. Second, the percentage of vulnerable samples is higher in our datasets than in the SySeVR test dataset (16.4% versus 12.4% respectively). As the SySeVR test dataset contains both SARD and NVD samples and our dataset contains only SARD samples, we deduced that SARD slices are more often labeled vulnerable. We did not further investigate this difference and accepted our model baseline performance.

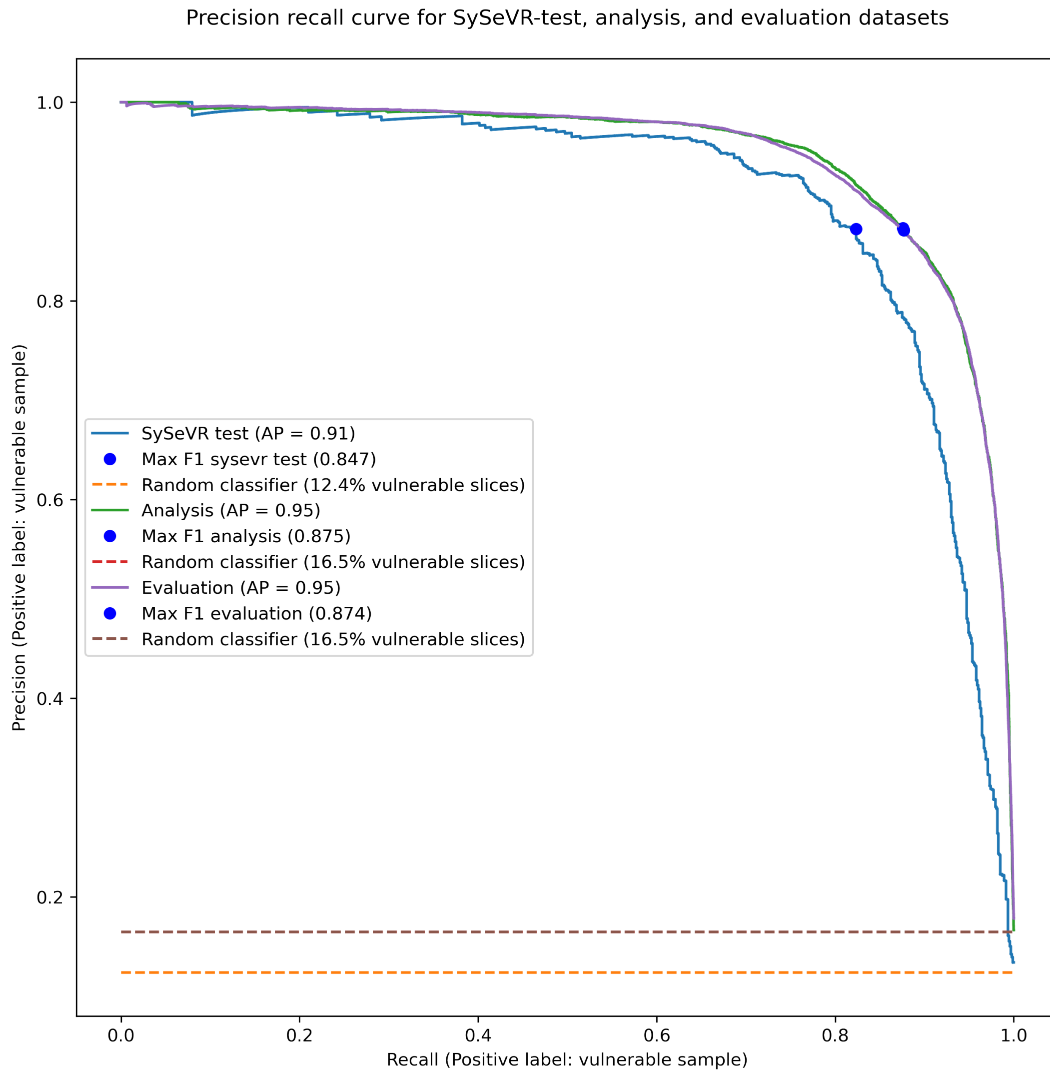


Figure 4.8: This figure shows the precision plotted against the recall, the average precision (AP), and the maximum F1 values for the SySeVR test, our analysis, and our evaluation datasets.

5

EXPLAINING SOFTWARE VULNERABILITY CLASSIFICATIONS

This chapter describes the explanation technique layer-wise relevance propagation (LRP) described by Bach et al. and Arras et al. which will be used to explain the classifications made by the SySeVR system and shows how we apply LRP to determine which parts of vulnerabilities are relevant in a deep learning setting [21, 3, 2].

5.1. EXPLAINING CLASSIFICATIONS

Our research goal is to improve the SySeVR system by explaining incorrect classifications. Such a classification is an incorrect result of the vulnerability detection model for a specific sample. The explanation thereof is the contribution (henceforth called relevance) of each sample feature towards this classification. We determine the relevance of features in a sample with the LRP technique described by Bach et al. and Arras et al. [3, 2].

5.1.1. LAYER-WISE RELEVANCE PROPAGATION

The LRP technique determines the relevance (expressed as a real number) of features towards a target class. Whereas a feature having a negative relevance value implies that it decreased the probability of the target class, a feature having a positive relevance value implies an increase of the probability of the target class. The relevance is determined by classifying a sample and then propagating the model output of the target class backward through the layers of a neural network. Each layer type (e.g., fully-connected or convolutional) has a specific transformation. An LRP explanation is executed with two movements through the network, a forward pass to set the neuron values and a backward pass to compute the explanation of the classification. It is seen as an computational efficient explanation technique compared to other commonly used explanation methods. In comparison, the explanation technique LIME requires the fitting of a linear classifier to generate an explanation for a single sample [1, 31]. In their studies, Arras et al. and Bach et al. showed that LRP can explain classifications in text and image classification tasks. Arras et al. introduced in their work on sentiment classification explanations transformations for the computations performed in an LSTM network. Arras et al. demonstrated that LRP can produce explanations of movie review sentiment classifications containing evidence agreeing or disagreeing with the classification. Finally, the LRP technique has the property that the total classification

output (of interest) is conserved in the explanation. Each step propagates its entire relevance (except for a small stabilizing term in the LRP equations) from layer to layer. As a result of these lossless transformation steps, the conservation of the relevance holds for the entire network. Therefore, LRP provides an explanation of classification in terms of its input.

To show how an explanation is created, we use an example vulnerability detection model. Figure 5.1 shows an example of a neural network.

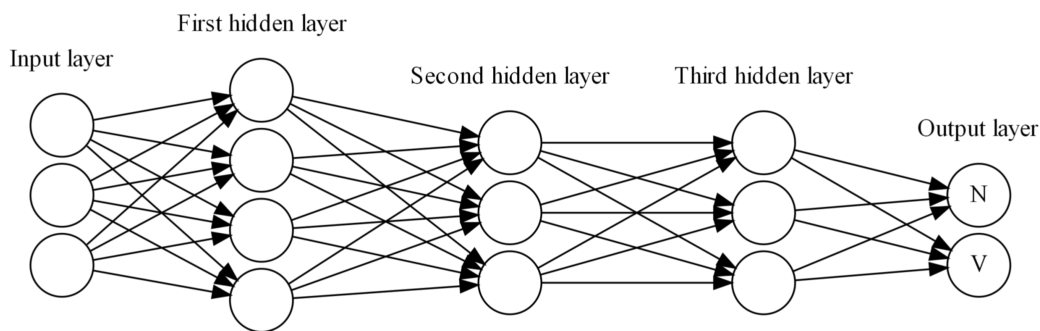


Figure 5.1: This figure shows an example deep neural network for vulnerability detection with an input layer, three successive hidden layers and an output layer.

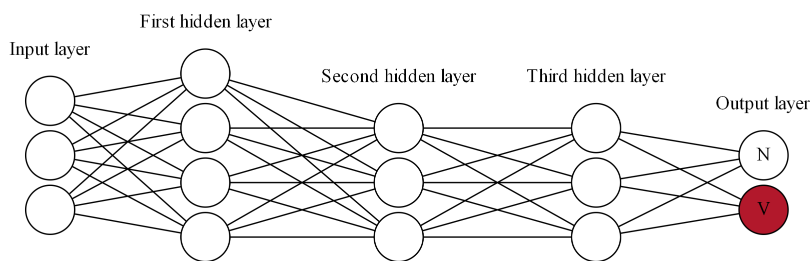


Figure 5.2: This figures shows which neuron is selected for explanation in our example vulnerability detection network

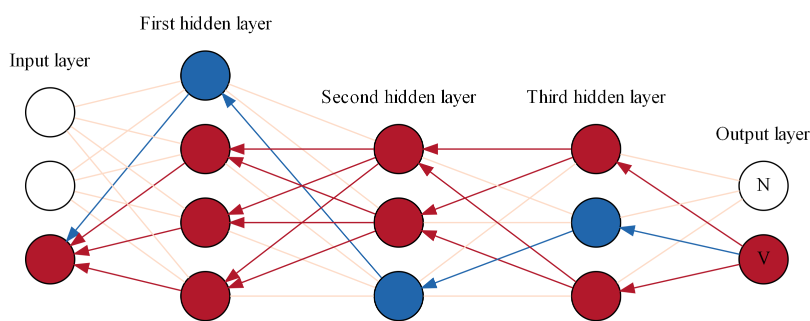


Figure 5.3: This figures shows how relevance flows backwards from the output layer to the input layers (i.e. the model features). The red and blue arrows indicate the positive or negative contribution respectively to the source neuron.

The output layer in this example is composed of two neurons that correspond to classes 'N' and 'V' (e.g., non-vulnerable and vulnerable in our example). The first step to generate an explanation is the forward pass, this computes the neuron values and classifies the sam-

ple. In figure 5.2 the red accentuated 'V' neuron shows the sample is being classified as vulnerable. The second step is choosing which target class to explain. As we are interested in the features contributing to vulnerabilities, we choose to explain the vulnerable case. The final step is the backward pass which steps backward through the model layers and assigns each neuron its relevance value. Figure 5.3 shows how positive and negative (red and blue respectively) relevance of the target class is redistributed from the output layer to the input layer. The redistribution of the relevance is performed layer by layer by sending messages from neuron to neuron. Bach et al. define these message as:

$$R_{i \leftarrow j}^{(l,l+1)},$$

where i is a neuron in layer l and j is a neuron in successive layer $l+1$ [3]. The $\leftarrow$ symbol shows the direction of the message. Using this definition, we can express the relevance of neurons (with the exception of the neurons in the output layer) as:

$$R_i^{(l)} = \sum_{\text{k: i is input for neuron k}} R_{i \leftarrow k}^{(l,l+1)}$$

For example, the relevance of the third neuron in the input layer shown in figure 5.3 can be expressed as:

$$R_3^{(1)} = R_{3 \leftarrow 1}^{(1,2)} + R_{3 \leftarrow 2}^{(1,2)} + R_{3 \leftarrow 3}^{(1,2)} + R_{3 \leftarrow 4}^{(1,2)}$$

5.1.2. VISUALIZING EXPLANATIONS

The features of our sample are vectors, and the relevance of each feature is expressed as a real number. To interpret these explanations as human beings intuitively, we convert the vectors to the corresponding source code tokens and visualize the relevance using a color scale.

Figure 5.4 shows the explanation of our running example (a stack-based buffer overflow) with a heatmap. The "Sample" column shows normalized tokens (tokens with their variable and functions names replaced with generic identifiers to improve the generalization of samples) with their relevance, and the "Slice" column shows the original slice tokens so we can easier interpret a sample. Since our vulnerability labels are assigned at line level, we group the tokens by line.

We observe some interesting facts. First, we see different relevance values for equal tokens (for example, in line eight, the number 50 has neutral relevance, but in line 13, a highly positive relevance). This is expected because the LSTM model uses previously (backward and forward) encountered token values to determine the current token value (i.e., it uses the recurrent state). Second, vulnerable line 12 shows no strong connection between the vulnerability in the sample and the positive relevance. Only the square brackets used for indexing the variable and the end-of-statement semicolon receive positive relevance. This indicates that the model (in this case) did not classify based on the actual vulnerability.

5.2. EXPLAINING SOFTWARE VULNERABILITY CLASSIFICATIONS

This section describes our modifications to the LRP procedure, how we answered the first research question and how we determined whether our initial hypothesis was valid. Figure 3.2 shows an overview of how we answered this research questions).

```
Slice: 283209
Header: 283209 65521/CWE121_Stack_Based_Buffer_Overflow__CWE806_char_alloca_loop_10.c dest 41
Ground truth: 1
Tensorflow model (output/act/label): 5.7/1.0/1
LRP model (output/act/label): 5.7/1.0/1
Heatmap:
```

Line #	Label	Sample	Slice
1	0	void func_0 ( )	void CWE121_Stack_Based_Buffer_Overflow__CWE806_char_alloca_loop_10_bad ( )
2	0	char * variable_0 ;	char * data ;
3	0	char * variable_1 = (char *) func_1 (100 * sizeof (char)) ;	char * dataBuffer = (char *) ALLOCA (100 * sizeof (char)) ;
4	0	variable_0 = variable_1 ;	data = dataBuffer ;
5	0	if (variable_2)	if (globalTrue)
6	0	memset (variable_0 , ' A ' , 100 - 1) ;	memset (data , ' A ' , 100 - 1) ;
7	0	variable_0 [100 - 1] = ' \ 0 ' ;	data [100 - 1] = ' \ 0 ' ;
8	0	char variable_3 [50] = " " ;	char dest [50] = " " ;
9	0	size_t variable_4 , variable_5 ;	size_t i , dataLen ;
10	0	variable_5 = strlen (variable_0) ;	dataLen = strlen (data) ;
11	0	for (variable_4 = 0 ; variable_4 < variable_5 ; variable_4 + +)	for (i = 0 ; i < dataLen ; i + +)
12	1	variable_3 [variable_4] = variable_0 [variable_4] ;	dest [i] = data [i] ;
13	0	variable_3 [50 - 1] = ' \ 0 ' ;	dest [50 - 1] = ' \ 0 ' ;

Figure 5.4: This figure shows an explanation of our running example. The relevancy of each token is visualized using colors with blue and red being negative and positive respectively. Non-vulnerable lines are labeled with a zero and vulnerable lines are labeled with a 1.

5.2.1. LRP IMPLEMENTATION

The classification type of the LRP model has been modified to allow for the explanation of binary classifications. Whereas the LRP model employed by Arras et al. is a multi-class classification task (5 output neurons), the TF model used by SySeVR system is a binary classification task (1 output neuron). Therefore the classification type of the LRP model was adapted. The most probable class is determined by applying the softmax function to the output neurons for the multi-class classification. This function normalizes the neuron values to a probability distribution (all neurons get values between zero and one and add up to one), with the highest value determining the classification. The logistic function applied to the single output neuron determines what the most probable class is in the binary classification. With an output value lower than 0.5, the classification is negative (non-vulnerable); otherwise, it is positive (vulnerable). The modifications to the LRP procedure were validated by hand on a small number of samples by checking that the classification score of the model was equal to the sum of the relevance assigned to the features.

We applied this validated LRP implementation to our dataset to obtain relevance values. We observed long run times with a limited number of samples in a test run. Because our average processing time was approximately 1000 samples per hour (e.g., 3.6s per sample), the expected total run-time for our complete dataset was 13 days (containing approximately 313K samples). We investigated whether this run-time could be lowered and observed that per sample the relevance calculation took much more time than the data-processing (loading and saving) of a sample.

We leveraged this discrepancy between the calculation and processing time by implementing a multi-process LRP implementation using the python multiprocessing [1] package. This multi-process LRP implementation analyzes samples simultaneously by distributing

[1]https://docs.python.org/3/library/multiprocessing.html

the processing of samples over multiple CPUs and thereby lowering the total running time. This optimized LRP implementation had an average processing time of 13K samples per hour (0.3s per sample) using 12 simultaneous processes (e.g. one proces per cpu-core).

Consequently, our total run-time was approximately 24h for our entire dataset when running on an HP Z820 workstation containing 64GB RAM, two Intel Xeon CPUs featuring six cores, a single 1TB solid-state hard drive, and a Nvidia RTX 2080ti GPU.

5.2.2. Line comparison statistic

The LRP procedure assigns relevance to each token in the sample, however to determine how much relevance the LRP procedure assigns to each line of code, we required a line comparison statistic. We calculated three comparison statistics, the mean, the maximum, and the median relevance per line. For example, in figure 5.5 we show these statistics for vulnerable line 12 in figure 5.4. We compared their relevance distributions in vulnerable and non-vulnerable lines. The left and right boxplots in figure 5.6 show that the mean and median line relevance distributions of vulnerable lines are overlapping and lower than the non-vulnerable lines. The middle boxplot shows that the vulnerable maximum line relevance distribution is overlapping but mostly higher than the non-vulnerable. As a result, the maximum criterion seems to distinguish better between vulnerable and non-vulnerable lines than the mean or median criteria.

Sample tokens	Slice tokens	Relevance
variable_3	dest	-0.229
[	[	0.32
variable_4	l	-0.29
]	]	0.538
=	=	0.229
variable_0	data	-0.581
[	[	0.449
variable_4	l	0.012
]	]	0.422
;	;	0.241
	Mean	0.1111
	Max	0.538
	Median	0.235

Figure 5.5: This figure shows the token relevance and the mean, maximum and median line statistics for vulnerable line 12 in the thesis example.

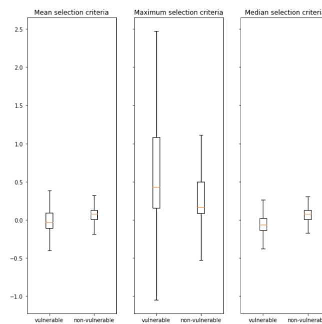


Figure 5.6: Boxplots for mean, maximum and median selection criteria broken down in vulnerable and non-vulnerable lines. The bulk of the vulnerable lines selected with the mean and median criteria have a relevance value lower than the non-vulnerable lines. The bulk of the vulnerable lines selected with the maximum selection criteria lies higher than the non-vulnerable lines. Outliers are not shown in the boxplots (i.e., the whiskers show 1.5 times the IQR)

Furthermore, to compare relevance values between samples, we require the values to

have the same semantic value. Each sample has its own relevance value distribution with a mean and standard deviation. Therefore, relevance values from different samples come from different distributions and cannot be compared directly. For example, a relevance value of one could signify a low relevance in one sample and a relatively much higher relevance value in another. For this reason, we performed standardization per sample by subtracting the mean sample relevance from sample relevance values and dividing the result by the relevance standard deviation). A standardized relevance value of one has the same meaning in different samples (i.e., the value is a single standard deviation above the mean relevance value in that sample). Standardizing the sample features allowed us to compare the relevance values.

5.2.3. Vulnerable line detection precision

In this subsection, we answer our first research question: "How precise are relevant parts of vulnerabilities detected using deep learning?". The line detection precision expresses how precise LRP can detect vulnerable lines in our dataset.

We obtain precision and recall metrics by comparing our line comparison statistic per line to different classification thresholds and counting the number of true positive, false positive, and false negative line classifications. These counts are then used to calculate the metrics at each classification threshold. We report these measurements with a precision-recall curve and average precision broken down into lines from false negative and true positive samples.

Findings

Figure 5.7 shows precision-recall (pr) curves with their average precision (AP), random classifier baselines and F1 metrics. The precision value is the ratio of ground-truth vulnerable lines to the detected vulnerable lines and expresses how accurate LRP detects the vulnerable lines. The recall value, on the other hand, is the ratio of truly vulnerable lines to all vulnerable lines and expresses how complete LRP detects the vulnerable lines. The average precision provides a summary value of the precision values at each threshold. The pr-curve is broken down into three vulnerable sample sets (TP, FN, and TP+FN) with their average precision shown in the legend. The highest F1 metrics values are shown as blue dots on the pr-curves, and the random classifier baselines are shown with dotted lines below them.

The figure illustrates our findings on vulnerable line detection precision. First, as the curves show higher pr- and AP-values than their baselines, the vulnerable line detection with LRP performs better than random line selection. Secondly, the precision levels from zero recall to about 0.05 recall show that LRP can detect a small part of the vulnerable lines with a precision of almost six times the random precision and the rest of the vulnerable lines slightly better than random line selection. Thirdly, the FN samples show a less steep decline in their pr-curve and a slightly higher AP (0.20 versus 0.17) compared to the TP samples. This indicates that, on average, the vulnerable lines in the FN samples have higher maximum relevance than those in the TP samples. Fourthly, the F1 metric for the FN samples (0.26) is slightly higher than the F1 metric of the TP samples (0.22), implying that the vulnerable line detection precision is slightly biased towards incorrect classifications. Altogether, we can conclude that the explanations of incorrectly classified vulnerable samples have higher vulnerable line detection precision than correctly classified vulnerabilities.

The value of these findings lies in the fact that they can predict how much vulnerable classified lines are likely to be vulnerable. For example, it can be used to guide a vulnerabil-

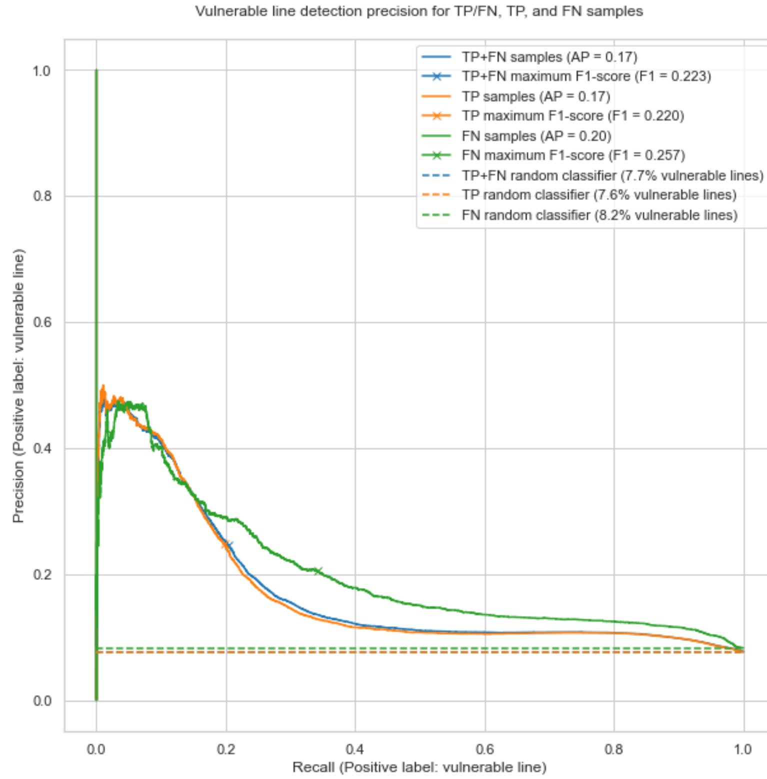


Figure 5.7: precision-recall curve line precision for TP+FN, TP, and FN samples based on maximum relevance selection.

ity analysis strategy. If the consequences of a vulnerable rule are hazardous, a low threshold can be chosen. Although more false positives will probably be encountered than using a higher threshold, more lines will be classified vulnerable with it. The reverse strategy is also possible. For example, a higher threshold can be chosen with a limited analysis capacity. The number of potential vulnerabilities to be analyzed and the number of false positives will probably be lower.

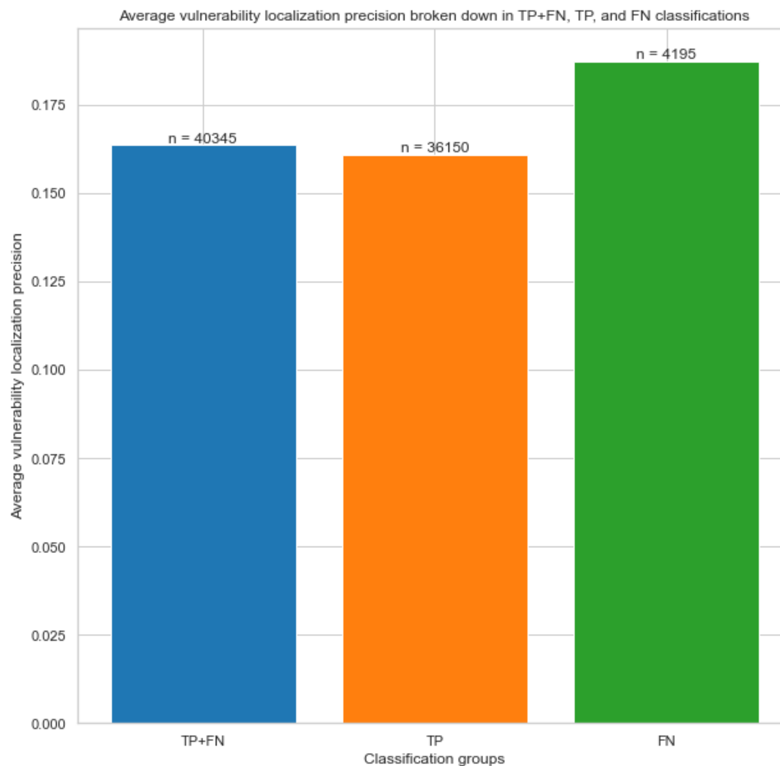


Figure 5.8: This figure shows the average localization precision in three sample groups.

5.2.4. VULNERABILITY LOCALIZATION PRECISION

In this paragraph, we extend the previous answer to our first research question: "How precise are relevant parts of vulnerabilities detected using deep learning?". The vulnerability localization precision determines how precise LRP locates vulnerabilities in slices. Towards this end, we measure the ratio of overlapping selected and vulnerable lines per sample using the intersection over union metric. Lines in samples are selected by comparing their line comparison statistic to the threshold which corresponds to the highest F1 score. We report the average precision broken down in vulnerable line count and the distribution broken down in false negative and true positive samples.

FINDINGS

Our findings on average localization precision are illustrated with several figures. First, the average localization precision in figure 5.8 shows that the false negative sample group has a slightly better average localization precision (0.19) than those in the true positive samples (0.16) and combined sample groups (0.16).

Secondly, the histogram in figure 5.9 shows three notable bins. Bin [0 – 0.6) contains only samples with zero precision, Bin [0.94 – 1.00] contains only samples with perfect precision, and bin [0.5 – 0.56) contains only samples with 0.5 IoU precision. We added the counts of other bins to appendix A.2.

Furthermore, in table 5.1 we observe that the TP (76.6%, 10.7%, and 8.0%) and FN (72.6%, 12.8%, and 8.4%) groups do not significantly differ (relatively) in these groups.

Although explanations of incorrectly classified vulnerable samples have a slightly higher vulnerability localization precision than correctly classified vulnerabilities, most vulnera-

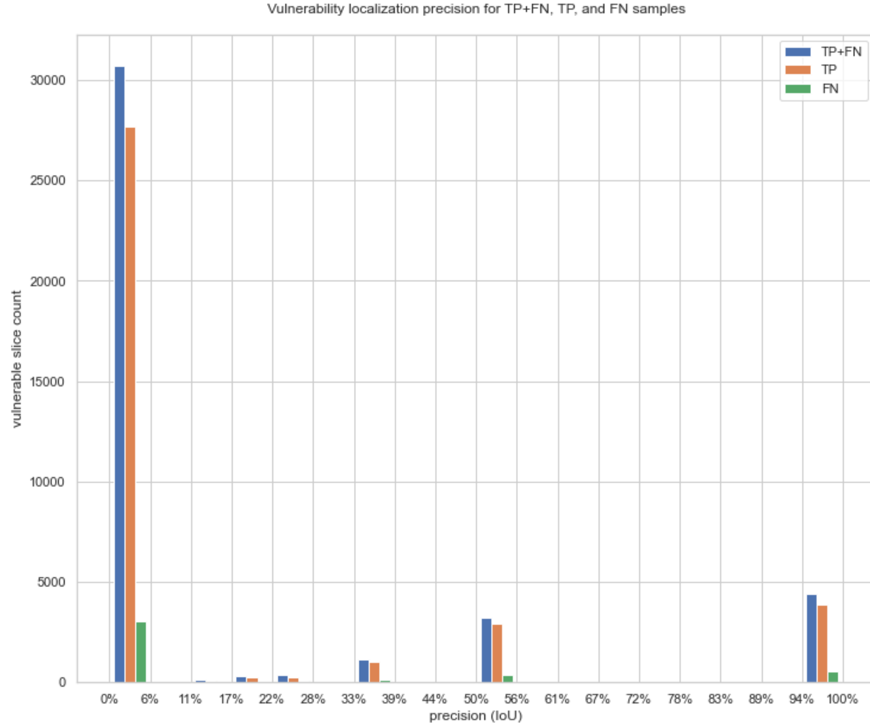


Figure 5.9: This figure shows the histogram of the localization precision in three sample groups.

	TP+FN		TP		FN	
IoU bins	#	%	#	%	#	%
[0.00 - 0.06)	30734	76,2%	27688	76,6%	3046	72,6%
[0.94 - 1.00]	4420	11,0%	3885	10,7%	535	12,8%
[0.50 - 0.56)	3249	8,1%	2898	8,0%	351	8,4%
[0.33 - 0.39)	1136	2,8%	1007	2,8%	129	3,1%
[0.22 - 0.28)	348	0,9%	278	0,8%	70	1,7%

Table 5.1: This figure shows Top 5 localization precision histogram bins in three sample groups.

bilities in the classification groups (TP: 76.6% and FN: 72.6% respectively) cannot be determined at all (zero IoU) using the line comparison statistic.

5.3. CONCLUSION

In conclusion, we have learned in this chapter that our model uses non-vulnerable lines more often than vulnerable lines in classifying vulnerable samples. This difference provides a strong argument not to limit our following analysis to vulnerable lines but instead use both non-vulnerable and vulnerable lines in order to determine which tokens have a negative impact on classification. It also raises a few questions. Are our labels of vulnerable lines genuinely correct, or do other lines also play a part in, or provide context for vulnerabilities? Is the model really learning to detect vulnerabilities in our samples, or is it over-fitted on the training dataset? We will elaborate on these questions in our discussion chapter 8.

6

Improving the SySeVR vulnerability detection system

6.1. Introduction

The goal of our experiments is to explore whether we can decrease incorrect classifications and thereby increase classification performance. Our first experiment, tries to decrease the number of false positive classifications, our second experiment tries to decrease the number of false negative classifications, and our third experiment tries to decrease the numbers of false positive and false negative samples. The first experiment is described in section 6.2.1, the second experiment in section 6.2.2, and the third experiment is described in section 6.2.3. Whereas the method of the first three experiments is to remove tokens from samples before classification, the method of our fourth experiment removes the tokens from samples before training. In this final experiment, described in section 6.3, we use the results from our previous experiments to determine which tokens to remove.

6.2. Improve the classification performance during inference

This sections describes our experiments towards answering our second research question: "How can we improve the classification performance using the relevance of features contributing to incorrect classifications during inference?".

6.2.1. Remove tokens with high relevance to prevent FP classifications

The goal of the first experiment is to decrease the number of false positive classifications using the token relevance. We hypothesize that:

> Highly relevant tokens in non-vulnerable samples contribute strongly to false positive classifications. Therefore, filtering these tokens during classification will produce a lower number of false positive classifications than during classification without filtering.

To verify this hypothesis, we will detect vulnerabilities in token-filtered samples and measure the classification results.

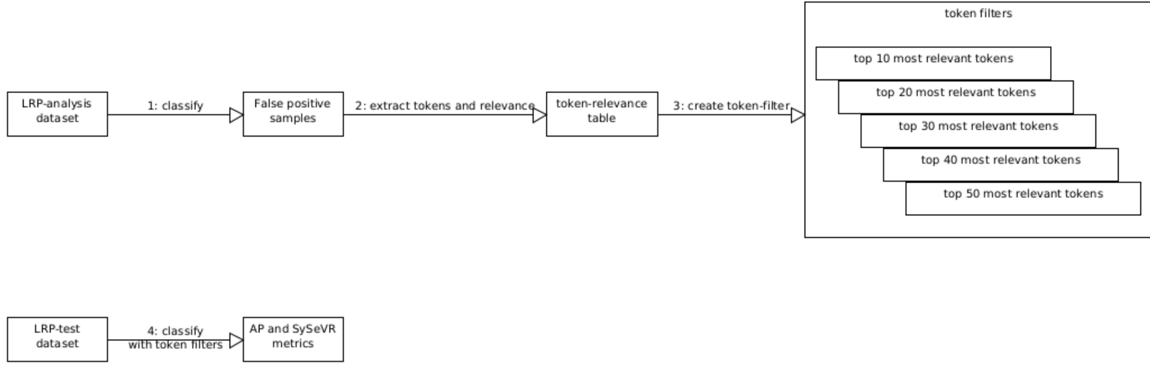


Figure 6.1: Construction and application of token filters

	Predicted vulnerable	Predicted non-vulnerable
Vulnerable	36150	4195
Non-vulnerable	6145	199978

Table 6.1: Binary classification results (confusion-matrix) of the LRP model on the LRP-analysis dataset.

The required steps to determine highly relevant tokens and filter these during classifications can be seen in figure 6.1.

Our first step is to determine the false positives samples in our lrp-analysis dataset. The next step is extracting the tokens and their relevance from these samples (step 2). In the third step, we sort the tokens on relevance and create five sets with the most relevant tokens (step 3). Finally, we classify our lrp-test dataset using the token-filters (step 4) and measure the average precision and the SySeVR metrics (see paragraph 4.1.3 for the SySeVR metrics).

Our lrp-analysis dataset contains 246,468 samples in total. In table 6.1 can be seen that 6,145 non-vulnerable samples are predicted vulnerable by our LRP model (i.e., the false positive samples). We extracted 694,477 tokens from these false positive samples containing 893 unique tokens.

We do not know how many tokens we should filter. Therefore, we start with five sets to determine an optimal number. The first set contains the top-10 most relevant tokens, the second contains the top-20, the third the top-30, the fourth the top-40, and the fifth set contains the top-50 tokens (see appendix B for the filter sets). We select unique tokens as they occur multiple times in the list.

To filter the tokens during classification, we alter our classifier program in two ways. We implement the token-filters as python set-containers for efficient O(1) lookup of tokens and add the sets to our prediction program. Furthermore, we compare sample tokens to the tokens in our filter list before classification. When a match is found, we remove the token from the sample that has to be classified.

To test whether the filters show an improvement, we classify unseen data, the lrp-test dataset, and measure the changes in classification results and metrics. Figure 6.2 shows that our filtered classifications have higher number of false positives compared to the baseline. The FP-10 filter has 53 (+ 3.5%), FP-20 has 258 (+17.0%), FP-30 has 116 (+7.6%), FP-40 has 440 (+29%), and FP-50 has 9864 (+ 650.2 %) more respectively.

Furthermore, a decrease of average precision in our filtered classifications with regard

label	LRP	FP-10	FP-20	FP-30	FP-40	FP-50
fn	1145	1245	1346	1639	3469	3270
fp	1517	1570	1775	1633	1957	11381
tn	50749	50696	50491	50633	50309	40885
tp	9172	9072	8971	8678	6848	7047

Table 6.2: This table shows the lrp-test dataset classification results computed after classification with highly relevant token filters

to the baseline can be observed in figure 6.2. The average precision of FP-10 decreases with 0.6%, FP-20 with 1.7%, FP-30 with 2.5%, FP-40 with 15.0%, en FP-50 with 57.2% respectively (see appendix C.1 for the details).

We do observe some notable differences in the metric values. While FP-10, FP-20, and FP-30 show a slightly higher or equal recall than precision, the FP-40 and FP-50 have more distinction in their values. FP-40 has lower recall than precision due to the significant increase of false negatives (+203%). FP-50, on the other hand, has a higher recall than precision. This difference can be attributed to the large increase of false positives (+650%). In this quantitative experiment, we did not further analyze which tokens from the FP-40 and FP-50 filters caused these significant changes.

We conclude from these results that removing relevant tokens from samples before classification does not lower but instead increases the number of false positive classifications and that our experiment hypothesis is rejected.

6.2.2. Remove tokens with low relevance to decrease FN classifications

Our second experiment aims to decrease the number of false negative classifications using the token relevance. We hypothesize that:

> Tokens with low relevance in vulnerable samples contribute strongly to false negative classifications. Therefore, filtering these tokens during classification will produce a lower number of false negative classifications than during classification without filtering.

To verify this hypothesis, we will detect vulnerabilities in token-filtered samples and measure the classification results.

This experiment follows the previous experiment except for the following three differences. Whereas the previous experiment created filters containing highly relevant tokens, this experiment uses the least relevant tokens (i.e., the tokens that carry the most weight towards a non-vulnerable classification). We extracted 467,628 tokens from 4,195 false negative samples containing 839 unique tokens. The first set contains the top-10 least relevant tokens, the second contains the top-20, the third the top-30, the fourth the top-40, and the fifth set contains the top-50 tokens (see appendix B for the filter sets).

Figure 6.3 shows that our filtered classifications have higher number of false negatives compared to the baseline. The FN-10 filter has 1249 (+109.1%), FN-20 has 2446 (+213.6%), FN-30 has 4757 (+415.5%), FN-40 has 4966 (+433.7%), and FN-50 has 6522 (+569.6 %) more respectively.

Furthermore, a decrease of average precision in our filtered classifications with regard

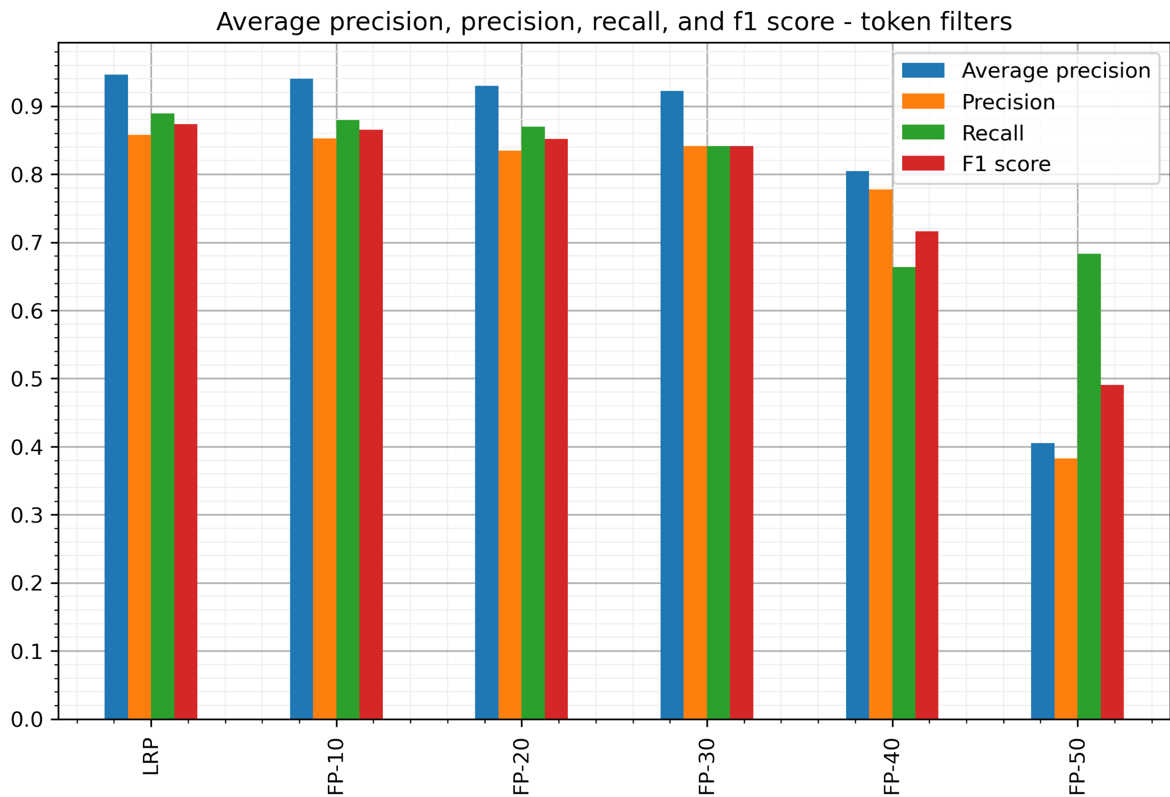


Figure 6.2: This figure shows the lrp-test classification metrics after classification with highly relevant token filters.

label	LRP	FN-10	FN-20	FN-30	FN-40	FN-50
fn	1145	2394	3591	5902	6111	7667
fp	1517	5471	9714	6425	5996	3932
tn	50749	46795	42552	45841	46270	48334
tp	9172	7923	6726	4415	4206	2650

Table 6.3: This table shows the lrp-test dataset classification results computed after classification with the least relevant token filters.

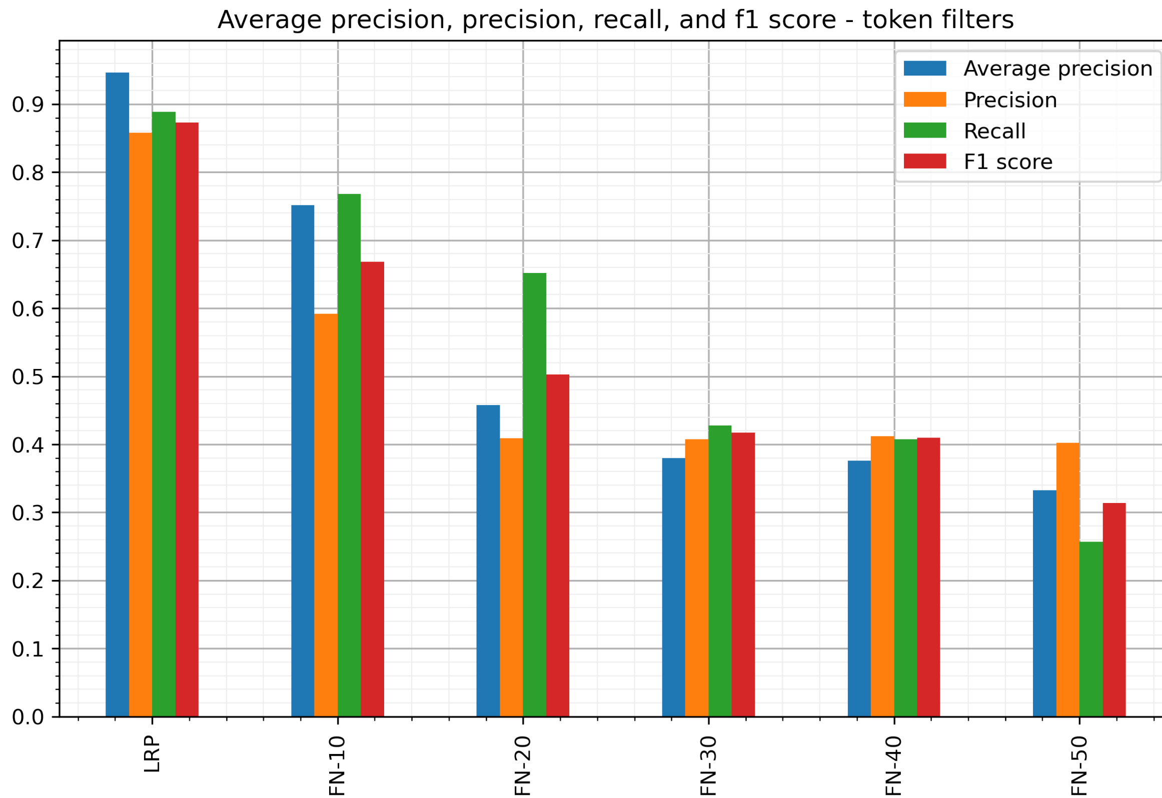


Figure 6.3: This figure shows the lrp-test classification metrics after classification with the least relevant token filters.

to the baseline can be observed in figure 6.3. The average precision of FN-10 decreases with 20.6%, FN-20 with 51.6%, FN-30 with 59.9%, FN-40 with 60.2%, en FN-50 with 64.9% respectively.

We observe some differences in the results of our experiment. The precision values in filters FN-10 and FN-20 decline more steeply than the recall values. This can be attributed to the relatively larger change in FP (-260.6% and -540.3%) than the change in FN samples (109.1% and 213.6%). Whereas the recall values drop in FN-30, FN-40, and FN-50, the precision values stay at the same level. This corresponds with the decrease of FP samples in these three sets. Although these sets contain tokens that improve the classification of non-vulnerable samples, these tokens also worsen the classification of vulnerable samples. We did not investigate which tokens were responsible for the improvement and noted this in our recommendations section in paragraph 9.

We note three differences when comparing the filters and results with our previous experiment (FP-experiment). The filters from our previous experiment contain relatively more normalized tokens (23 out of 50 tokens) than the FN filters (13 out of 50 tokens). Whereas the FP-experiment metrics decline in the FP-40 filter, the FN-experiment metrics decline immediately in the first (FN-10) filter. The change in FN samples in the FN experiment is more pronounced than the change in FP samples in the FP experiment.

We conclude from these results that removing the least relevant tokens from samples before classification does not lower but instead increases the number of false negative classifications and that our experiment hypothesis is rejected.

token filter	token count
10	19
20	39
30	56
40	72
50	90

Table 6.4: Token counts of FP-FN filter-sets

label	LRP	FP-FN-10	FP-FN-20	FP-FN-30	FP-FN-40	FP-FN-50
fn	1145	2387	3944	6449	7306	8320
fp	1517	5488	9403	5345	3470	4083
tn	50749	46778	42863	46921	48796	48183
tp	9172	7930	6373	3868	3011	1997

Table 6.5: This table shows the lrp-test dataset classification results computed after classification with highly relevant and least relevant token filters.

6.2.3. Combining FP/FN improvements

Our third experiment aims to decrease the number of incorrect (both false negative and false positive) classifications using the token relevance. We hypothesize that:

> The most and least relevant tokens in samples contribute strongly to false positive and false negative classifications respectively. Therefore, filtering these tokens during classification will produce a lower number of false positive and false negative classifications than during classification without filtering.

To verify this hypothesis, we will detect vulnerabilities in token-filtered samples and measure the classification results.

This experiment follows the same steps as the previous experiments except for the construction of the filters. The filters are constructed as a union between the filter sets containing the same number of tokens (e.g., the union of the top-10 relevant and top-10 least relevant). Due to this union, filter sets containing the same tokens (e.g., FP-10 and FN-10 both contain token "."), contain less tokens than the sum of tokens in the individual filter sets. See table 6.4 for the token-filter counts.

Table 6.5 shows that our filtered classifications performs worse than the baseline. The FP-FN-10 filter has 3971 (+261.8%) FP and 1242 (+108.5 %) FN, FP-FN-20 has 7886 (+519.8%) FP and 2799 (+244.5%) FN, FP-FN-30 has 3828 (+252.3%) FP and 5304 (+463.2%) FN, FP-FN-40 has 1953 (+128.7%) FP and 6161 (+538.1.7%) FN, and FP-FN-50 has 2566 (+ 169.2 %) FP and 7175 (+626.6%) FN more respectively. (onleesbaar, omzetten naar tabel?)

Furthermore, we see in figure 6.4 a decrease of average precision in our filtered classifications with regard to the baseline. The average precision of FP-FN-10 decreases with 20.5%, FP-FN-20 with 53.7%, FP-FN-30 with 61.3%, FP-FN-40 with 61.3%, en FP-FN-50 with 70.8% respectively.

We observe some differences in the results of our experiment. Whereas precision values in the FP-FN-10 and FP-FN-20 filter sets are lower than the recall value, this is reversed in the other three filter sets. The reason for this difference is that the latter sets have fewer FP samples than the former sets.

Combining the filters yields results worse than both FP and FN experiments. This trend

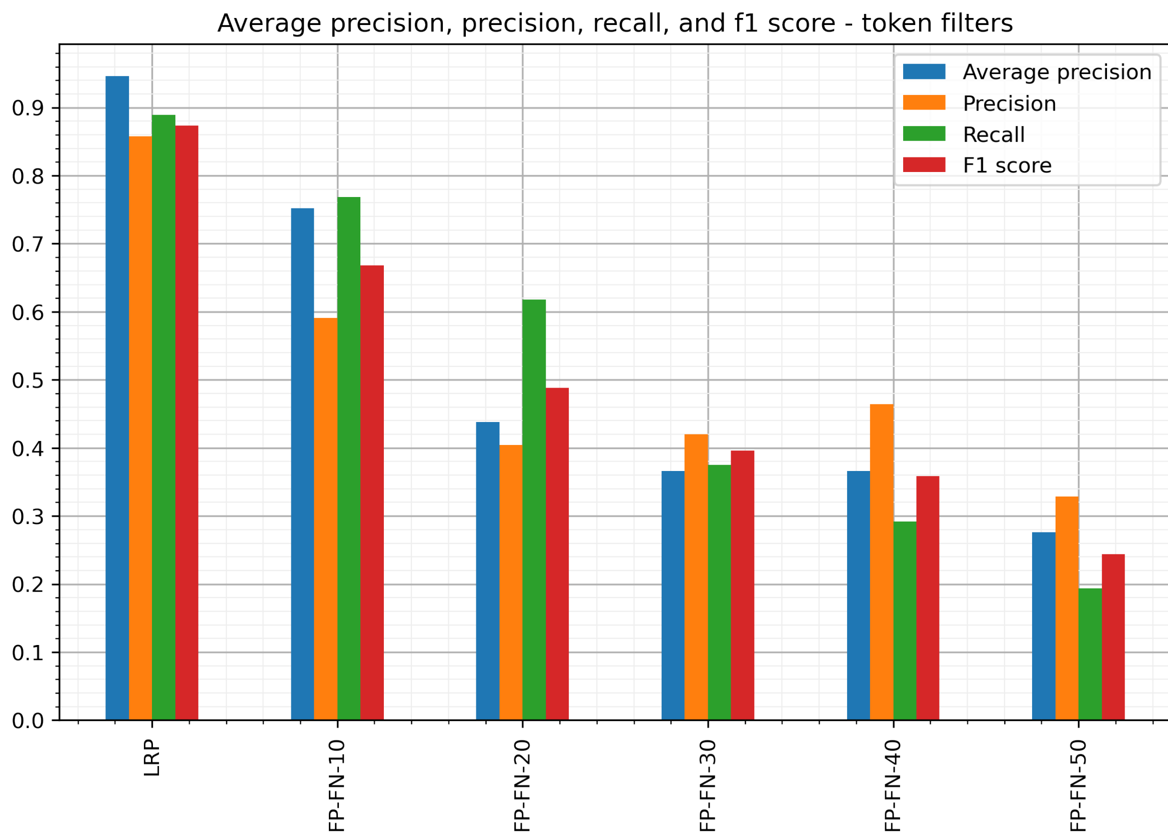


Figure 6.4: This figure shows the lrp-test classification metrics after classification with highly relevant and least relevant token filters.

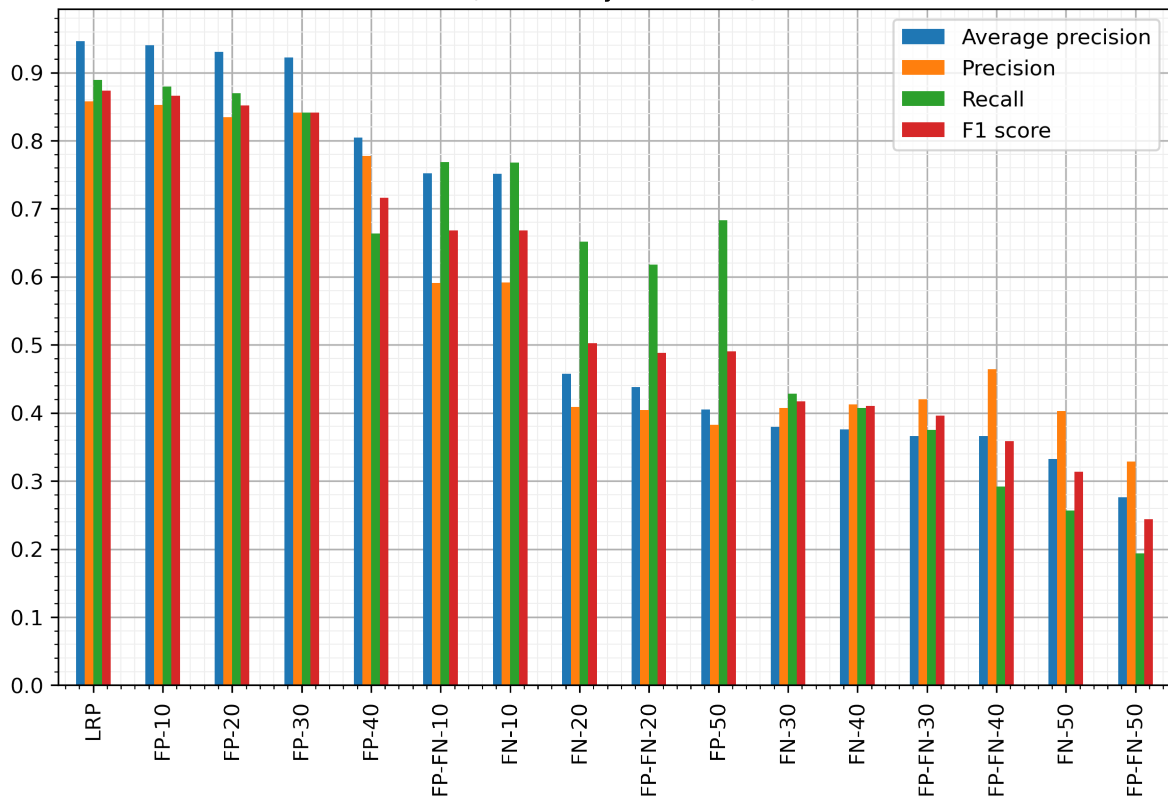


Table 6.6: This table shows the comparison of the lrp-test dataset classification metrics computed after classification with all filters. To compare our filter sets we sort the metrics by average precision.

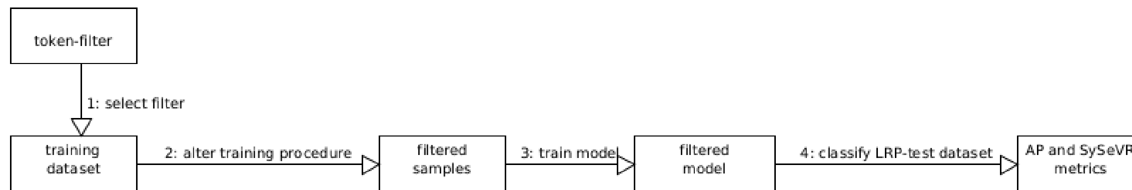


Figure 6.5: This figure shows the steps needed to train a new model and to classify our lrp-test dataset

can be observed in figure 6.6. This figure, sorted on average precision, shows that the FP-FN filters have lower AP than the FN filters, except for FP-FN-10, which is slightly higher than FN-10.

We conclude from these results that removing the highest and least relevant tokens from samples before classification does not lower but instead increases the number of incorrect classifications and that our experiment hypothesis is rejected.

6.3. Improve classification performance during training

This sections describes our experiment towards answering our third research question: "How can we improve the classification performance using the relevance of features contributing to incorrect classifications during training?".

The goal of our final experiment is to decrease the number of incorrect classifications by changing the model training procedure. We choose a token filter and test if filtering tokens in samples before training produce a lower number of incorrect classifications (of the filter category) than classifications produced with the model trained on unfiltered input. We detect vulnerabilities in our lrp-test dataset and measure the classification results to verify this hypothesis.

The steps needed to train a new model and classify our samples can be seen in figure 6.5. Our first step is to determine which token filter is used in this experiment. The next step is to filter samples in our training procedure. The third step is training our model with the new procedure on our training dataset. Finally, we classify our lrp-test dataset using the newly trained model and measure the average precision and the SySeVR metrics.

To select a token filter in this experiment, we review the results from our previous experiments. Although the FP-10 results show a slightly lower average precision than the LRP results (0.9405 versus 0.9460), we see in figure 6.6 that this filter has the highest average precision amongst our filters. Therefore we choose this filter to change our training procedure.

The goal of our final experiment is to decrease the number of false positive classifications by changing the model training procedure. We hypothesize that:

> Highly relevant tokens in non-vulnerable samples contribute strongly to false positive classifications. Therefore, a model trained on samples without these tokens will produce a lower number of false positive classifications than a model trained on samples with these tokens

We will detect vulnerabilities in our lrp-test dataset and measure the classification results to verify this hypothesis.

We made two changes to our training program to apply our token filter to the sam-

label	SySeVR	SySeVR FP-10	LRP	LRP FP-10
fn	141	173	1145	1596
fp	158	112	1517	1075
tn	6413	6459	50749	51191
tp	788	756	9172	8721

Table 6.7: This table shows the training and lrp-test datasets classification results from classification with our first model and the model trained on samples filtered with the FP-10 filter.

ples. First, the training data in our model training program is implemented as a Tensorflow dataset, and we did not find a way to filter our training data using this implementation. Therefore we converted the training data to NumPy arrays, an input array with samples, and another array containing the ground truths. Secondly, we filtered the samples in the former array and used both in our training procedure.

The classification results of our retrained model have changed when compared to the baselines. In table 6.7 we show the results of our test and lrp-test datasets (column pairs SySeVR/SySeVR FP-10 and LRP/LRP FP-10). Whereas we see an almost equal percentage decrease of false positive samples in both datasets (-29.11% and -29.14%), we see two different percentage increases of false negative samples (+22.7% versus +39.4%).

Furthermore, we see in figure 6.6 minor changes in the average precision metric, the test dataset gains 0.002% and the LRP-test dataset loses 0.005% with regard to their baselines. Whereas precision values in both test and lrp-test datasets have increased (4.6% and 3.8%), the recall values have decreased (4.1% and 4.9%). We conclude from the classification results that a model trained on samples without highly relevant tokens has a lower number of false positive classifications than a model trained on samples with highly relevant tokens and that our experiment hypothesis is accepted.

6.4. Conclusion

Our experiments towards answering RQ2 have shown that filtering the most or least relevant tokens from samples before classification does not improve the SySeVR vulnerability detection performance. This contrasts with our experiment towards answering RQ3 that shows , when the model is trained on samples without highly relevant samples, the detection of non-vulnerable samples improves in the SySeVR approach.

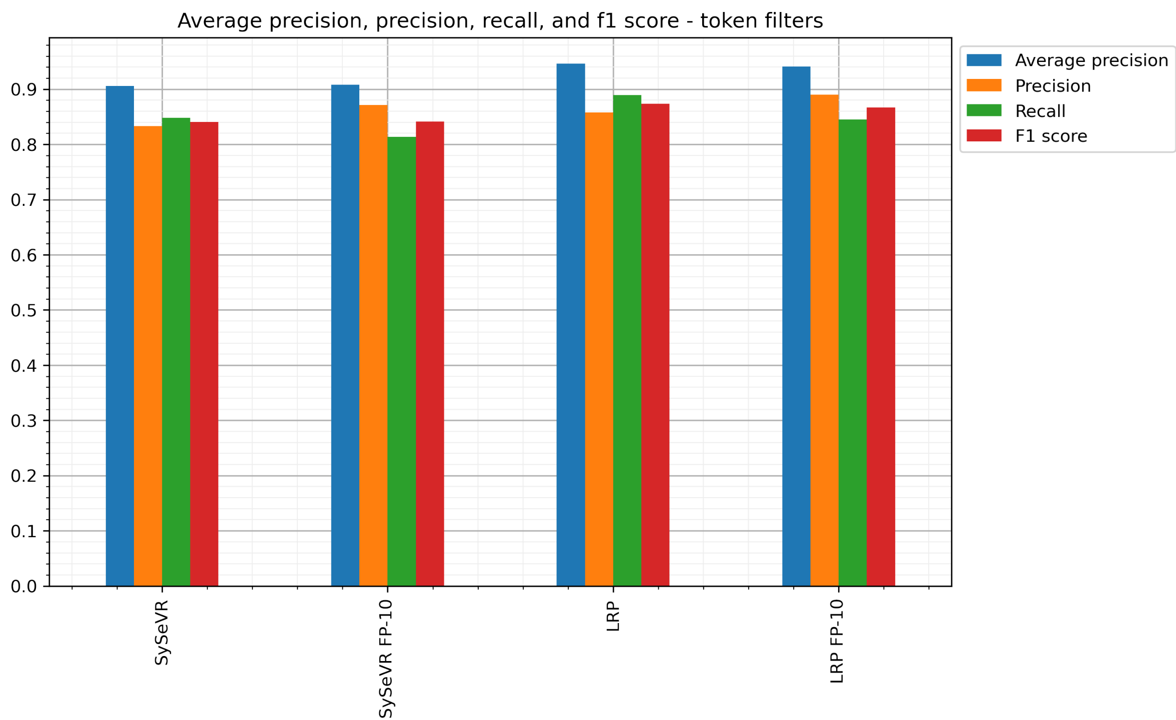


Figure 6.6: This figure shows the training and lrp-test datasets classification metrics from classification with our first model and the model trained on samples filtered with the FP-10 filter.

7
Related work

Since the popularity of deep neural networks, attention has also been given to explaining their classifications. Examples of this are highlighting the pixels in an image that contribute to object detection or showing which words are relevant in movie review classifications [3, 2]. However, our research does not focus on these classification tasks but on the intersection of explaining classifications and software vulnerability detection. It uses the research of Bach and Arras on the explanations of classifications and the research into software vulnerability detection using deep learning from Li et al. [3, 2, 21]. The research by Warnecke et al. and the research by Guo et al. focus on the same intersection[47, 13]. The former study yielded a new method, LEMNA, that determines which features have contributed strongly to classifications within the software security domain. The latter study defines criteria for comparing vulnerabilities detection methods (including LEMNA).

The work of Guo et al. has yielded a model agnostic (i.e., black-box) explanation method (LEMNA) that focuses on explaining classifications in the security domain [13]. Their work is inspired by the popular LIME method of Ribeiro et al. (see 2.4) but argues that the LIME method is not well suited to explain classifications of software security tasks. They state that in these tasks, features often have dependencies (e.g., consecutive tokens in source code) that are not taken into account by the LIME method. To overcome this limitation, they apply the fused-lasso technique of Tibshirani et al. to create a simple explicable linear model [42, 43]. This technique has two advantages in fitting the linear model. It reduces the number of coefficients and reduces the distance between successive coefficients. They evaluate their method with three tests. The first test removes the parts from a sample having the best explanation according to their method. In the second test, they augment an image from an opposite class with these parts, and in the final test, they construct random samples containing only these parts. After adapting the samples, Guo et al. measure the change in accuracy.

Our work differs in several ways from that of Guo et al. First, where we use the LRP method in our research, which uses the weights and structure of the model (white box), the work of Guo et al. uses the LEMNA method in which only the input and output of the model is used (black box). Second, where we use a bi-directional recurrent network (LSTM) as the subject, Guo et al. use both a bi-directional recurrent network (RNN) and a layered fully connected neural network as subjects for its classification task. Finally, where our evaluation of the LRP method determines accuracy using ground truth labels of the vulnerable

parts, the evaluation of the LEMNA method determines accuracy using labels from explanation generated by the LEMNA method itself.

The research of Warnecke et al. answers the question of how to choose an explanation method in the field of software vulnerability detection [47]. It specifies five criteria (accuracy, sparsity, completeness, stability, and efficiency) for explanation methods to detect software vulnerabilities successfully. These criteria are measured in six explanation methods applied to four current software security systems, and the results are compared to each other.

Our research differs from that of Warnecke et al. in several ways. Their accuracy property (descriptive accuracy) removes the most relevant tokens from samples and measures the change in classification accuracy. This is in contrast to our measurements, where we determine the accuracy of LRP using ground truth labels of the vulnerable parts. Their distribution characteristic (descriptive sparsity) measures whether the relevant parts of explanations are limited to a small group. We measure to what extent the relevant features occur in the explanations. Another difference is seen in the focus of their research. Whereas they focus on comparing their criteria in six explanation methods and providing guidelines for selecting an appropriate explanation method, our research focuses on measuring how precise vulnerable parts in samples are used in a vulnerability detection system and whether the explanations can be used to improve the classification performance thereof.

The similarity between our research and the research by Warnecke et al. is that both apply the LRP method to comparable software vulnerability detection systems. We use the SySeVR system. They use a predecessor, the Vuldeepecker system. The models of both systems have the same architecture and have learned from samples from the same (SARD) dataset to detect vulnerabilities. The difference between the two systems is that the Vuldeepecker system uses data dependency in slicing the samples and that the SySeVR system also applies control-flow dependency. As in our findings, Warnecke et al. observe in classifications of their model that relevance is assigned to parts which seem unrelated to task of vulnerability detection (for example, semicolons or brackets).

Comparing our findings to the relevant findings of Guo et al. and Warnecke et al. we observe some differences. Whereas Guo et al. report improvements in the test results when comparing their LEMNA method to the LIME method and a random baseline, Warnecke et al. observe that the LEMNA method has consistent lower accuracy on all four classification topics when compared to LIME. Warnecke et al. also report that black box explanation methods produce less accurate and sparse explanations when compared to explanations produced by white box explanation methods. Both Guo et al. and Warnecke report that the accuracy of predictions quickly declines when removing highly relevant tokens from samples. Hence they show that explanation methods can determine which features are the most relevant in classifications. While our results on removing the most relevant tokens from samples (described in section 6.2.1) also finds this relation (although less pronounced), our research also compares the features to the ground truth relevance and shows (in paragraphs 5.2.3 and 5.2.4) that our model uses non-vulnerable lines more often than vulnerable lines in classifying vulnerable samples.

8
Discussion

The objective of our research was to improve a software vulnerability detection approach using explanations of incorrect classifications. In this chapter, we interpret the results of our work, place the relevance of the results in the context of software vulnerability detection, and discuss the limitations of the research.

8.1. Explaining software vulnerability classifications

To measure how precise relevant parts of vulnerabilities are detected, we compared actual vulnerable lines to suspected vulnerable lines according to the LRP method. In section 5.2.3 we compared the line relevance distribution of line selection criteria. We determined that the maximum criterion was likely to distinguish between vulnerable and non-vulnerable lines better than the mean or median criteria. Selecting vulnerable lines using this maximum relevance criteria yields an average precision which is twice as good as selecting randomly (seventeen percent versus eight percent average precision, respectively).

Although we hypothesized that false negative (FN) samples in our dataset would have less precise explanations than true positive (TP) ones, the data (see figure 5.7) on precision showed a reversed relation. Our FN samples have a slightly higher average precision when compared to the TP samples (twenty percent and seventeen percent AP, respectively). Also, the harmonic mean of the precision and recall values (F1-metric) in the FN samples (0.26) is higher than in the TP samples F1-metric (0.22). This indicates that, on average, the vulnerable lines in FN samples have higher maximum relevance than those in the TP samples. Therefore, the vulnerable line detection precision is slightly biased towards FN samples when using the maximum F1-metric.

In addition to determining how precise LRP can detect vulnerable lines in the entire dataset, we also measured how precise the relevant parts (i.e., the vulnerable lines) in individual samples can be detected using LRP. In section 5.2.4 is shown that, on average, sixteen percent of the actual vulnerable lines are selected and that in the majority of vulnerabilities (76.2%), the vulnerable lines cannot be determined at all. Moreover, when the data on vulnerable samples are broken down into classification groups, we discerned that the FN group has a slightly higher average precision than the TP group (0.19 versus 0.16). This is in line with our previous results on the differences between FN and TP samples.

Showing that counter-intuitive, relevant parts of vulnerabilities detected using the SySeVR system are less likely to be used than the irrelevant ones raises some questions.

Could our labels of vulnerable lines be imprecise, thereby lowering the precision metrics? Some vulnerabilities could either not be labeled in our dataset because they did not manifest in vulnerable lines of code (e.g., memory or resource leaks) or because the sample creation method did not include the vulnerable lines. These missing labels were only seen in a small (571 out of 51586) part of the samples.

Also, because the C/C++ samples originate from the SARD dataset, which has commonly been used in studies, we assumed that vulnerable lines are described correctly. However, the vulnerable lines described in this dataset might not completely describe the vulnerabilities. For example, only line 41 is marked as vulnerable in the SARD testcase in figure 4.2 (paragraph 4.1.2). Although this line contains the actual writing outside of the buffer, there are multiple places in the example that make this out of bounds writing possible. One could argue that the destination buffer is created to small in line 35 or that the length of the for-loop is calculated using the wrong buffer at line 37.

Could the excellent vulnerability detection performance result from over-fitting on the training data? We followed the SySeVR training procedure described by Li et al. to train our deep learning network. In the left plot in figure 8.1 it can be observed that the loss of the models is becoming lower in the training dataset than in the validation dataset around epoch 6. Although this implied that the model became better at predictions in the training data, we observed a declining trend in the validation data loss, which indicated that the model was still improving on the validation data. However, starting at epoch 17, we saw inclining trend lines in the validation data, suggesting that the models were starting to overfit on the training data. The training of the models has been stopped at epoch 19 with small increases of the validation loss. This suggests that models were not over-fitted much at that point.

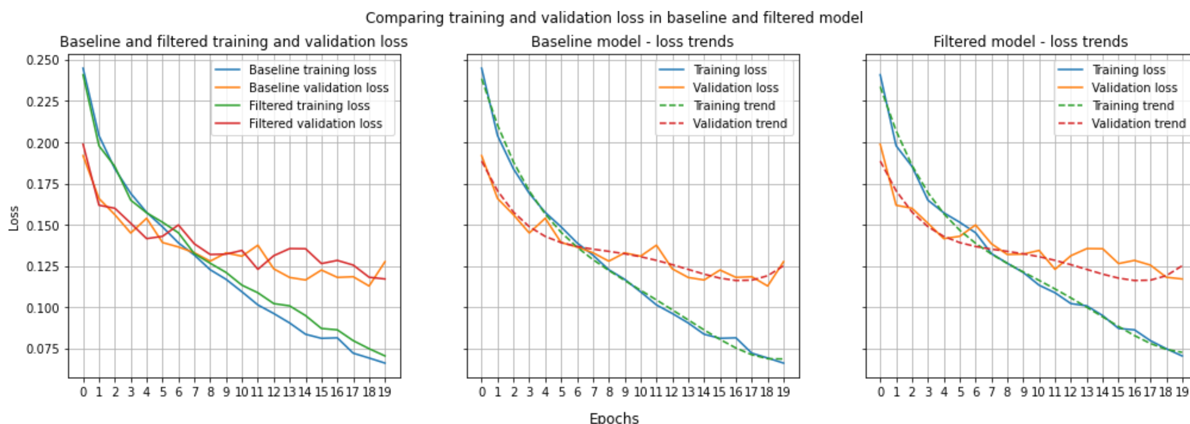


Figure 8.1: These figures show training and validation loss of two models. The recreated SySeVR model (i.e., the baseline model) trained on the SySeVR dataset, and the model trained on the SySeVR dataset with our token filter applied (i.e., the filtered model). The left figure compares the training and validation loss of the baseline model and the improved model. The middle figure shows the training and validation loss of the baseline model and their regression lines. The right figure shows the training and validation loss of the filtered model and their regression lines. We see in the middle and right figures inclining trend lines (polynomial regression with degree 4) starting at epoch 17 in the validation data of both models which indicates that the models were starting to over-fit on the training data.

Although we do not have definitive answers to these questions, our results are still relevant to the problem of vulnerability detection because they urge us to be critical in assessing the value of deep learning models in this context. In this case, we showed that the lines

we labeled as being vulnerable (according to the SARD metadata) do not play a significant role in determining whether a sample is vulnerable. Because the SySeVR system detects vulnerabilities without these lines, it is uncertain whether it can find vulnerabilities when applied in a different setting. For example, in the detection of vulnerabilities obtained from other, non-synthetic, source code. Furthermore, we showed that in our case, using the SySeVR model and SARD dataset, the explanations of vulnerabilities did not seem to help resolve the vulnerabilities themselves.

8.2. Improving the SySeVR vulnerability detection system

To determine whether we could use explanations of incorrect classifications to improve the SySeVR system we performed three experiments towards answering research questions RQ2 and one experiment towards answering RQ3. Each experiment aimed to measure whether removing the most or least relevant parts (i.e., tokens) from our samples during inference (RQ2) or during training (RQ3) could lower incorrect classifications. In contrast to the research by Guo et al. and Warnecke et al. where tokens to be removed are determined per sample, our research has determined which tokens to remove by analyzing the entire dataset [13, 47]. Our first experiment hypothesized that removing the most relevant tokens of false positive (FP) samples before classification would lower the number of FP samples after classification. We constructed five sets with increasingly larger amounts of most relevant tokens found in FP samples and compared the classification results and metrics on another set of samples to test this hypothesis. Table 8.1 shows the results of this experiment in the column group "FP experiment". Although our experiment hypothesis was rejected due to the increase of FP classifications and decrease of average precision in all five sets, we observed some notable differences in the results. The decrease in false positives in set size 30 and the large increase in set size 50 show us there are tokens in these two sets that slightly direct the model towards non-vulnerable or strongly towards vulnerable classifications, respectively.

Table 8.1: This table shows the results of our experiments in removing the most and least relevant tokens before classification. The FP and FN columns contain the change in classification results of the model at a 0.5 probability threshold, and the AP columns contain the change in average precision of the model. The experiment results marked in bold indicate the lowest decrease in average precision.

	FP experiment		FN experiment		FP&FN experiment			
Set size	FP	AP	FN	AP	Set size	FP	FN	AP
10	**3.5%**	**-0.6%**	109.1%	-20.6%	19	261.8%	108.5%	-20.5%
20	17.0%	-1.6%	213.6%	-51.6%	39	519.8%	244.5%	-53.7%
30	7.6%	-2.4%	415.4%	-59.9%	56	252.3%	463.2%	-61.3%
40	29.0%	-14.2%	433.7%	-60.2%	72	128.7%	538.1%	-61.3%
50	650.2 %	-54.1%	569.6%	-64.9%	90	169.2%	626.6%	-70.8%

Conversely, in our second experiment, we hypothesized that removing the least relevant tokens of false negative (FN) samples before classification would lower the amount of FN samples after classification. In this experiment, we also refuted our test hypothesis. We showed that all five sets increased the number of false negative classifications and

decreased the average precision in our test dataset. However, whereas the previous experiment metrics started declining in set size forty, in this experiment, the metrics started declining more pronounced and immediately in the first set. This implies that the tokens in the sets significantly reduce the model's power to detect vulnerable samples.

Our third experiment yielded worse results than both our previous experiments. We tested the hypothesis that removing the most and least relevant tokens of incorrect classified samples (FN and FP respectively) before classification would lower their numbers after classification. Only the first combined set (size 19, due to overlapping token ".") has a smaller FN and AP change than the FN experiment.

Whereas our previous three experiments removed tokens before classification, our fourth and final experiment tried to decrease false positive classifications by changing the model training procedure. We removed the set of ten highly relevant tokens (the set having the lowest decrease in AP) from the training samples before fitting the model. We compared the classification results and metrics to their baselines.

Table 8.2: This table shows the results of our experiments in removing highly relevant tokens before fitting the model. The AP column contains the average precision of the model. The P, R, and F1 columns show the precision, recall, and F1 metric. The TP, FN, TN, and FP columns contain the classification output of the model at a 0.5 probability threshold.

	AP	P	R	F1	TP	FN	TN	FP
SySeVR	0.91	**0.83**	**0.85**	0.84	788	141	6413	158
SySeVR-FP	0.91	**0.87**	**0.81**	0.84	756	173	6459	112
LRP	0.95	**0.86**	**0.89**	0.87	9172	1145	50749	1517
LRP-FP	0.94	**0.89**	**0.86**	0.87	8721	1596	51191	1075

Whereas this experiment showed (see table 8.2) a decrease of false positive classifications in the test (-29.11%) and lrp-test datasets (-29.14%), it also showed an increase of false negative classifications (22.7% and 39.4% respectively). Correspondingly, the precision values have increased by 4.6% and 3.8%, and the recall values have decreased by 4.1% and 4.9%. Furthermore, we saw that the average precision had not changed significantly (+0.002% and -0.005%). We concluded from these results that filtering highly relevant tokens samples before training a model decreases the number of false positives and that our experiment hypothesis is accepted. However, the decrease in recall values also indicates that while our new model has become better in discerning whether samples are vulnerable, it can, on the other hand, perform this detection only on a smaller number of samples. This raises the question of how relevant our research and its results are towards the problem of vulnerability detection. When answering this question from a theoretic perspective, we see that the related research (by Guo et al. and Warnecke et al.) measures their explanation methods performance where we measure whether the model uses relevant parts of vulnerabilities. Also, whereas their research works with fixed models, we have analyzed the impact of removing the most or least relevant tokens before and after the fitting of a model [13, 47]. From a practical point of view, our method can increase the precision of a software vulnerability detection system and, therefore, can reduce the effort required to find vulnerabilities. However, when reviewing code to find implementation errors that can pose a risk to the intended use of the application, a vulnerability detection system with a lower recall value can potentially overlook vulnerabilities. Improving a vulnerability detection system should thus, show improvements on both metrics.

8.3. LIMITATIONS

Determining whether the software vulnerability detection model uses the relevant parts of vulnerabilities and improving this model was limited in several ways.

We depend on the metadata of the SARD dataset to determine the labels of our vulnerable lines. This metadata contained the vulnerable line numbers in the source files. Because the SySeVR samples do not include these line numbers, we have labeled the lines in our dataset by comparing the vulnerable lines from the metadata to the source code used to create the samples. This process could have mislabeled lines in our samples. We found several vulnerable samples without vulnerable lines and non-vulnerable samples with vulnerable lines during our validation of the labels. Also, the SySeVR dataset contained vulnerable samples with vulnerability types that do not have any source code (for example, memory leaks). As the vulnerable lines in these samples cannot be determined, they decrease the vulnerable line detection precision and the average localization precision.

Our model has been trained with SySeVR samples, which were created by transforming NVD and SARD samples [26, 27]. The proportions between the sample origins are roughly $\frac{1}{5}$ NVD and $\frac{4}{5}$ SARD. Such a difference in sample origin proportions impacts the model's ability to detect vulnerabilities from both origins. This could be deduced from the fact that the model performed better on the lrp-analysis (SARD only) dataset (0.95 average precision) compared to the test (SARD+NVD) dataset (0.91 average precision). Additionally, because the samples with SARD origins are created by transforming smaller and less complex programs, the model could be biased towards samples having these qualities. Furthermore, we only used samples with SARD origins in our analysis because their vulnerable line numbers could be obtained. Therefore, this bias could play an even more prominent role in the retrained model. Finally, because the SARD contains synthetic samples, we could argue that they probably contain similar source code sections in the training and validation datasets. Whether our model can generalize to unseen data is validated by comparing its training loss to its validation loss and stopping the training when the validation loss is increasing. If the samples in both datasets are very similar the validation loss could be lower than it would be without similarity and this could produce a model that has high performance on synthetic samples but is unable to perform well on truly unseen data.

The deep learning model of the study under investigation is not publicly available. We have reached out to the research group several times to request the model used in the SySeVR study. Since they did not respond, we had to recreate their model. Thanks to the thorough descriptions of the training procedure in the SySeVR paper we could create a comparable model having the same metrics. However, our model contains a single lstm layer, but the SySeVR model contains two layers. Although the performance metrics were comparable, this limits the results of our findings to our model architecture. Also, our trained models could have learned from different input data because the training procedure included randomization of input data and because this was not reproducible, its limits our metrics' comparisons to theirs.

The normalization step of the samples is done using a hand-built lexer. This lexer introduces errors when normalizing some parts of the source code. We found that a += token always yield two separate tokens (+ and =). Furthermore, because the lexer ignores escape characters in strings, the rest of the characters in such a string are skipped during the lexical analysis. Since the word-embeddings are created from the normalized tokens, these faults will negatively influence the generalizability of the embeddings. For example, the embed-

dings will learn to represent that the addition (+) operator often precedes the assignment (=) operator, which is not valid syntax in C or C++.

9

CONCLUSION & RECOMMENDATIONS

CONCLUSION

Our research was aimed at discovering whether we could improve the SySeVR system by explaining its incorrect classifications using an adaptation of the LRP method by Arras et al. [21, 2]. Towards this end, we stated three research questions. The answers to our first question guided our other two research questions whether removing the most or least relevant parts before and after model training can improve the classification performance.

To determine which parts of vulnerabilities are relevant in a deep learning setting (RQ1), we measured how precise LRP can detect vulnerable lines across our entire dataset and how precise LRP locates vulnerable lines in individual samples. With an average precision of seventeen percent, the LRP method is better at detecting vulnerable lines than a random guess with an average precision of eight percent. In spite of this improvement there are still many vulnerable lines which are not detected using our approach. This shows that our application of the LRP method is not well suited to detect vulnerabilities.

Our research shows that the LRP method can detect, on average, seventeen percent of the vulnerable lines in our dataset. Although this is an improvement when compared to our baseline performance of eight percent (a random guess), the majority of vulnerable lines are not detected using our approach. Therefore, we conclude that our application of the LRP method is not well suited to detect vulnerable lines in our dataset. Furthermore, our application of LRP locates on average sixteen percent of the vulnerable lines in individual samples. Because the majority of our vulnerable samples has a single vulnerable line, our application LRP will rarely locate it which limits its practical use in explaining these samples.

To ascertain whether explanations of incorrect classified samples can be used to improve the SySeVR system, we performed experiments that filtered the most and least relevant parts originating from incorrect classifications before (RQ3) and after (RQ2) fitting the model. After fitting the model, the number of correct classifications and the average precision decreased by removing the most relevant parts from our samples. Conversely, removing them before fitting the model did not change the average precision significantly but decreased the number of FP classifications by 29% at the expense of increasing the number of FN classifications by 23%.

Our analysis of explanations improves the understanding of deep learning approaches in the context of software vulnerability detection. We show that the lines we labeled as

being vulnerable do not play a significant role in our model's classifications. This insight into the behavior of our model was unexpected and lowered our trust in its ability to detect vulnerabilities when applied in a different setting.

Our adjustments to the model training procedure yielded a model in which, at the same classification threshold, a larger proportion of detected vulnerabilities are relevant. However, they also resulted in a model that selects a smaller proportion of the vulnerabilities. An improvement would require higher proportions on both terms and therefore, we did not improve our model's performance but rather shifted its focus in this classic trade-off in vulnerability detection systems.

Recommendations

Compare performance of models on realistic samples

The SySeVR dataset is mainly based on simple synthetic vulnerabilities. In addition, the sample extraction procedure does not prevent overlap between the samples, which could lead to many similarities between the training and validation datasets. Examining whether vulnerability detection systems such as VulDeepecker, SySeVR, and Russell's work deliver the same high performance on more complex samples obtained from natural occurring vulnerabilities could determine the generalization power of deep learning vulnerability detection systems and assess whether they have practical value in software engineering practices [22, 21, 34].

Measure and compare the precision of other explanation methods

Our precision analyses have determined how precise the layer-wise relevance propagation method detects vulnerable lines and to what extent these lines are used in the classification task. To this end, we have created a dataset that can be re-used to determine which explanations methods yield the best results. For example, the precision of the explanation methods employed by Warnecke et al. could be compared after training on our dataset [47].

Improve labels of vulnerable parts

Our analysis compared vulnerable lines with predicted vulnerable lines and determined the model's average precision. Our unit of precision, lines, contains tokens that either attribute towards the vulnerability or against it. We marked lines as vulnerable when the maximum token relevance was higher than a threshold. This coarse approach was necessary because the vulnerabilities were described at line-level. Future work could make the approach more precise by determining the precision at the token level. For this purpose, a dataset with vulnerabilities labeled at the token level would have to be created and analyzed. On the other hand, there could be other ways to mark lines as vulnerable (for example, training a classifier or other statistical inference methods) which could increase average precision and provide better insight into the model's usage of the vulnerable parts in classifying samples.

Prevent normalization errors

The SySeVR system normalize tokens to prevent over-fitting on unique tokens when fitting the model. As described in paragraph 8.3, this introduced errors which resulted in the exclusion of tokens (for example, the addition and subtraction assignments += and -=) and

string-parts (for example, the characters in strings following an escaped string-endings \'). Future work could prevent these errors by normalizing the tokens using the rich information present in the abstract syntax tree used to create the program slices.

Improve the semantic value of tokens

The normalization step renames methods and variables given to them by programmers. The meaning given to these names is hidden because they have been renamed to a generic name and a number (e.g., func_02 or variable_19) signifying the position in the sample. Future work could improve the semantic value of tokens by applying techniques commonly used in natural language processing to retain the meaning given to names. For example, stemming, reducing words to their root stem or word could be used to limit the uniqueness of names.

Include the position of tokens in the analysis

Our explanations contain relevance values obtained from the bi-directional recurrent model. Each relevance value is obtained by summing this model's forward and backward relevance value. Therefore, the token position relative to its predecessor or successor cannot be used in the analysis. Future work could improve the analysis by breaking down the analysis in forward and backward relevance values. This would make interpreting the explanations easier because the changes in relevance values from successive tokens can be observed.

Filter tokens individually

Although the FN-30, FN-40, and FN-50 filters contain tokens that improve the classification of non-vulnerable samples, these tokens also worsen the classification of vulnerable samples. Future work could investigate which tokens precisely were responsible for these changes. This could be done by decreasing the filter step size from ten to 1. In this way, the effect of removing each token can be determined.

Increase the maximum sample size

The SySeVR system works with a maximum sample size of 500 tokens. When a sample is found that exceeds this maximum size, it is truncated in such a way that the starting point of the program slice (the parts suspected to be involved in a vulnerability) is kept. We do not know the reason for limiting the sample size. However, future work could investigate whether this maximum is necessary and the relation between the sample size and the localization precision.

Break down the analysis in different types of vulnerabilities

Although vulnerabilities have different forms of manifestation in source code (for example, buffer overflows or memory leaks), our analysis does not differentiate between types of vulnerabilities. Our results could be broken down into specific vulnerability types. If some of these types are detected with low precision, the model's overall performance could be increased by the approach employed by Guo et al. to generate samples in categories that are under-represented in the dataset [13].

References

Literature

[1] Maximilian Alber et al. "iNNvestigate Neural Networks!" In: *Journal of Machine Learning Research* 20.93 (2019), pp. 1–8. ISSN: 1533-7928. URL: http://jmlr.org/papers/v20/18-540.html (visited on 09/28/2021).

[2] Leila Arras et al. "Explaining Recurrent Neural Network Predictions in Sentiment Analysis". In: *Proceedings of the 8th Workshop on Computational Approaches to Subjectivity, Sentiment and Social Media Analysis*. Copenhagen, Denmark: Association for Computational Linguistics, Sept. 2017, pp. 159–168. DOI: 10.18653/v1/W17-5221. URL: https://www.aclweb.org/anthology/W17-5221 (visited on 10/24/2019).

[3] Sebastian Bach et al. "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation". In: *PLOS ONE* 10.7 (July 2015). Ed. by Oscar Deniz Suarez, e0130140. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0130140. URL: https://dx.plos.org/10.1371/journal.pone.0130140 (visited on 10/31/2019).

[4] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". English (US). In: 3rd International Conference on Learning Representations, ICLR 2015 ; Conference date: 07-05-2015 Through 09-05-2015. Jan. 2015.

[5] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning Long-Term Dependencies with Gradient Descent is Difficult". In: *IEEE TRANSACTIONS ON NEURAL NETWORKS* 5.2 (1994).

[9] Ruth Fong and Andrea Vedaldi. "Interpretable Explanations of Black Boxes by Meaningful Perturbation". In: *2017 IEEE International Conference on Computer Vision (ICCV)* (Oct. 2017). ZSCC: 0000306 arXiv: 1704.03296, pp. 3449–3457. DOI: 10.1109/ICCV.2017.371. URL: http://arxiv.org/abs/1704.03296 (visited on 02/24/2020).

[11] Xiaodong Gu et al. "Deep API learning". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. New York, NY, USA: Association for Computing Machinery, Nov. 2016, pp. 631–642. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950334. URL: http://doi.org/10.1145/2950290.2950334 (visited on 09/29/2020).

[13] Wenbo Guo et al. "LEMNA: Explaining Deep Learning based Security Applications". en. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto Canada: ACM, Jan. 2018, pp. 364–379. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243792. URL: https://dl.acm.org/doi/10.1145/3243734.3243792 (visited on 11/17/2020).

[15] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997). Conference Name: Neural Computation, pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

[16] Andrej Karpathy and Justin Johnson and Fei-Fei Li. "Visualizing and Understanding Recurrent Networks". In: *CoRR* abs/1506.02078 (2015). arXiv: 1506.02078. URL: http://arxiv.org/abs/1506.02078.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[18] Sebastian Lapuschkin et al. "Unmasking Clever Hans predictors and assessing what machines really learn". In: *Nature Communications* 10.1 (Mar. 2019), pp. 1–8. ISSN: 2041-1723. DOI: 10.1038/s41467-019-08987-4. URL: https://www.nature.com/articles/s41467-019-08987-4 (visited on 11/12/2019).

[19] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998). ZSCC: 0024688, pp. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791.

[20] Alexander LeNail. "NN-SVG: Publication-Ready Neural Network Architecture Schematics". In: *Journal of Open Source Software* 4.33 (2019), p. 747. DOI: 10.21105/joss.00747. URL: https://doi.org/10.21105/joss.00747.

[21] Zhen Li et al. "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities". In: *IEEE Transactions on Dependable and Secure Computing* (2021). Conference Name: IEEE Transactions on Dependable and Secure Computing, pp. 1–1. ISSN: 1941-0018. DOI: 10.1109/TDSC.2021.3051525.

[22] Zhen Li et al. "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection". In: *Proceedings 2018 Network and Distributed System Security Symposium* (2018). DOI: 10.14722/ndss.2018.23158. URL: http://arxiv.org/abs/1801.01681 (visited on 08/29/2019).

[23] Gary McGraw. "Software Security: Building Security In". In: *2006 17th International Symposium on Software Reliability Engineering*. Nov. 2006, pp. 6–6. DOI: 10.1109/ISSRE.2006.43.

[24] Tomás Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2013. URL: http://arxiv.org/abs/1301.3781.

[25] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. "Methods for interpreting and understanding deep neural networks". In: *Digital Signal Processing* 73 (Feb. 2018), pp. 1–15. ISSN: 1051-2004. DOI: 10.1016/j.dsp.2017.10.011. URL: http://www.sciencedirect.com/science/article/pii/S1051200417302385 (visited on 12/18/2019).

[30] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/14bfa6bb14875e45bba028a21ed38046-Paper.pdf.

[31] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “"Why Should I Trust You?": Explaining the Predictions of Any Classifier”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. New York, NY, USA: Association for Computing Machinery, Aug. 2016, pp. 1135–1144. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939778. URL: https://doi.org/10.1145/2939672.2939778 (visited on 03/31/2022).

[32] Dennis W. Ruck, Steven K. Rogers, and Matthew Kabrisky. “Feature selection using a multilayer perceptron”. In: *Journal of Neural Network Computing* 2.2 (1990), pp. 40–48.

[33] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[34] Rebecca Russell et al. “Automated Vulnerability Detection in Source Code Using Deep Representation Learning”. In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. Dec. 2018, pp. 757–762. DOI: 10.1109/ICMLA.2018.00120.

[35] Wojciech Samek and Klaus-Robert Müller. “Towards Explainable Artificial Intelligence”. In: *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*. Ed. by Wojciech Samek et al. Lecture Notes in Computer Science. ZSCC: NoCitationData[s2]. Cham: Springer International Publishing, 2019, pp. 5–22. ISBN: 978-3-030-28954-6. DOI: 10.1007/978-3-030-28954-6_1. URL: https://doi.org/10.1007/978-3-030-28954-6_1 (visited on 10/31/2019).

[36] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. “Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models”. In: *arXiv:1708.08296 [cs, stat]* (Aug. 2017). URL: http://arxiv.org/abs/1708.08296 (visited on 12/18/2019).

[37] Wojciech Samek et al. “Evaluating the visualization of what a deep neural network has learned”. In: *IEEE transactions on neural networks and learning systems* 28.11 (2016), pp. 2660–2673.

[39] Hossain Shahriar and Mohammad Zulkernine. “Classification of Static Analysis-Based Buffer Overflow Detectors”. In: *Secure Software Integration and Reliability Improvement Companion, IEEE International Conference on* 0 (June 2010), pp. 94–101. ISSN: 978-0-7695-4087-0. DOI: 10.1109/SSIRI-C.2010.28.

[40] Abhinav Shrivastava, Abhinav Gupta, and Ross Girshick. “Training Region-Based Object Detectors With Online Hard Example Mining”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[42] Robert Tibshirani. “Regression Shrinkage and Selection via the Lasso”. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 58.1 (1996). Publisher: [Royal Statistical Society, Wiley], pp. 267–288. ISSN: 0035-9246. URL: https://www.jstor.org/stable/2346178 (visited on 02/21/2022).

[43] Robert Tibshirani et al. “Sparsity and Smoothness via the Fused Lasso”. In: *Journal of the Royal Statistical Society. Series B (Statistical Methodology)* 67.1 (2005). Publisher: [Royal Statistical Society, Wiley], pp. 91–108. ISSN: 1369-7412. URL: http://www.jstor.org/stable/3647602 (visited on 02/21/2022).

[44] John Viega et al. “Token-based scanning of source code for security problems”. In: *ACM Transactions on Information and System Security (TISSEC)* 5.3 (2002), pp. 238–261.

[45] Oriol Vinyals et al. “Show and tell: A neural image caption generator”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3156–3164.

[46] David A Wagner et al. “A first step towards automated detection of buffer overrun vulnerabilities.” In: *Network and Distributed System Security Symposium*. Vol. 20. 0. 2000.

[47] Alexander Warnecke et al. “Evaluating Explanation Methods for Deep Learning in Security”. In: *2020 IEEE European Symposium on Security and Privacy (EuroS P)*. Sept. 2020, pp. 158–174. DOI: 10.1109/EuroSP48549.2020.00018.

[48] Martin White et al. “Deep learning code fragments for code clone detection”. en. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. Singapore, Singapore: ACM Press, 2016, pp. 87–98. ISBN: 978-1-4503-3845-5. DOI: 10.1145/2970276.2970326. URL: http://dl.acm.org/citation.cfm?doid=2970276.2970326 (visited on 06/06/2019).

[49] Martin White et al. “Toward Deep Learning Software Repositories”. In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. MSR ’15. event-place: Florence, Italy. Piscataway, NJ, USA: IEEE Press, 2015, pp. 334–345. ISBN: 978-0-7695-5594-2. URL: http://dl.acm.org/citation.cfm?id=2820518.2820559.

WEB LINKS

[6] *Checkmarx*. 2018. URL: www.checkmarx.com.

[8] European Commission. *Communication Artificial Intelligence for Europe*. Apr. 2018. URL: https://ec.europa.eu/digital-single-market/en/news/communication-artificial-intelligence-europe (visited on 02/24/2020).

[26] NIST. *NVD - Home*. URL: https://nvd.nist.gov/ (visited on 01/14/2020).

[27] NIST. *Software Assurance Reference Dataset*. URL: https://samate.nist.gov/SRD/index.php (visited on 01/14/2020).

[28] Christopher Olah. *Understanding LSTM Networks*. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[29] S Özkan. *CVE details*. 2020. URL: https://www.cvedetails.com/ (visited on 03/10/2020).

BOOKS

[7] Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. ISBN: 0-262-33737-1.

[14] Samuel Z Guyer, Emery D Berger, and Calvin Lin. *Detecting errors with configurable wholeprogram dataflow analysis*. Computer Science Department, University of Texas at Austin, 2002.

[38] Wojciech Samek et al., eds. *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*. Vol. 11700. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019. DOI: 10.1007/978-3-030-28954-6.

[41] IEEE Computer Society, Pierre Bourque, and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3rd. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014. ISBN: 978-0-7695-5166-1.

Theses

[12] Philip Jia Guo. "A Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs". 2006, p. 112.

A

IoU DATA

	TP+FN		TP		FN	
IoU bins	#	%	#	%	#	%
[0.00 - 0.06)	30734	76,2%	27688	76,6%	3046	72,6%
[0.06 - 0.11)	15	0,0%	15	0,0%	0	0,0%
[0.11 - 0.17)	110	0,3%	99	0,3%	11	0,3%
[0.17 - 0.22)	286	0,7%	242	0,7%	44	1,0%
[0.22 - 0.28)	348	0,9%	278	0,8%	70	1,7%
[0.28 - 0.33)	16	0,0%	14	0,0%	2	0,0%
[0.33 - 0.39)	1136	2,8%	1007	2,8%	129	3,1%
[0.39 - 0.44)	22	0,1%	20	0,1%	2	0,0%
[0.44 - 0.50)	2	0,0%	2	0,0%	0	0,0%
[0.50 - 0.56)	3249	8,1%	2898	8,0%	351	8,4%
[0.56 - 0.61)	1	0,0%	1	0,0%	0	0,0%
[0.61 - 0.67)	0	0,0%	0	0,0%	0	0,0%
[0.67 - 0.72)	5	0,0%	1	0,0%	4	0,1%
[0.72 - 0.78)	0	0,0%	0	0,0%	0	0,0%
[0.78 - 0.83)	1	0,0%	0	0,0%	1	0,0%
[0.83 - 0.89)	0	0,0%	0	0,0%	0	0,0%
[0.89 - 0.94)	0	0,0%	0	0,0%	0	0,0%
[0.94 - 1.00]	4420	11,0%	3885	10,7%	535	12,8%
Total	40345	100%	36150	100%	4195	100%

Figure A.1: localization precision histogram bins

IoU (number)	IoU (fraction)	TP+FN	TP	TP (%)	FN	FN (%)
0,00	0	30734	27688	76,6%	3046	72,6%
1,00	1	4420	3885	10,7%	535	12,8%
0,50	1/2	3249	2898	8,0%	351	8,4%
0,33	1/3	1129	1002	2,8%	127	3,0%
0,25	1/4	344	276	0,8%	68	1,6%
0,20	1/5	163	135	0,4%	28	0,7%
0,17	1/6	122	106	0,3%	16	0,4%
0,14	1/7	63	52	0,1%	11	0,3%
0,13	1/8	36	36	0,1%	0	0,0%
0,40	2/5	17	16	0,0%	1	0,0%
0,29	2/7	15	14	0,0%	1	0,0%
0,11	1/9	10	10	0,0%	0	0,0%
0,38	3/8	7	5	0,0%	2	0,0%
0,09	1/11	6	6	0,0%	0	0,0%
0,08	1/12	5	5	0,0%	0	0,0%
0,43	3/7	5	4	0,0%	1	0,0%
0,67	2/3	5	1	0,0%	4	0,1%
0,22	2/9	4	2	0,0%	2	0,0%
0,10	1/10	3	3	0,0%	0	0,0%
0,44	4/9	2	2	0,0%	0	0,0%
0,08	1/13	1	1	0,0%	0	0,0%
0,13	2/15	1	1	0,0%	0	0,0%
0,18	2/11	1	1	0,0%	0	0,0%
0,30	3/10	1	0	0,0%	0	0,0%
0,56	5/9	1	1	0,0%	0	0,0%
0,80	4/5	1	0	0,0%	0	0,0%
	Total	40345	36150	100,0%	4193	100%

Figure A.2: Vulnerability localization precision in IoU. The first columns shows the numeric IoU, the second column shows the fractional IoU, and the third column shows the count.

B

TOKEN FILTERS

Position	Token	Relevance	FP10	FP20	FP30	FP40	FP50
1	variable_45	64.298241	V	V	V	V	V
2	variable_58	60.128542	V	V	V	V	V
3	variable_57	55.746562	V	V	V	V	V
4	.	51.453395	V	V	V	V	V
5	func_12	50.608787	V	V	V	V	V
6	variable_59	47.123687	V	V	V	V	V
7	variable_55	43.929762	V	V	V	V	V
8	variable_46	42.215565	V	V	V	V	V
9	variable_56	42.100341	V	V	V	V	V
10	variable_37	41.683834	V	V	V	V	V
11	variable_50	39.504745	X	V	V	V	V
12	variable_7	38.482806	X	V	V	V	V
13	fscanf	38.329897	X	V	V	V	V
14	variable_38	32.846796	X	V	V	V	V
15	variable_44	32.817806	X	V	V	V	V
16	[	30.368421	X	V	V	V	V
17	variable_48	29.3365	X	V	V	V	V
18	variable_51	29.248835	X	V	V	V	V
19	->	28.811576	X	V	V	V	V
20	<	27.736428	X	V	V	V	V
21	variable_43	27.481129	X	X	V	V	V
22	for	25.238955	X	X	V	V	V
23	variable_34	22.39254	X	X	V	V	V
24	rand	21.431771	X	X	V	V	V
25	]	21.321187	X	X	V	V	V
26	variable_49	20.846717	X	X	V	V	V
27	2f	19.329627	X	X	V	V	V
28	malloc	18.629495	X	X	V	V	V
29	||	18.385245	X	X	V	V	V
30	variable_53	16.742198	X	X	V	V	V
31	*	16.633307	X	X	X	V	V
32	'	15.312781	X	X	X	V	V
33	variable_10	14.99465	X	X	X	V	V
34	0	14.840038	X	X	X	V	V
35	;	14.802512	X	X	X	V	V
36	!	14.301812	X	X	X	V	V
37	variable_47	13.556854	X	X	X	V	V
38	50	13.081739	X	X	X	V	V
39	int	12.960234	X	X	X	V	V
40	=	12.647106	X	X	X	V	V
41	PSNR	12.333354	X	X	X	X	V
42	memset	12.279753	X	X	X	X	V
43	A	12.079588	X	X	X	X	V
44	fwrite	11.960222	X	X	X	X	V
45	variable_60	11.917929	X	X	X	X	V
46	wcsncat	11.687396	X	X	X	X	V
47	static	11.678635	X	X	X	X	V
48	++	11.519237	X	X	X	X	V
49	snprintf	11.473445	X	X	X	X	V
50	variable_52	11.311738	X	X	X	X	V

Figure B.1: This list shows the top 50 relevance values in the lrp-analysis dataset. Tokens can have duplicate relevance values and therefore we show unique tokens.

Position	token	relevance	FN-10	FN-20	FN-30	FN-40	FN-50
1	struct	-101.262873	V	V	V	V	V
2	somewhy_mutter	-95.825466	V	V	V	V	V
3	variable_13	-66.880423	V	V	V	V	V
4	char	-64.300028	V	V	V	V	V
5	*	-38.182653	V	V	V	V	V
6	variable_15	-37.543332	V	V	V	V	V
7	strlen	-34.818731	V	V	V	V	V
8	variable_17	-32.183596	V	V	V	V	V
9	)	-29.918172	V	V	V	V	V
10	.	-28.806291	V	V	V	V	V
11	(	-24.405715	X	V	V	V	V
12	wcscpy	-22.640079	X	V	V	V	V
13	"	-19.522448	X	V	V	V	V
14	realloc	-19.344156	X	V	V	V	V
15	for	-18.700141	X	V	V	V	V
16	func_0	-18.407131	X	V	V	V	V
17	variable_16	-18.029269	X	V	V	V	V
18	void	-17.790379	X	V	V	V	V
19	]	-17.720663	X	V	V	V	V
20	fopen	-17.660313	X	V	V	V	V
21	=	-17.460237	X	X	V	V	V
22	func_19	-16.413662	X	X	V	V	V
23	100	-16.24149	X	X	V	V	V
24	A	-15.753497	X	X	V	V	V
25	strncat	-15.387826	X	X	V	V	V
26	fscanf	-14.859488	X	X	V	V	V
27	,	-14.316588	X	X	V	V	V
28	sscanf	-12.525322	X	X	V	V	V
29	;	-12.346908	X	X	V	V	V
30	variable_19	-12.267249	X	X	V	V	V
31	fgetws	-11.842288	X	X	X	V	V
32	String	-11.655856	X	X	X	V	V
33	malloc	-11.438162	X	X	X	V	V
34	fgets	-11.054693	X	X	X	V	V
35	wchar_t	-11.021056	X	X	X	V	V
36	variable_14	-10.697026	X	X	X	V	V
37	sizeof	-10.584139	X	X	X	V	V
38	-	-10.45989	X	X	X	V	V
39	sprintf	-10.403002	X	X	X	V	V
40	C	-10.307526	X	X	X	V	V
41	variable_20	-9.935989	X	X	X	X	V
42	/	-8.845482	X	X	X	X	V
43	Initialize	-8.778893	X	X	X	X	V
44	to	-8.593478	X	X	X	X	V
45	variable_11	-8.483465	X	X	X	X	V
46	printf	-8.362429	X	X	X	X	V
47	[	-8.237116	X	X	X	X	V
48	func_1	-8.159127	X	X	X	X	V
49	variable_23	-8.066963	X	X	X	X	V
50	func_21	-8.020141	X	X	X	X	V

Figure B.2: This list shows the bottom 50 relevance values in the lrp-analysis dataset. Tokens can have duplicate relevance values and therefore we show unique tokens.

C

FILTER RESULTS

	label	auc	ap	tn	fn	tp	fp	fpr	fnr	a	p	r	mcc	f1	n
0	SySeVR	0.906	0.906	6413	141	788	158	0.024	0.152	0.960	0.833	0.848	0.818	0.841	7500
1	SySeVR FP-10	NaN	NaN	10	0	0	0	0.000	0.000	1.000	0.000	0.000	0.000	0.000	10
2	SySeVR FN-10	0.846	0.846	6188	141	788	383	0.058	0.152	0.930	0.673	0.848	0.717	0.750	7500
3	SySeVR FP-FN-10	0.848	0.848	6310	193	735	262	0.040	0.208	0.939	0.737	0.792	0.730	0.764	7500
4	LRP	0.946	0.946	50749	1145	9172	1517	0.029	0.111	0.957	0.858	0.889	0.848	0.873	62583
5	LRP FP-10	0.941	0.941	51191	1596	8721	1075	0.021	0.155	0.957	0.890	0.845	0.842	0.867	62583
6	FP-10	0.940	0.940	50696	1245	9072	1570	0.030	0.121	0.955	0.852	0.879	0.839	0.866	62583
7	FP-20	0.930	0.930	50491	1346	8971	1775	0.034	0.130	0.950	0.835	0.870	0.822	0.852	62583
8	FP-30	0.922	0.922	50633	1639	8678	1633	0.031	0.159	0.948	0.842	0.841	0.810	0.841	62583
9	FP-40	0.804	0.804	50309	3469	6848	1957	0.037	0.336	0.913	0.778	0.664	0.668	0.716	62583
10	FP-50	0.405	0.405	40885	3270	7047	11381	0.218	0.317	0.766	0.382	0.683	0.379	0.490	62583
11	FN-10	0.751	0.751	46795	2394	7923	5471	0.105	0.232	0.874	0.592	0.768	0.600	0.668	62583
12	FN-20	0.458	0.458	42552	3591	6726	9714	0.186	0.348	0.787	0.409	0.652	0.393	0.503	62583
13	FN-30	0.379	0.379	45841	5902	4415	6425	0.123	0.572	0.803	0.407	0.428	0.299	0.417	62583
14	FN-40	0.376	0.376	46270	6111	4206	5996	0.115	0.592	0.807	0.412	0.408	0.294	0.410	62583
15	FN-50	0.332	0.332	48334	7667	2650	3932	0.075	0.743	0.815	0.403	0.257	0.220	0.314	62583
16	FP-FN-10	0.752	0.752	46778	2387	7930	5488	0.105	0.231	0.874	0.591	0.769	0.600	0.668	62583
17	FP-FN-20	0.438	0.438	42863	3944	6373	9403	0.180	0.382	0.787	0.404	0.618	0.374	0.488	62583
18	FP-FN-30	0.366	0.366	46921	6449	3868	5345	0.102	0.625	0.812	0.420	0.375	0.286	0.396	62583
19	FP-FN-40	0.366	0.366	48796	7306	3011	3470	0.066	0.708	0.828	0.465	0.292	0.275	0.358	62583
20	FP-FN-50	0.276	0.276	48183	8320	1997	4083	0.078	0.806	0.802	0.328	0.194	0.145	0.244	62583

Figure C.1: This table shows the classification results and metrics of our token-filter experiments