MASTER'S THESIS 2021

Vulnerability detection using ensemble learning

Rasmus Lindqvist, Viktor Bard

ISSN 1650-2884
LU-CS-EX: 2021-07

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-07

Vulnerability detection using ensemble learning

Rasmus Lindqvist, Viktor Bard

Vulnerability detection using ensemble learning

Rasmus Lindqvist
lindqvist.rasmus@hotmail.com

Viktor Bard
viktor.bard@hotmail.com

March 2, 2021

Master's thesis work carried out at Debricked AB.

Supervisors: Emil Wåreus, emil.wareus@debricked.com
Pierre Nugues, pierre.nugues@cs.lth.se

Examiner: Jacek Malec, jacek.malec@cs.lth.se

Abstract

Managing vulnerabilities in open-source software is becoming increasingly more important when the use of open-source is only increasing as a standard in software development. The vast amount of open-source versions makes manual inspection, both impractical and costly. We propose an automated approach to vulnerability detection by using ensemble techniques. We benefit from models predicting on git commit messages and code changes, and use these in an ensemble together with meta data. Furthermore we explore the possibility to expand the ensemble by implementing a graph neural network to classify abstract syntax trees in code changes. Our logistic regression model, shows an increase of F1 score by 4.95% from the best individual model in the ensemble. Our Neural Network model shows an increase of 4.7% in precision from previous models.

Acknowledgements

Looking back at the work carried out in this project there are a number of people we want to recognize, as the job would have been next to impossible without them. We want to start with a special thanks to both of our supervisors, Emil Wåreus at Debricked and Pierre Nugues at LTH, for all the invaluable feedback and guidance throughout the entire thesis work. Both have taken a great amount of time to assist us in this project. We would also like to thank all of Debricked's employees for their recognition and welcoming to the team with a special thanks to Magnus Tullberg and Anton Duppils, assisting us in challenges along the way.

Contents

1 Introduction **7**
- 1.1 Background . . . 7
- 1.2 Task . . . 7
- 1.3 Outline . . . 8

2 Data **9**
- 2.1 Dataset . . . 9
  - 2.1.1 Features . . . 10
  - 2.1.2 Dataset Sampling . . . 10
  - 2.1.3 Common Weakness Enumeration . . . 11
- 2.2 Exploratory Data Analysis . . . 12
  - 2.2.1 Data Structure EDA . . . 12
  - 2.2.2 Prediction Data . . . 13
  - 2.2.3 Meta Data . . . 14
  - 2.2.4 EDA Conclusion . . . 15

3 Related Work **17**
- 3.1 Security Identification . . . 17
  - 3.1.1 Issues . . . 17
  - 3.1.2 Commit . . . 17
  - 3.1.3 Code Change . . . 18
  - 3.1.4 Metadata . . . 18
- 3.2 Ensemble Learning . . . 18

4 Approach **19**
- 4.1 Proposed Ensemble Architecture . . . 19
  - 4.1.1 Models . . . 20
  - 4.1.2 Neural Network Parameters . . . 21
- 4.2 Proposed Code Change Addition . . . 24
  - 4.2.1 Graph Neural Networks . . . 24

5 Experiments 27
5.1 Validation . . . 28
5.2 Hyper-tuning . . . 28

6 Results 29
6.1 Performance Metrics . . . 29
6.2 CWE performance . . . 32
6.3 Ngrams . . . 32
6.4 Ablation Study . . . 33
6.5 Manual Inspection . . . 33

7 Discussion 37
7.1 Results Discussion . . . 37
7.2 Setbacks and mistakes . . . 38
7.2.1 Dataset . . . 38
7.2.2 Relational Links . . . 38
7.3 Validity and Limitations . . . 39
7.3.1 Validity . . . 39
7.3.2 Repository Distribution . . . 39
7.3.3 Language restriction . . . 39

8 Conclusion 41
8.1 Future Work . . . 41
8.1.1 Data . . . 41
8.1.2 AST Improvement . . . 41
8.1.3 Additional Languages . . . 42

References 43

Chapter 1
Introduction

1.1 Background

Over the years, open-source software has become increasingly attractive in software development as it offers a cheap and fast way of production. According to a report by Synopsis (2020), 99% of analyzed code bases contained open-source components.

However, managing security risks with an increased amount of software components, that are developed by a third party, has become increasingly difficult. Unintentional flaws in open-source libraries can create security vulnerabilities, which may be exploited by malicious parties. As the open-source libraries often have fast update cycles and complicated dependencies, it is hard to keep track of possible vulnerabilities with manual inspection. An automated approach to vulnerability detection has the potential to increase software security while reducing costs.

1.2 Task

Essentially all widely used open-source libraries use git as their version handling system. This thesis investigates the use of information in git patches to identify potential vulnerabilities in open-source code.

We benefit from three previously developed models at Debricked, predicting on git issues, commits and code changes, separately. By utilizing information from different aspects of git data, this thesis aims to explore the possibilities to use ensemble learning as a method to capture multiple vulnerability aspects.

Ensemble learning is the term used when combining several machine learning models by training another model on top of these. As the underlying models only have access to parts of the whole data spectrum, the goal with an ensemble model is to capture a larger portion of the data and increase the amount information used in prediction.

We analyze how such ensemble techniques can be used to identify more vulnerabilities with a higher precision than individual models. Furthermore, we explore the possibility to use code syntax trees as additional information to the ensemble, by implementing a *graph neural net* model.

1.3 Outline

The thesis is divided into the following sections: Data, Related Work, Approach, Experiments, Results, Discussion, and Conclusion:

- The *Data chapter* explains the data at hand and the processing of data to fit the specific task.
- *Related Work* presents the work from previous research in the subject of security classification.
- The *Approach chapter* presents our solution to the task and the models used in this project.
- In the *Experiments chapter*, we explain the different experiments that we ran.
- We then present the results from these experiments in the *Results chapter*.
- We analyze and discuss the results in the *Discussion*.

Chapter 2
Data

The data in this thesis is supplied by *Debricked*, which has acquired it from GitHub and the *National Vulnerability Database* (NVD). The description of the data is followed by a presentation of our exploratory data analysis.

2.1 Dataset

Vast amounts of publicly posted issues, commits, and code changes are available from version control systems such as GitHub and Gitlab. In this thesis, we used an internal data set of 10M+ issues, 20M+ commits and 20M+ code changes covering popular open-source repositories on Github. We present a brief explanation of what the issues, commit messages and code changes data contains:

- The issues contain text describing different software improvement suggestions, such as feature requests, enhancements, bug reports as well as security reports.
- The commit contains a message with text description of the fix made.
- The code change contains the actual code diff, with information about the lines in the code that changed in the fix.

As the Github data alone does not include labels of security vulnerabilities, we use the National Vulnerability Database (NVD) to label Github data as security related. NVD is managed by the U.S. government and provides a database of disclosed vulnerabilities, called *Common Vulnerabilities and Exposures* (CVE). Each CVE entry contains information about the type of security and severity, as well as information about affected software versions. The CVEs sometimes contain a link to a specific issue, commit or code change.

Hence, by retrieving these links, we can label the Github issues, commits and code changes as security related. Throughout the rest of the report we refer to these labeled data points as security entities.

2.1.1 Features

The features for the ensemble can be split into two major components: prediction data from previous models and meta data. We plan to use these features in an ensemble architecture to combine the predictions from previous models and meta data.

The prediction data is referred to as the confidence interval of the issue, commit, and code change classifiers, and is a score of how confidently the models have predicted an entity as a security fix. Throughout the rest of the report, these models will be referred to as the *Level 0* models.

In addition to the predictions, we will investigate the impact of the commit and code change metadata. Examples of commit metadata includes: the number of files changed and the number of lines of code (LOC) added and deleted in the entire commit. Examples of code change metadata include lines of code added, changed and deleted, as well as the number of functions changed.

2.1.2 Dataset Sampling

When selecting a dataset, the largest limitation lies in the number of security entities for a specific language. As the code change model is language dependent, we decided to choose the language with the most known security entities. Thus we considered the repositories using mainly C with a total of 1709 security entities distributed over 56 repositories.

A second limitation is the issue of *data leakage*. Data leakage is the term used when a model in testing has access to data that is not available in production, and hence shows an over-performance. To mitigate this risk when training an ensemble, it is important to separate the training set of the Level 0 models, from the training set of the ensemble. Hence, the training set of the Level 0 models was separated from the data set for the ensemble.

As NVD consists only of known security vulnerabilities, there are no data points labeled as non-security fixes. With the assumption that only a small fraction of the total amount of issues, commits, and code changes are security fixes, it is improbable to randomly sample a security related data point. With this assumption, we randomly sampled a proportional amount of negative (non-security related) data points from the 56 repositories.

We selected a final data set of **1709** security data points and **1709** non-security data points distributed over **56** repositories. We separated it into a training set for the Level 0 models and a train/test set for the ensemble.

Performing k-fold on the ensemble data set allows us to train and evaluate on the whole data set. In extension, this means that we can also compare the performance to the level 0 models on a larger data set.

	#Samples	#Positive	#Negative
Level 0 train set	1386	693	693
Ensemble train/test set	2032	1016	1016
Total	3418	1709	1709

Table 2.1: Number of samples in data sets

2.1.3 Common Weakness Enumeration

Every security vulnerability can be placed in a category depending of the type of flaw that it contains. As a standard, the *Common Weakness Enumeration* (CWE) category system is used to place each security vulnerability in over 600 different categories. CWE is a community-developed list of common weaknesses and acts as a tool to label and describe weaknesses in standardized terms. Figure 2.1 shows the distribution of the most common CWEs in the data set. We give a brief explanation of the four most common vulnerabilities in the dataset below. For the interested reader on all of the software CWEs, we refer to: `https://cwe.mitre.org/data/`.

CWE-119 Improper Restriction of Operations within the Bounds of a Memory Buffer

> The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.

CWE-200 Exposure of Sensitive Information to an Unauthorized Actor

> The product exposes sensitive information to an actor that is not explicitly authorized to have access to that information.

CWE-125 Out-of-bounds Read

> The software reads data past the end, or before the beginning, of the intended buffer.

CWE-20 Improper Input Validation

> The product receives input or data, but it does not validate or incorrectly validates that the input has the properties that are required to process the data safely and correctly.

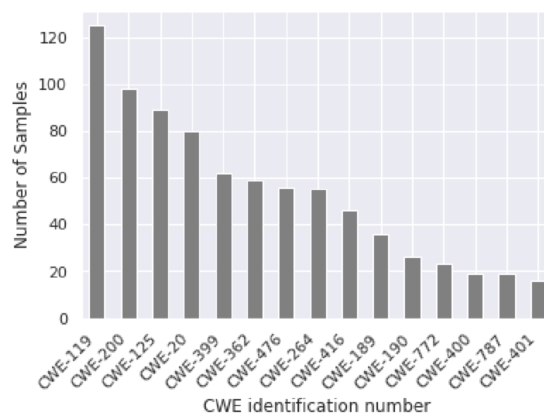


Figure 2.1: Top 15 CWEs

2.2 Exploratory Data Analysis

In this section, we describe the results of an exploratory data analysis (EDA) on the data structure, the Level 0 models, and the metadata. The purpose of the EDA is to get a better understanding of the data, and how to best approach the task. Furthermore, we investigate where we can find signal in the data, which will be an important base for decisions in the model design.

2.2.1 Data Structure EDA

In git version systems, the issues and commits are linked through pull-requests. The ideal scenario would be that all commits fix a specific issue so that information from all three classifiers can be used in every case, like in Figure 2.2.

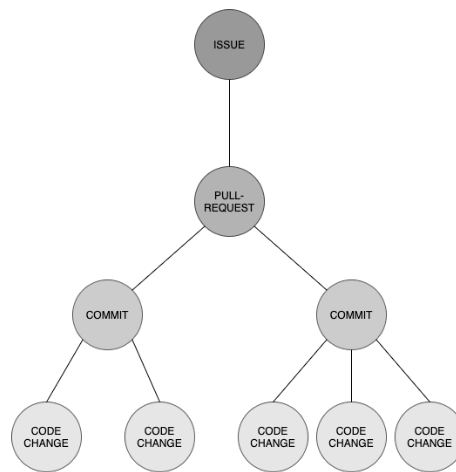


Figure 2.2: Ideal structure

However, we found that the completeness of these links is very low in the data set. Only 0.03% contain a link between an issue and a commit. A part of this problem lies in that the scraping of the Github pull-request information is incomplete. The second part is that not all commits are actually linked to an issue. Figure 2.3 shows the actual structure found from the EDA.

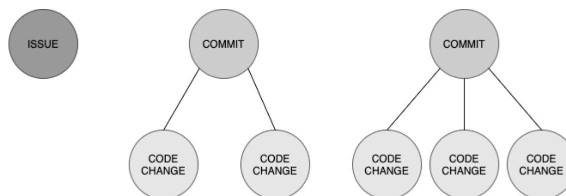
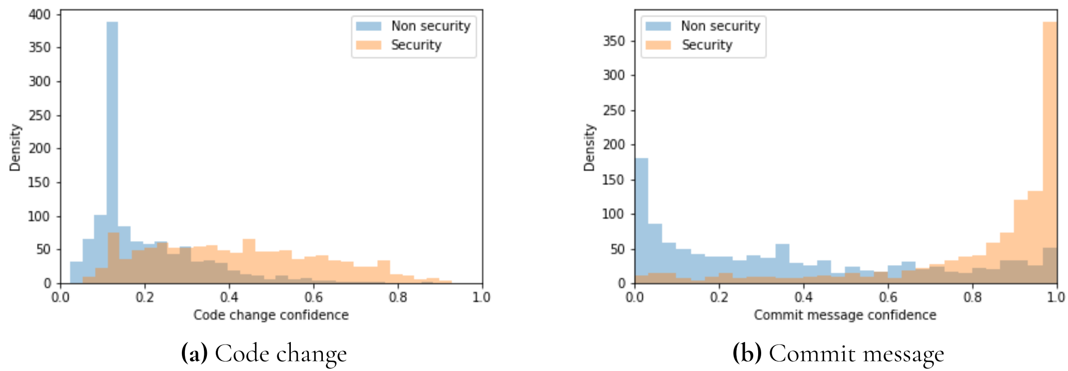


Figure 2.3: Actual structure

A decision was made to focus the project on commits and code changes as these are always linked to each other. More about this decision in the Approach chapter. Though, due to this decision, the EDA does not include issues.

2.2.2 Prediction Data

We investigated the performance of the Level 0 models on the ensemble test set. The results are presented in the plots below. Figures 2.4a and 2.4b display the confidence score distributions for the code change classifier and the commit classifier. The commit message classifier distributions are more separated and is expected to perform better than the code change classifier.



(a) Code change **(b)** Commit message

Figure 2.4: Classifiers confidence

As shown in Figure 2.5, the correlation between their predictions is low. This fact makes an ensemble technique promising as it can hopefully capture a larger part of the union from the level 0 models. That is to say, increase the recall without loosing precision.

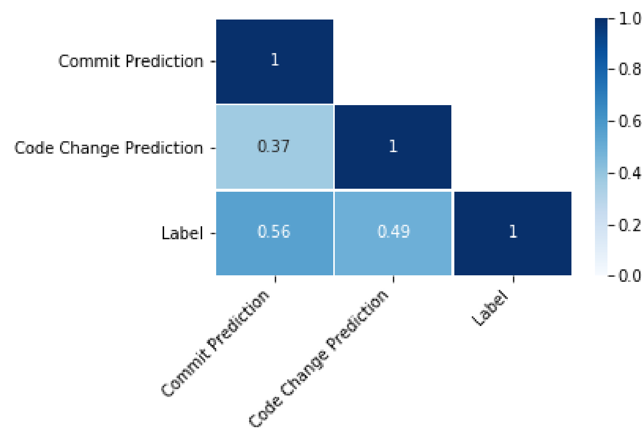


Figure 2.5: Correlation

To analyze if the models perform better on certain types of vulnerabilities, we compared the classifiers performance per CWE.

- We found that the commit classifier performs equally across all CWEs, which indicates that people describe the security fix that was made in the commit message, independently of type of CWE.
- The code change classifier varies drastically in performance between CWEs, from as low as 42% to as much as 92% accuracy. This indicates that different types of CWEs vary in complexity and some are harder to find by only looking at a code diff. The code change classifier is expected to perform better on simple and localized changes, as a code diff only includes a small code context.

Looking at the CWEs, where the code change classifier performs best and worst, respectively, it is evident that the code change classifier does perform better on localized changes. For example, CWE-787 concerns Out-of-bounds Write and are possibly quite localized changes. CWE-416 (Use-after-free) and CWE-264 (Permissions, Privileges, and Access Controls), on the other hand, are possibly more spread out in the code and hence not captured in the code diff.

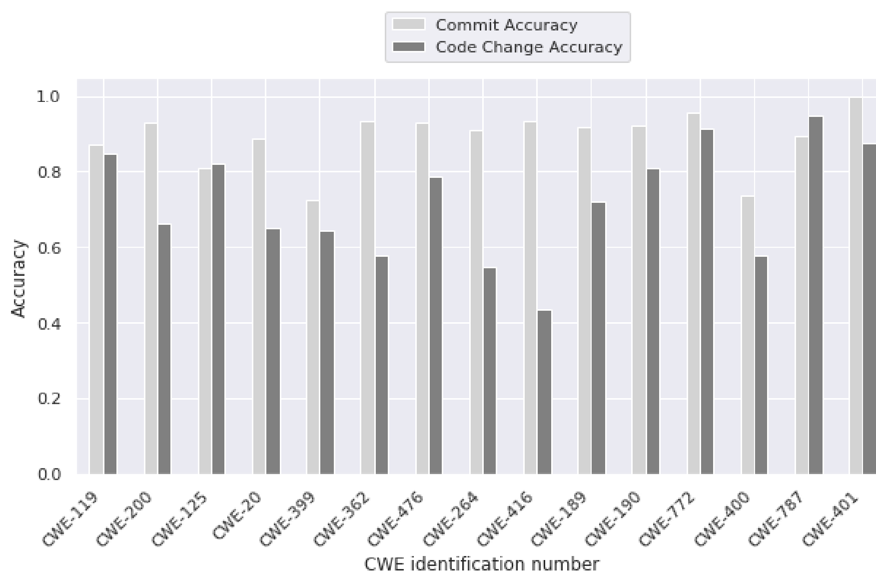


Figure 2.6: Classifiers Accuracy per CWE

2.2.3 Meta Data

As suggested by Li and Paxson (2017), there is a significant difference between security and non-security. We performed an extended analysis with a similar approach and found possible cues in several meta data points.

In Table 2.2, we present the differences between security and non-security related patches regarding metadata. Security related patches are often smaller and more localized. Furthermore security patches result in a net reduction of code base less frequently than ordinary patches.

	Security Related	Non-Security Related
Code change characteristics		
Net reduction of code base	10.2 %	21.2 %
More then 100 LOC changed	6.8 %	15.4 %
Average number LOC changed	45.0	109.4
Change of only one function	79.3 %	59.6 %
Commit characteristics		
Average number of files changed	1.83	2.77
Change of only 1 file	71.1 %	58.4 %

Table 2.2: Results of EDA on the meta data

2.2.4 EDA Conclusion

The data structure proved to be less ideal than expected due to a lack of relations between data points. However, the correlation between the commit classifier and the code change classifier is low. Furthermore, we found differences in meta data between security and non-securities. This bodes well for that an ensemble approach with the code change and commit classifier, together with the metadata will increase the performance from the level 0 classifiers.

Chapter 3

Related Work

3.1 Security Identification

3.1.1 Issues

Vulnerability detection based on posted issues in open-source software repositories can be used to automatically monitor known vulnerabilities. As a potential piece of the ensemble classification, we have an issue-based vulnerability classifier provided by Debricked through the master's thesis of Duppils and Tullberg (2020). They present a novel approach of using natural language processing (NLP) for automating surveillance on open-source issues. The approach is using the semi-supervised learning technique virtual adversarial training (VAT) that was originally developed for image tasks, that later has found relevance in text classification problems. The purpose of VAT is to trick the model to make mistakes during training to get a more robust model in the end. they use VAT in a hierarchical attention network (HAN) which is an architecture developed for text document classification.

3.1.2 Commit

To decide if a patch is meant to fix a vulnerability or not, a potential method is to analyse the messages written by developers to each commit. Debricked has previously developed a model that utilized this method. The provided model from Debricked utilizes transfer learning by implementing Google's BERT language model (Devlin et al., 2018). The version of BERT used is a pre-trained model on text from the software domain, and is then fine-tuned on the specific task of classifying commits messages as security relevant or not.

3.1.3 Code Change

Vulnerability detection through code change analysis is a widely researched area. Different techniques have been proposed to determine if code changes in patches are security related or not. A code change vulnerability classifier has already been developed by Debricked, and will be the one of the classifiers used in the ensemble in this thesis. This classifier uses a bag-of-words technique that has the raw changes from each commit as input.

Another take on Code Change classification is to look at the entire function where the change has been made. Zhou et al. (2019) have investigated this through in their work on Devign (Deep vulnerability identification via graph neural networks), utilizing graph representations of functions and implementing a graph neural network (GNN) to learn the internal semantics of functions and identify security related functions.

3.1.4 Metadata

Empirical studies on meta data related to security entities have been performed. In a large scale study on security patches, Li and Paxson (2017) showed significant difference in meta data on code change in security-related patches compared to non-security patches.

3.2 Ensemble Learning

Hoang et al. (2019) presented an ensemble algorithm in their model called PatchNet. The classifier takes in two embedding vectors from a commit message module and a code change module. They show that the collection of two features shows a significant increase in performance. Although, PatchNet does only use the actual message and codechange of the available data in a commit. We will investigate an ensemble with the models previously developed at Debricked together with meta data from the individual features, meaning we will investigate the addition of meta data and complementary models.

Chapter 4
Approach

This chapter presents the workflow for the project and the decisions made during development. The workflow can roughly be split into three sections: EDA, ensemble modeling, and addition of a new model to the ensemble.

As this project aims to combine multiple individually developed models, the project is heavily dependant on available data and relations between the data points. As mentioned in the EDA section, we discovered a lack of connections between issues and commits in the data set. Due to this, we decided to limit the input models to commits, code changes and meta data. Because of this decision, we also decide to use a flat data structure instead of using a graph data structure for the ensemble. In clarification, this means that in the cases where a commit has multiple code changes we merge these to a single code change. Per commit, the resulting output from the level 0 models is hence one prediction value from the commit message classifier and one prediction value from the code change classifier.

When comparing the level 0 models during the EDA, we also noticed that the code change model was under-performing compared to the commit-classifier which led us to investigate additional methods for code change classification to find out if the addition of a yet another model could increase the overall ensemble performance. The additional model will utilize the syntactic relation in the code using *abstract syntax trees* (AST), which is a graph representation of the code structure. More on this in section 4.2.

4.1 Proposed Ensemble Architecture

Our proposed architecture utilizes predictions from models in addition with the meta data directly. Since the meta data is also used by the level 0 classifiers, the purpose of adding it directly to the ensemble is that the ensemble can better learn which predictions are more valuable under certain meta-states. The data flow is presented in Figure 4.1

We tested multiple models for the ensemble: two simpler models of logistic regression and random forest as well as two more complex ones based on a neural network. We evaluated all

ensemble models by a 5-fold cross validation and compared to the level 0 models and versus the meta- and AST-code-change classifiers individually.

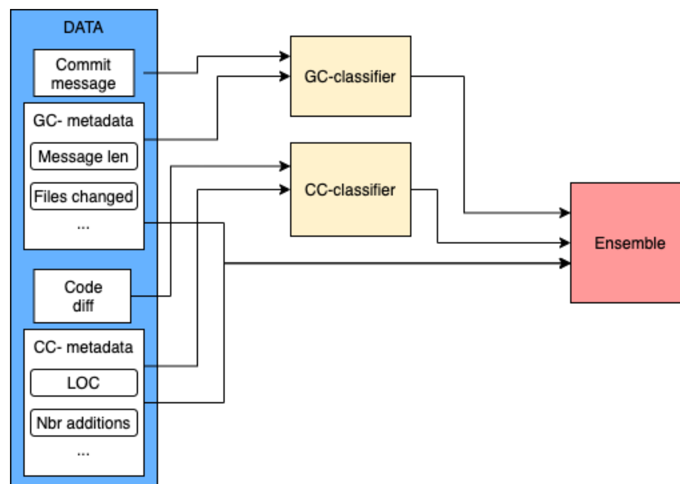


Figure 4.1: Ensemble architecture including git commit-classifier (GC-classifier) and code change classifier (CC-classifier) together with meta data

4.1.1 Models

For the ensemble, we chose to evaluate four different models. The first two are commonly used basic classifiers: logistic regression and random forest. The second two are two setups for a neural network: one ordinary neural network with the assumption that all data points that are not registered in the NVD are not a security related commit. The other set up is a form of semi-supervised learning called Positive-unlabeled-learning (PU-learning), where only the confirmed positive labels are regarded as relevant to avoid the assumption regarding negative labels.

Logistic Regression

Logistic regression is one of the more basic classification algorithms. It predicts the probability of a certain outcome $p(Y)$ based on a set of variables, X. The logistic model is trained on a set of data points and fits a curve that best split the data set into the different categories.

Random Forest

Random forest in itself is an ensemble learning algorithm. It's an ensemble of decision trees which all predicts the output of a certain data point and the output of the random forest model is the mean output of these decision trees.

Neural Networks

The theory of neural networks has been around for a long time, but with the advances in GPU processing capacities in the 2000s, neural networks came to practical use to solve real

world problems (Grossi and Buscema, 2008). Since then, a vast amount of new neural network algorithms have been developed to solve previously unsolvable problems and increase performance. Now, neural networks come in many different shapes and sizes depending on complexity and the task at hand.

Neural networks can be compared to how the neurons in the brain work, hence the name. A neural network is constructed of several layers of neurons, an input layer, an output layer and one or more hidden layers in between. The layers are connected to each other with weighted connections. The model learns from data points and the weighted connections are updated so that the model learns how to best classify the data.

Positive-Unlabeled Learning (PU-Learning)

As mentioned in the data chapter, our positive data consists of disclosed vulnerabilities in the NVD. Additionally, the negative dataset then comes from a sample from these repositories that is not connected to NVD.

With PU-learning however, we don't need to assume that there is a negative dataset at all but instead call the sampled part of the dataset unlabeled. The probability that a certain label y of entity x is positive thus is the probability that a sample s is labeled divided by the probability that a positive sample is labeled in our data set according to Equation 4.1

$$P(y = 1 \,|x) = \frac{P(s = 1 \,|x)}{P(s = 1 \,|y = 1)} \tag{4.1}$$

4.1.2 Neural Network Parameters

Loss Function

A loss function is used to calculate a measure of how wrong a model's prediction was from the truth. Generally the model gets punished more if the prediction was far from the result. By calculating the gradient of the loss function, it is possible to update the weights in the model so that it better predicts the data. The type of loss function used can affect the model performance. Commonly used loss functions are cross entropy and mean square error.

Optimizer

In the context of neural networks, an optimizer is an algorithm or method used to change the networks attributes in order to reduce losses. Two examples of these attributes are weights and learning rate. Gradient descent is the base of most optimizers in neural networks. It is a first order algorithm which is dependent on the derivative of a loss function to calculate in which way to alter the weights in order to reach a minimum. The main advantage of this algorithm is its simplicity but in a more complex data set it might be trapped at a local minimum or in a large data set taking too long to calculate the gradient and taking too long to converge. This section will dig deeper into gradient descent optimizers used in this project.

Adagrad modifies the learning rate at each timestep based on the sum of squares of the past computed gradients, but that comes with a problem. The accumulation of the squared gradients in the denominator of the update rule seen in Equation 4.2 caused the learning rate

to shrink and eventually become infinitesimally small, causing the algorithm to essentially stop learning (Ruder, 2016).

Equations for Adagrad optimization

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t \tag{4.2}$$

where θ_{t+1} are the updated parameters, θ_t are the current parameters, η is the general learning rate, G_t is a diagonal matrix with the sum of squares of the gradients, ϵ is a smoothing term to avoid division by zero and g_t is the gradient.

RMSprop (Root Mean Square Propagation) is the successor of Adagrad. It optimizes the gradient descent using the addition of adaptive learning. RMSprop adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. The solution to Adagrad's problem in RMSprop were to, as the name suggests, divide the learning rate η with the square root of the sum of squared gradients seen in Eq. 4.3, i.e. divide the learning rate by an exponentially decaying average of squared gradients (Ruder, 2016).

Equation for RMSprop optimization:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t \tag{4.3}$$

where

$$E[g^2]_t = 0.9 \cdot E[g^2]_{t-1} + 0.1 \cdot g_t^2 \tag{4.4}$$

where θ_{t+1} are the updated parameters, θ_t are the current parameters, η is the general learning rate, $E[g^2]_t$ exponentially decaying mean square of past gradients, ϵ is a smoothing term to avoid division by zero and g_t is the gradient.

Activation Function

In a neural network, each node takes an input from a previous level and uses some function to calculate an output to the next level. This function is called an activation function. Some activation functions introduce a non-linear behavior into the network in order to find complex relations in the data. Since the activation function determines the output of each layer, the choice of activation function can greatly impact the behavior of a neural network. There are many possible activation functions. We describe the ones we used in this project.

The Rectified Linear Unit function (ReLU) is zero for all negative inputs and linear for all positive inputs (and zero) shown in Equation 4.5 and Figure 4.2. This is one of the less computationally heavy functions since it involves mathematical functions simpler than some of the following methods. One of its limitations however is that due to its output characteristics of $[0, \infty]$, it should only be used within hidden layers and not as output layer.

$$f_{ReLU}(x) = \max(0, x) \tag{4.5}$$

The sigmoid function takes any real value as input and maps to another value in range $[0, 1]$ as seen in Equation 4.6 and Figure 4.3. It basically has all characteristics you want in an activation function: it is non-linear, continuously differentiable and has a limited output

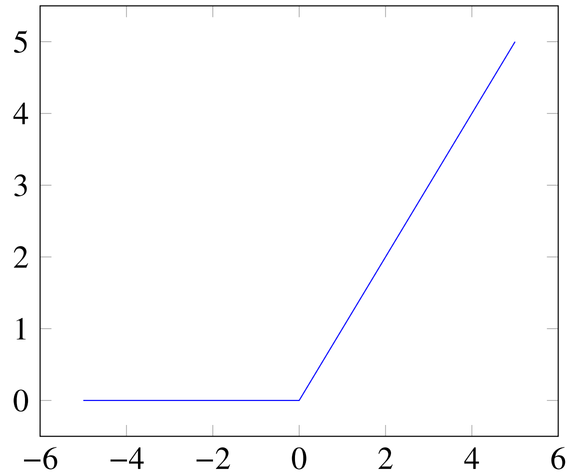


Figure 4.2: ReLU function.

range. The drawbacks of the sigmoid function is that towards either end of the function the y values respond fractionally of changes in $\mathbf{x}$ introducing the "vanishing gradient problem", where the neural network effectively stops training further. The limited output range however is perfect for an output layer in a classification model.

$$f_{sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{4.6}$$

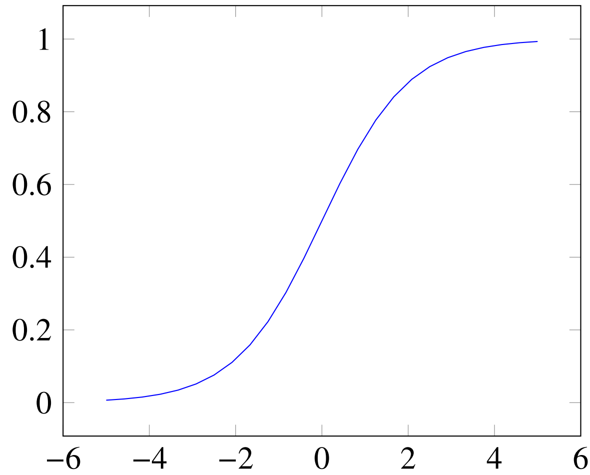


Figure 4.3: Sigmoid function.

4.2 Proposed Code Change Addition

The level 0 code change model handles the code as pure text. We propose an architecture to take advantage of syntactic information in the code as well. By extracting the entire function in which there has been a code change, we can label functions as vulnerable or not. We can then parse the functions into abstract syntax trees (AST) and train a model on these. The graph structure of an AST provides the relational aspect of a function, in contrast to the pure text used in the level 0 code change model.

To test the performance of this approach, we have adopted the Devign model (Zhou et al., 2019). The model uses a fuzzy C language parser enabling us to parse the functions, where code changes occur into abstract syntax trees. Devign then uses a GNN to predict on each data point. Similar to the previous level 0 models, we extract a confidence score that we feed to the ensemble model. The architecture after this addition can be seen in Figure 4.4

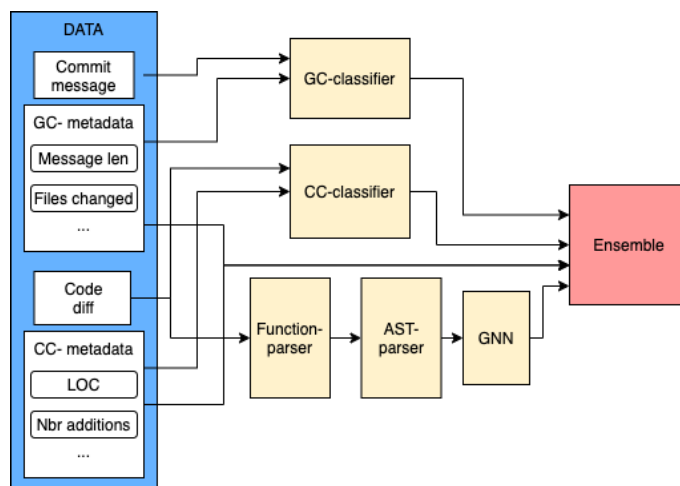


Figure 4.4: Ensemble architecture with AST classifier added

4.2.1 Graph Neural Networks

Graph neural networks (GNN) are a relatively new research area. They have proven a strong potential since they are able to describe and predict graph structured data.

Graph structured data is defined as a set of connected nodes and edges. Previous neural network models have proven strong performance on tasks such as image recognition, where the input format is fixed. To fit graph structured data into ordinary neural networks would mean the need to reduce the complexity of the graph and thus potentially loose important information which lies within this complexity (Zhong et al., 2020),

GNN retrieves information from nearby nodes by message passing between the nodes, through the edges. The nodes and edges can have different attributes which describe them. The message passing is performed for a certain number of time steps and this enables the model to extract relational information within the graph, which can be used to predict nodes or edges that have no information, or even the whole graph, called graph classification. The goal of the graph neural net is to learn a state embedding for every node.

If $h_v{}^t$ describe the state of node v at a certain time step t, we can describe the whole graph as a sum of all embedding vectors, H^t. H^t can then be used to classify the property of the whole graph. Many variants of graph neural networks exist, but what is shared among all variants is that each node acts as a recurrent unit, and all the edges act as a neural network. The model is trained so that the neural networks learn what information is most important to pass between nodes for specific classification problems. Exact update function f for $h_v{}^{t+1} = f(h)$ varies between different GNNs. For the interested reader on a GNNs, we refer to Zhou et al. (2018).

To extract the semantic relations within a given code block, the code can be regarded as a graph structure. Knowing this, we decided to investigate if a GNN could be used to classify the functions we extract from code changes. An example of a function parsed into a abstract syntax tree is shown in Figure 4.5.

```
int add(int a, int b)
{
    return a + b;
}
```

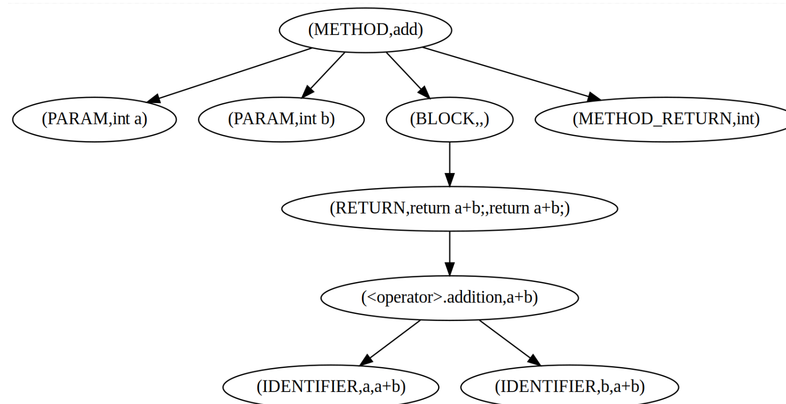


Figure 4.5: Example of an abstract syntax tree from a simple add method with two parameters a and b returned as a sum

Chapter 5

Experiments

In this chapter, we explain the experiments we carried out. Working from the hypothesis that an ensemble would perform better than any individual level 0 model, we first performed an initial investigation of two baseline ensemble-models: logistic regression and random forest. We continued with experiments on more complex models: neural net and neural net with PU-learning. Thereafter, we tested our new AST classifier. Finally, we evaluated the usefulness by predicting on recent commits and we performed a manual analysis. In summary, the following experiments are executed:

- 5-fold cross validation with Level 0 models;
- Hypertuning on neural net;
- 5-fold cross validation for the 4 ensemble models;
- 5-fold cross validation for the AST-classifier;
- 5-fold cross validation with the AST in the ensemble;
- Analysis of accuracy per CWE;
- Ablation study;
- Predict on recent commit and perform manual inspection;
- Bigram analysis of commit message for high and low confidence predictions.

5.1 Validation

In order to validate the performance of the models, we evaluated the performance on precision, recall, and F1. These are defined according to equations 5.1 through 5.3

$$precision = \frac{true\ positives}{true\ positives + false\ positives} \tag{5.1}$$

$$recall = \frac{true\ positives}{true\ positives + false\ negatives} \tag{5.2}$$

$$F_1\ score = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{5.3}$$

We evaluate these macro metrics on 5-fold splits of the dataset, where we train on 4/5 and test on 1/5. This procedure is repeated 5 times, meaning the whole data set is used for both training and testing. By doing this, we can also present the standard-deviation between the folds and get a robustness metric of the models.

To compare the ensemble to the level 0 models, we split the prediction results of these models 5 times. We calculate macro metrics for these splits to get a comparable metric. In addition, we test the correlation between the level 0 models to get an indication if the ensemble performance is reasonable.

5.2 Hyper-tuning

To establish the layout of the neural network, we perform a hyper-tuning to evaluate the network with varying composition of layers, number of nodes in each layer and optimizer. The hyper-tuning is evaluated to maximize the F1 score.

The resulting layout is a two-layer neural network with 64 and 32 nodes respectively. The resulting optimizer is RMSProp. All possible selections are presented in Table 5.1

Component	**Choices**
Number of layers	1, 2, 3, 4
Number of nodes per layer	4, 8, 16, 32, 64, 128, 256
optimizer	adam, RMSProp

Table 5.1: Choice of neural network parameters

Chapter 6
Results

6.1 Performance Metrics

The performance metrics presented come from three different groups of classifiers:

- The level 0 classifiers – input classifiers to ensemble
- Ensemble classifiers – utilizing commit, code change and the metadata classifier
- Ensemble classifiers with AST classifier – utilizing all level 0 classifiers.

In Table 6.1, we can see that when combining the models we do in fact get an increased F1 score in all tested models. The gain in F1 score comes from both an increased precision and recall. However the addition of the AST classifier was not able to show any further increase in performance.

Figure 6.2 shows the ensemble model with highest F1 score compared to all level 0 classifiers. All results are from a 5-fold cross-validation on the final ensemble data set. The logistic regression model showed the highest F1 score and will be the model used for comparison towards the level 0 models and presentation of results in Figures 6.1 to 6.3.

By running the meta data features in an individual model, we can compare the correlation of the predictions between the classifiers and the meta data. In order for models in an ensemble to contribute to an improved result, it is desired that their predictions do not overlap completely, i.e that they have low correlation. In Figure 6.4, we can see that the classifiers have a relatively low correlation, which is expected with the increased performance in the ensemble.

	Precision	Recall	F1
Level 0 classifiers			
Commit Classifier	76.6 ± 2.1	79.9 ± 3.7	78.25 ± 2.6
Code Change Classifier	78.9 ± 2.9	62.1 ± 2.2	69.5 ± 2.2
Meta Data Classifier	64.7 ± 2.8	74.65 ± 5.3	69.2 ± 3.0
AST Code Change Classifier	57.8 ± 4.5	58.4 ± 1.5	58.0 ± 2.2
Ensemble classifiers			
Logistic regression	81.3 ± 2.1	85.2 ± 1.5	**83.2 ± 1.4**
Random forests	80.5 ± 1.9	85.0 ± 1.1	82.7 ± 1.1
Neural network	**83.6 ± 2.8**	80.6± 6.8	81.9 ± 2.7
Neural network w PU-learning	80.6 ± 1.3	83.9 ± 2.4	81.1 ± 2.3
Ensemble classifiers w AST Classifier			
Logistic regression	80.5 ± 1.3	85.3 ± 4.5	82.7 ± 2.0
Random forests	80.0 ± 2.0	**86.3 ± 4.0**	83.0 ± 1.7
Neural network	80.6 ± 2.7	80.5 ± 6.7	80.4 ± 2.5
Neural network w PU-learning	81.6 ± 3.0	80.2 ± 7.5	80.6 ± 2.8

Table 6.1: Results of level 0 and ensemble classifiers. Bold entries signifies the best result for each metric.

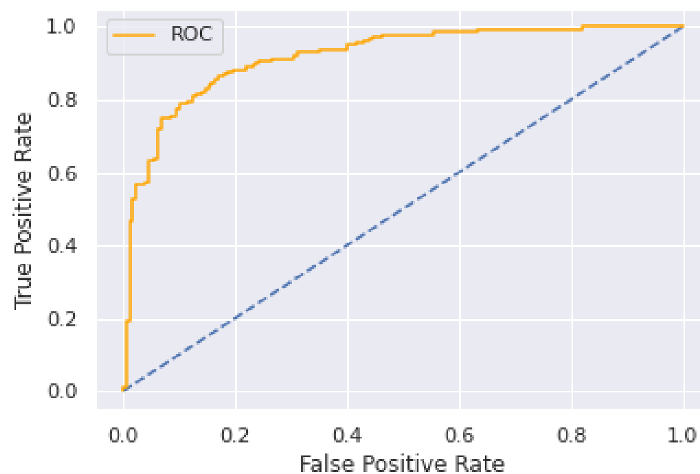


Figure 6.1: ROC of logistic regression ensemble model

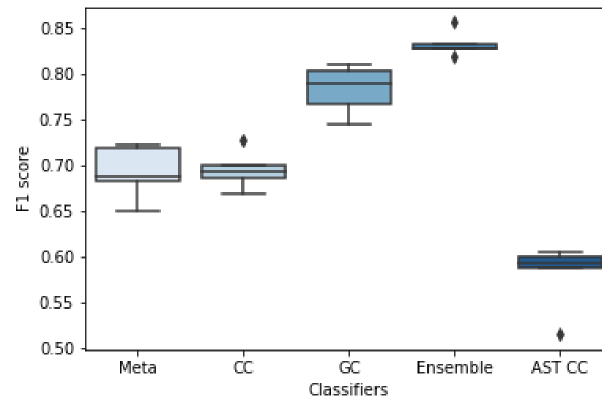


Figure 6.2: Boxplot F1-score

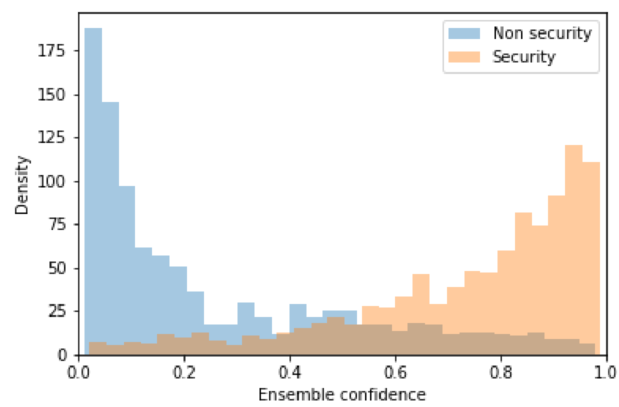


Figure 6.3: Logistic Regression Ensemble, confidence distribution

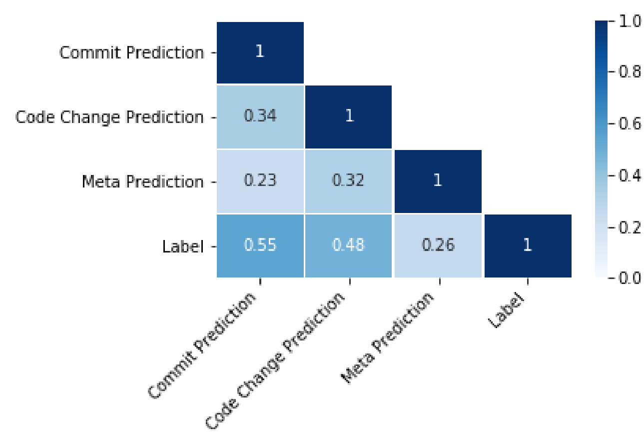


Figure 6.4: Correlation between predictors

6.2 CWE performance

We compare the *per-cwe-performance* between the ensemble and Level 0 models in Figure 6.5. We observe an accuracy increase in 8 out of 15 CWEs and equal performance in 3 out of 15 CWEs. The commit classifier out-performs the ensemble in 4 out of 15 CWEs.

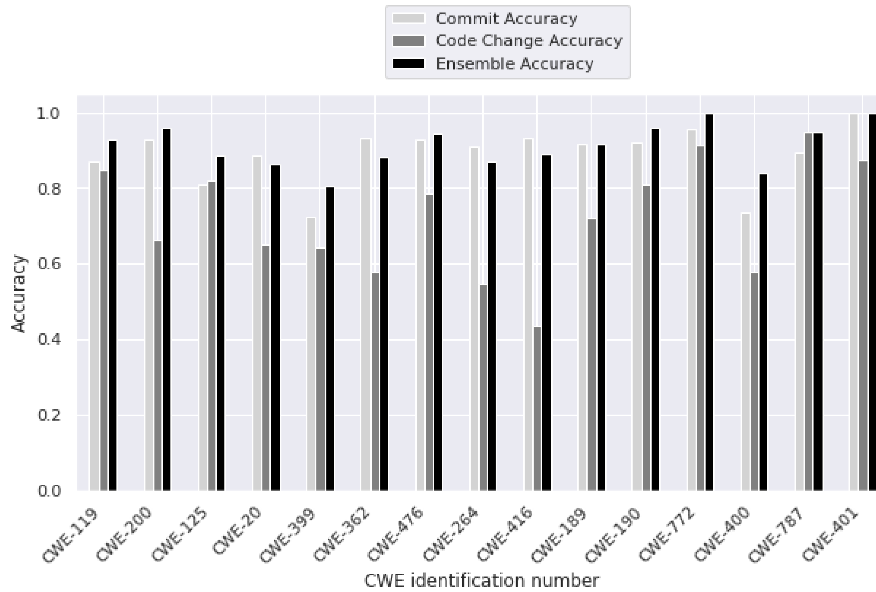
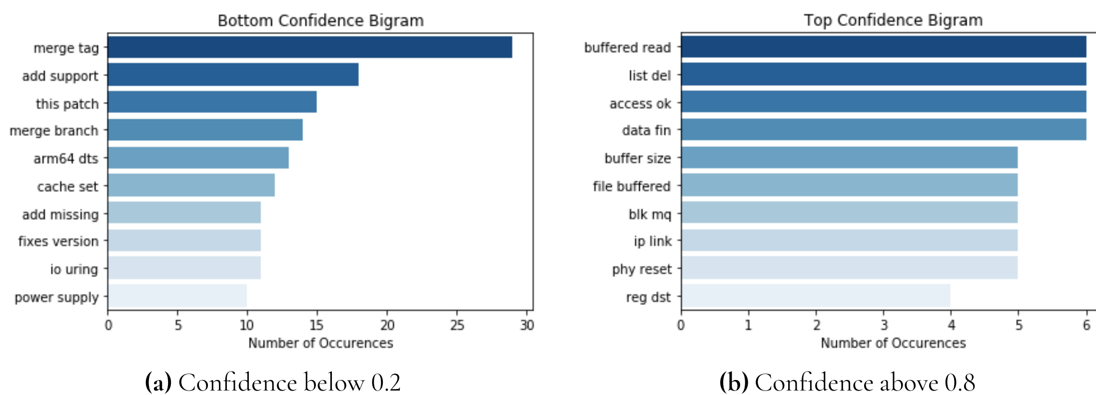


Figure 6.5: Ensemble accuracy per CWE

6.3 Ngrams

We performed a bigram analysis on the most common words found in the commit messages for two different confidence levels of our ensemble classifier. In Figure 6.6, we present the bigram results of a confidence level below 0.2 and above 0.8 of the ensemble.



(a) Confidence below 0.2

(b) Confidence above 0.8

Figure 6.6: Bigram for different confidence levels

6.4 Ablation Study

We rated the performance of the features by utilizing the Predictive Power Score (PPS) library, 8080labs (2020). It is a tool developed to find linear and non-linear relationships between features and labels. As expected, GC-confidence and CC-confidence are the features with the highest score. Delta change comes next and is basically just difference between additions - deletions. In the EDA, we could see that security related commits were less likely to have a negative delta change as well as a very large positive delta change, meaning that security related commits more rarely change in size by a lot. Furthermore, “other” proved to have a relatively strong performance. *Other* is referred to changes in the code base which is not in a `.c` file. For example changes in a config file. In the EDA, we could see that it was more likely that these types of changes occurred in non-security related changes.

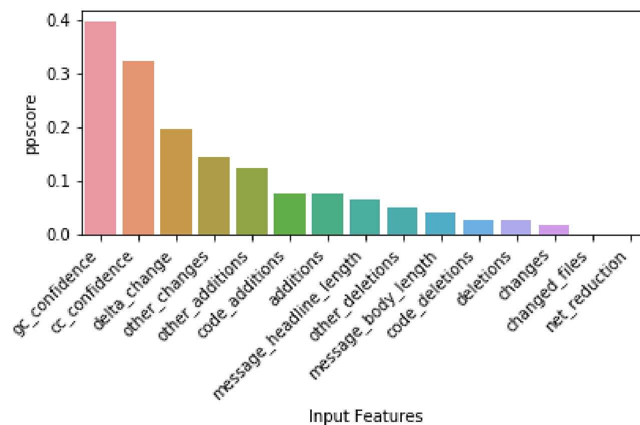


Figure 6.7: PPScore

6.5 Manual Inspection

We manually inspected the results on the test set to get a deeper understanding on where the model classifies incorrectly. By performing a k-fold, we can get an ensemble confidence over the whole ensemble data set. To get an understanding of where the classifier is completely off, we do a deeper analysis on false positives with a confidence above 0.8 and on false negatives below 0.2. Out of the test set of 1016 security commits and 1016 normal commits, we find 55 false positives above 0.8 confidence and 43 false negatives below 0.2.

Our findings are:

1. Incorrect predictions are due to the Level 0 models inaccurate confidence;
2. An error in the commit classifier where it classifies commits with no messages with a very high confidence, accounting for 18.5 % of false positives. Hence, potentially providing an easy fix for higher performance;
3. Many of the false positives contain phrases such as “overflow”, “memory”, “block”, “unbound loop” and “race condition”;

4. Four examples of false negatives where the commit message explicitly states “CVE”;

5. 16.3 % of false negatives contains a message with fewer than 6 words;

6. A few examples of the false positives are potentially actual vulnerabilities, meaning that they are mislabeled.

We can see a significant performance increment in the macro-metrics. Though, to test the actual usefulness of the ensemble classifier, we predicted on 900 previously unseen commits from October and November 2020 and analyzed the results. We are assisted by a software security expert at Debricked to investigate if the top ranked commits from the ensemble model can potentially be exploitable.

Below, we present two examples of commits with high model confidence, and that the Debricked security expert deemed interesting from a security aspect.

The first commit shown in Figure 6.8 and 6.9, is from the Linux repository, where the developer clearly states in the message that the commit fixes a use-after-free issue which can be related to CWE-416 (Use After Free) and in the added lines of code, there is a check added to make sure the resource is active.

In the second presented message from FFMPEG, shown in Figures 6.10 and 6.11 we can see that the developer states that the commit probably fixes an infinite loop bug, which can be related to CWE-835 (Loop with Unreachable Exit Condition, ‘Infinite Loop’) and a check with an exit condition is added in the code.

Read more about the CWEs at: `https://cwe.mitre.org/data/`

mlxsw: core: Fix use-after-free in mlxsw_emad_trans_finish()

```
Each EMAD transaction stores the skb used to issue the EMAD request
('trans->tx_skb') so that the request could be retried in case of a
timeout. The skb can be freed when a corresponding response is received
or as part of the retry logic (e.g., failed retransmit, exceeded maximum
number of retries).

The two tasks (i.e., response processing and retransmits) are
synchronized by the atomic 'trans->active' field which ensures that
responses to inactive transactions are ignored.

In case of a failed retransmit the transaction is finished and all of
its resources are freed. However, the current code does not mark it as
inactive. Syzkaller was able to hit a race condition in which a
concurrent response is processed while the transaction's resources are
being freed, resulting in a use-after-free [1].

Fix the issue by making sure to mark the transaction as inactive after a
failed retransmit and free its resources only if a concurrent task did
not already do that.
```

Figure 6.8: Linux commit message

```
3 drivers/net/ethernet/mellanox/mlxsw/core.c

@@ -620,6 +620,9 @@ static void mlxsw_emad_transmit_retry(struct mlxsw_core *mlxsw_core,
620 620                 err = mlxsw_emad_transmit(trans->core, trans);
621 621                 if (err == 0)
622 622                         return;
    623 +
    624 +               if (!atomic_dec_and_test(&trans->active))
    625 +                       return;
623 626         } else {
624 627                 err = -EIO;
625 628         }
```

Figure 6.9: Linux code change

avformat/moflex: Check m->size before seeking

Fixes: Infinite loop
Fixes: 26016/clusterfuzz-testcase-minimized-ffmpeg_IO_DEMUXER_fuzzer-6195663833137152

Figure 6.10: FFMPEG commit message

```
5 libavformat/moflex.c

@@ -342,8 +342,11 @@ static int moflex_read_packet(AVFormatContext *s, AVPacket *pkt)
342 342
343 343             m->in_block = 0;
344 344
345     -           if (m->flags % 2 == 0)
    345 +           if (m->flags % 2 == 0) {
    346 +               if (m->size <= 0)
    347 +                   return AVERROR_INVALIDDATA;
346 348                 avio_seek(pb, m->pos + m->size, SEEK_SET);
    349 +           }
347 350         }
348 351
349 352         return AVERROR_EOF;
```

Figure 6.11: FFMPEG code change

Chapter 7

Discussion

7.1 Results Discussion

We have brought significant improvement with a percentage point increase of 4.7 in precision, 5.3 in recall and 4.95 in F1 score, comparing the best level 0 model (commit classifier) and the logistic regression ensemble model. The different ensemble models achieve similar results. The neural network did not outperform simpler models such as logistic regression and random forest. Our theory is that it is due to the simplicity in the input data for the ensemble, which only contains numbers without immensely complex relations. A neural network does not show its superiority in finding complex relations since there exists none.

The manual inspection showed that inaccurate predictions of the ensemble were largely an extension of misclassifications of the Level 0 models. Some "errors" in the commit classifier could be fixed to increase the performance of the ensemble, for example that it predicts a high confidence when the message length is zero. Though, we could also show that a high percentage of the misclassifications are due to ambiguous commit messages. The ambiguity in the commit messages makes it harder to accurately predict with the commit classifier. Therefore, we also argue that in order to further increase the performance, it is a necessity to include a model in the ensemble that can accurately predict vulnerabilities by looking at the actual code.

Instead of further exploring more complex semi-supervised learning models than PU-learning for the ensemble, we opted to keep the ensemble classifier simple and investigated the impact of an additional level 0 model. Since the level 0 code change classifier only takes the actual change into consideration with a bag-of-word technique, our assumption was that there could be more information available if a model could analyse the entire changed function.

The addition of our implementation of AST classification did not provide an increase of the achieved result. However, there is a low difference, only 0.2 percentage points, in F1 score between the best ensembles with and without AST-classifier. Furthermore, we were

able to show that some signal can be found analysing the semantics of changed functions. It is important to remember that the results presented are those from a model in the early stages. Still, this model shows that signal can be found in code semantics. Further development on this would be necessary to explore the full potential.

We could see that the standard deviation in F1 was significantly reduced with the ensemble architecture, compared to the level 0 models. The standard deviation is to say that the robustness of the predictions are higher. Adding the AST classifier, we could see an increase in standard deviation since we were adding a weak performing model.

We extracted the bigrams to investigate the common word pairs of commits predicted with high and low confidence from an unseen data set. From this, we can read that the entities with high confidence mention key-words like *buffer read* and *buffer size* which showed common in security entity commits during the manual inspection. The Bottom confidence bigrams showed more general bug fix and commit terms such as *add support* and *merge branch* which was a nice confirmation of the metrics results.

7.2 Setbacks and mistakes

7.2.1 Dataset

As mentioned in the manual inspection results, a part of the false positives predicted had similar appearance to true positives with commit messages mentioning various CWE keywords. These findings made us question the initial assumption that a sample of commits without a link to the NVD would include an insignificant part of security commits. What we take from this is that it was a mistake to assume that commits not included in the NVD were mainly non-security commits. A more fundamental research and EDA of the unlabeled data could have helped us capture this issue in an earlier stage.

7.2.2 Relational Links

A setback in this project was the lack of links between issues and commits. This lack of relations made us deviate from our initial plan of developing a graph neural network for the ensemble classifier and instead settling for a more linear data structure. The reason for this is that graphs of commits and code changes, in most cases, would have been a one-to-one graph with one commit and one code change, which essentially is the completely linear data structure that we used in the end. Linking commits to issues, would also mean that more commits could have been connected. This possible link would not only add a completely different level 0 classifier of the issue messages but also the relation between commits linking to the same issue could have been utilized.

7.3 Validity and Limitations

7.3.1 Validity

As mentioned, we have some indications that some data points are mislabeled. Because of this, it is hard to know what part of the inaccurate classifications that are actually due to wrong predictions and what are due to mislabeling. Therefore we cannot assure the performance in absolute terms.

We regard NVD as a ground truth and can therefore be certain that the positive data set is correct as NVD states them as security related, but we cannot be certain that the negative data set is completely correct since it essentially is unlabeled. Therefore, the mislabeling would most likely mean that the presented scores have lower precision but higher recall. This is because false positives would decrease but false negatives would increase, if the potentially mislabeled data points were corrected.

Because of this uncertainty, it was important for us to perform a manual inspection. By manually analysing commits from October and November 2020, we could see indications that the model would be useful as an automated approach to vulnerability detection. We found examples that can be clearly linked to CWEs, which is promising. Though, we can not be 100% certain that this is the case, since a qualitative study on a larger sample would be necessary to ensure the absolute performance. However, we can be certain of increased relative performance from the level 0 models.

As in any machine learning problem, there is a risk of data leakage. Measures have been taken to reduce this risk but we can never be 100% sure.

7.3.2 Repository Distribution

The sizes of the tracked repositories vary significantly, which means that a few repositories account for a large part of our dataset both regarding vulnerability and non-vulnerability commits. We cannot assure validity among all C repositories as the data points are largely unbalanced between repositories. Some of the smaller repositories will naturally have less registered vulnerabilities and will thus have less weight in the training of a model.

This means that if some repositories have a unique way of writing commit messages. For example, the commit classifier will possibly be unable to evaluate if the commits are security related or not since its not heavily represented in the training data.

7.3.3 Language restriction

The final results are exclusively from commits of changes in C language repositories and the reason for this is that we prioritized getting results from a single language as proof of concept rather than spending time on training multiple code change classifiers. The ensemble results therefore cannot be assumed to be identical for all programming languages since the level 0 models will be trained on different of data with unique structures.

Although, since the data structure to the ensemble model will be identical, we believe that if the code change models of different languages can produce similar results as the one trained on C language, the ensemble should be able to increase the performance.

Chapter 8
Conclusion

We have investigated an automated approach to security classification by utilizing information in different parts of git commits. Through our proposed ensemble architecture, we have proven significant performance improvement, compared to level 0 models in the ensemble.

8.1 Future Work

8.1.1 Data

In order to mitigate the risk of data leakage between the Level 0 models and the ensemble, it was necessary to split the data set. Inevitably, this results in smaller data sets. A larger data set would be desirable, as machine learning models often increase performance with more data points.

In addition, the lack of completeness of links between issues and commits made it impractical to utilize the predictions from the issues classifier. We believe adding the issue classifier to the ensemble could increase the performance. Therefore, it would be desirable to increase the completeness of these links. This could be achieved by improving the scraping from Github.

Although it would be a tedious task, we believe that a properly labeled data set by a security expert would give a more robust model and an increased performance if some of the false positives turn out to be true. This work would most likely not increase the relative performance to the level 0 models, but could potentially increase the absolute performance, and be more valuable for Debricked.

8.1.2 AST Improvement

We could observe that there is signal available in the abstract syntax tree. Though, due to time limitation, we put only a small amount of testing and development into this model. Further

development would be interesting to see if it could contribute to increasing the performance of the ensemble.

As mentioned in Section 7.2.1, one development possibility of improvement is to work on the dataset. Another way of attacking the problem with unlabeled commits could be to implement a model with pseudo-labeling. In this way, only a small dataset could be manually labeled and then the model would have a ground truth data set in combination with a pseudo-labeled data set to train on. However, ideally the evaluation should in any case be done on a properly labeled data set.

8.1.3 Additional Languages

We have evaluated the performance of the model on a data set containing `.c` files. Expanding to other languages would increase the usefulness of the model further.

To expand this ensemble model to work on other languages, the code change models would have to be extended. The bag-of-words cc-model needs to be trained on a language specific data set for each language and the AST classifier will need a tool for fuzzy parsing functions for each language as well as a language specific model.

References

8080labs (2020). ppscore. `https://github.com/8080labs/ppscore`.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Duppils, A. and Tullberg, M. (2020). Semi-supervised text classification: Automated weak vulnerability detection. Master's thesis, Lund University.

Grossi, E. and Buscema, M. (2008). Introduction to artificial neural networks. *European journal of gastroenterology & hepatology*, 19:1046–54.

Hoang, T., Lawall, J., J. Oentaryo, R., Tian, Y., and Lo, D. (2019). Patchnet: A tool for deep patch classification. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 83–86.

Li, F. and Paxson, V. (2017). A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference*.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747.

Synopsis (2020). Open source security and risk analysis report. `https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2020-ossra-report.pdf`.

Zhong, Z., Li, C.-T., and Pang, J. (2020). Hierarchical message-passing graph neural networks. *arXiv:2009.03717*.

Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., and Sun, M. (2018). Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434.

Zhou, Y., Liu, S., Siow, J. K., Du, X., and Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *CoRR*, abs/1909.03496.

EXAMENSARBETE Vulnerability Detection Using Ensemble Learning
Automatisk detektering av sårbarheter i öppen källkod med hjälp av maskininlärning
STUDENT Rasmus Lindqvist, Viktor Bard
HANDLEDARE Pierre Nugues (LTH), Emil Wåreus (Debricked)
EXAMINATOR Jacek Malek (LTH)

Automatisk detektering av sårbarheter i öppen källkod med hjälp av maskininlärning

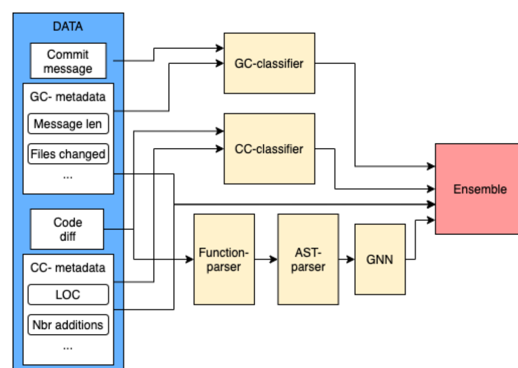
POPULÄRVETENSKAPLIG SAMMANFATTNING **Rasmus Lindqvist, Viktor Bard**

Användadet av öppen källkod ökar ständigt då det bidrar till ett billigt och snabbt sätt att utveckla mjukvara. Samtidigt blir är det svårare att hantera säkerhetsrisker i denna kod då koden är tillgänglig för utomstående parter att hitta delar som kan exploateras. Detta arbete undersöker potentialen i att kombinera flera olika maskininlärningsmodeller för att automatiskt hitta sårbarheter i kod.

Enligt en undersökning av Synopsis innehöll 99 % av analyserade kodbaser öppen källkod. Manuell detektering är dyrt och opraktiskt när mängden kod växer. Automatisk detektering av sårbarheter i öppen källkod har potentialen att vara billigare och bidra till större säkerhet.

Vi använder oss av data från GitHub och en kombination av maskininlärningsmodeller till olika delar av den tillgängliga datan för att med större säkerhet kunna bedömma riskerna än vad dessa modeller kan klara av var för sig. Den första modellen tittar på meddelanden som utvecklarna skriver när de laddar upp sin kod och den andra tittar på själva ändringen i koden. Resultaten från dessa modeller är en prediktion på hur sannolikt det är att specifika ändringar i kodbasen kan medföra säkerhetsrisker. Dessa prediktioner används tillsammans med tillhörande metadata så som antal ändrade filer och rader kod. Slutligen testar vi inverkan av att lägga till en ytterligare enskild modell som klassificerar en kodändring som säkerhetsrelaterad eller inte, baserat på syntaxträd - kodens uppgyggnad.

I den slutgiltiga arkitekturen, visat i figuren, använder vi oss av de tre olika modellerna, Git-commit (GC), Code-change (CC) och syntaxträd(AST), som input till ensembeln. Vi experimenterar med olika maskininlärningsalgoritmer till ensembel-modellen och logistisk regression visar sig vara den sammantaget bästa modellen.



Resultaten visar att modellen kan hitta 85.2 % av alla sårbarheter med en precision på 83.6 %. Detta är en ökning med 5.3 procentenheter fler hittade sårbarheter och 4.7 procentenheter bättre precision, jämfört med den bästa tidigare modellen. Med hjälp av en säkerhetsexpert analyserar vi prediktioner på oannoterad data och hittar säkerhetsbrister i kända kodbaser.