technische universität
dortmund

Masterarbeit

Explainable Vulnerability Detection on Abstract Syntax Trees with Arithmetic, LSTM-based Neural Networks

Simon Schröder

6. September 2019

Betreuer:
Prof. Dr. Falk Howar
Simon Dierl, M. Sc.

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 14 für Software Engineering
Arbeitsgruppe AQUA
https://ls14-www.cs.tu-dortmund.de/

Abstract

To support developers in creating vulnerability-free software in our increasingly digitial world, we create and evaluate an static analyzing tool based on Artificial Neural Networks (ANNs). Since many researcher already showed the general feasibility of such tools, we propose and evaluate a novel combination of techniques.

We construct two feature representations based on Abstract Syntax Trees (ASTs) differing in the amount of manually crafted features and the ANN's structure. The Long Short-Term Memory (LSTM) technique allows our tool to capture long-distance relationships within source code snippets. To support our tool in learning basic arithmetic operations, we employ and evaluate Neural Arithmetic Logical Units (NALUs). We utilize the Integrated Gradients attribution technique to put explanatory overlays on the source code snippets with the goal of overcoming the lack of ANNs' understandability.

Our evaluation showed our feature representations suffering from overfitting depending on the their complexity and the size of the utilized synthetic datasets. Roughly, the simpler feature representation worked better on small datasets while the other one achieved higher performance on bigger and more complex datasets. The presence of NALUs did not worsen our tool's prediction power and ability to generalize. In many cases, their presence resulted in better scores compared to without them. However, we could not conclude that NALUs increase our tool's extrapolation capability. In comparison with conventional and ANN-based static analyzers from the literature, we showed or tool better or equally. The explanatory attribution overlays helped us understanding our tool's predictions in many cases.

By evaluating the use of our tool in Integrated Development Environments (IDEs), we showed the practical contribution of our tool. Future work comprises countermeasures of overfitting and evaluation on other (natural) datasets.

Contents

Chapter 1

Introduction

Software systems are omnipresent in almost every aspect of modern life such as (automotive) transportation, energy supply, and healthcare systems [GP15]. As a result, a growing amount of sensitive personal and business-related data is processed, exchanged, and stored by software. Since these valuable data are a promising target for attackers, criminality in this field is increasing as well as the costs of combating it [PA17].

To mitigate both, one approach is the reduction of a software's vulnerabilities in the first place since these are necessary for successful attacks [Avi+04]. During a software's development process this can be done with the help of vulnerability detection techniques. These techniques aim at creating highly reliable and correct software by detecting vulnerable source code snippets and the weaknesses[1] they belong to [Avi+04; All+18]. Such detections support software developers in their work of creating vulnerability-free software systems.

One vulnerability detection technique is the static analysis of a given software's source code [Avi+04]. This analysis provides a "scalable way for security code review that can be used early in the life cycle, does not require the system to be executable, and can be used on parts of the overall code base" [GP15]. However, the presence of a vulnerability does not only depend on the source code snippet itself but also on dynamic influences during its execution. Examples for such influences are user inputs, random generators, concurrency, and interactions with other software.

For these reasons, among others, it is generally undecidable whether a given source code snippet contains a (given) vulnerability [CM04; GP15]. Thus, the creation of a static analysis that is both complete and sound is impossible in general. A *complete* analyzer produces no *False Positives* (FPs), i.e., it does not report a vulnerability that the source code snippet does not contain. A *sound* analyzer generates no *False Negatives* (FNs), i.e., it reports all vulnerabilities that the source code snippet

[1]The term *weakness* denotes a class of vulnerabilities in this thesis. See section 5.1 for more details.

contains [SSV18; CM04]. Since the reduction of both is usually contrary, multiple metrics which differ in their focus on FPs and FNs exist. For example, the F_1 metric weights FP and FN equally.

Current commercial state-of-the-art static analyzers mostly build on flow-based program verification, or they employ a set of more-or-less manually crafted rules derived from known weaknesses [GP15]. Goseva-Popstojanova and Perhinschi benchmarked three such static analyzers on two datasets and observed performances comparable to or worse than random guessing [GP15].

Motivated by this and by the success of machine learning techniques in other fields, many static analyzers using Artificial Neural Networks (ANNs) were presented in the literature during the last years [Rus+18; Mou+16; WLT16; WL17; Li+18; Har+18]. These are capable of learning generalized rules for vulnerability detection from a set of vulnerable and non-vulnerable source code snippets with the result of performances better than random guessing. Drawbacks of ANNs are that they neither guarantee sound nor complete results and a poor understanding of their predictions, as the learned generalized rules are usually not intuitively recognizable by humans [Bis+95].

In the context of ANN-based vulnerability detection, we elaborate five research questions of this thesis in section 1.1. They further tighten the scope of this thesis and allow well-structured references to our core scientific tasks during this thesis. Section 1.2 subsequently presents the structure of this thesis.

1.1 Research Questions

We derive our research questions (RQs) along existing literature by showing combinations of techniques that have remained unexplored so far. Afterwards, we summarize them into the main objective of this thesis.

One static analyzer using ANNs has been proposed by Russell et al. [Rus+18]. They achieved good results on multiple datasets by employing a Recurrent ANN (RNN). To feed an source code snippet to their RNN, they use a token-based feature representation by splitting each source code snippet into a sequence of tokens. For each token, they build an integer-valued feature that uniquely identifies the token's kind or the token's value if any. With a so-called embedding they map each feature integer to a feature vector. The embedding mapping is dynamic such that it can be trained to automatically map similar tokens to similar feature vectors and vice versa.

By only considering the tokens' kinds and literal values, their feature representation ignores a lot of an source code snippet's information. A variable's or function's name and type as well as values of float, boolean, and string literals are ignored. Those features are important to distinguish vulnerable and non-vulnerable source code snippets. For example, variable names are crucial for the reliable detection of an erroneous dereferencing of a variable previously assigned null. Therefore, this thesis addresses the problem by creating and using a feature representation composed of all the above-mentioned information.

Wang, Liu, and Tan also presented an ANN-based approach for vulnerability detection [WLT16]. In contrast to Russell et al., their feature representation extracts information from the nodes of the source code snippet's Abstract Syntax Tree (AST). Such a tree abstracts from the concrete syntax of an source code snippet and incorporates semantic information and relationships of the snippet's entities.

We favor an AST-based approach similar to Wang, Liu, and Tan for this thesis. As a consequence, we can incorporate the semantic contexts in which the entities are declared to distinguish identically named variables across functions. ASTs also allow function calls to be easily matched with their respective callee function in many situations. This enables the detection of inter-function vulnerabilities.

Both authors' ANNs have one single input layer for all their features. Since we incorporate diverse information to build features, we expect separate input layers for different features to be beneficial. For example, while layers with embeddings are successful for features representing kinds, we do not expect so for features representing integer literals.

Based on the above discussion, we work out our first RQ as follows.

RQ1 Which features can be constructed from the information in an source code snippet's AST? How does a simple single-input-layer feature representation compare to one with multiple input layers and embeddings? How do they differ in terms of vulnerability detection performance and the training's resource consumption?

Another approach to be used in this thesis is Long Short-Term Memory (LSTM). Hochreiter and Schmidhuber presented this memorizing technique which is able to capture long-distance relationships between an RNN's sub-inputs [HS97]. This is particularly relevant to this thesis since a program's semantically related parts often lie far apart in the source code. Besides, LSTM-based ANNs have recently been successfully used in the field of source code processing [Dam+17; WL17] and the somewhat similar Natural Language Processing (NLP) [Gho+16; SNS15].

To the best of our knowledge, using the above-mentioned AST-based feature representations together with LSTM-based ANNs for vulnerability detection has not been done yet. We therefore evaluate this approach as summarized below.

RQ2 How does a static analyzer using feature representations as in RQ1 and employing an LSTM-based RNN perform in terms of vulnerability detection? How do the RNN's Hyper-Parameters (HPs) influence that performance?

Another important ability necessary for detecting vulnerabilities in source code is basic math. For example, if a C++ array is declared with length eight and its ninth element is later accessed in a loop, an overflow occurs. In general there might be a vulnerability if the distance between the array's declaration length[2] and the greatest index used for accessing the array is greater than or equal to zero. As can be seen from this example, the ANN needs to be able to perform a subtraction and a comparison to detect vulnerabilities of this specific weakness.

According to Trask et al., ANNs are usually not able to generalize basic arithmetic operations well [Tra+18]. In particular, extrapolation of the operations to the range outside the values encountered during training works poorly. As a countermeasure they developed a new kind of neurons called Neural Arithmetic Logic Unit (NALU).

We therefore evaluate whether the presence of such NALUs benefits the vulnerability detection ability of our static analyzer.

RQ3 How does the presence of at least one layer of NALUs affect the vulnerability detection power? Does it allow the ANN to extrapolate its predictions to ranges not encountered during training?

While ANNs often accomplish very good results in various tasks, it is usually not obvious how they come to their predictions/outputs. One way to mitigate this is the use of the gradient-based attribution techniques Gradient*Input and Integrated Gradients [Anc+18]. Both employ derivations of the ANN's output with respect to a specific feature of the source code snippet. The resulting gradients correspond to the feature's contribution to the analyzer's prediction.

By keeping track of each feature's corresponding location in the source code snippet, we can place an overlay on it [Rus+18]. In case of a predicted vulnerability, such an overlay marks the source code snippet's entities, whose specific values highly contribute to the ANN predicting a vulnerability. If embedded in an Integrated Development Environment (IDE), this supports the developer in localizing and understanding the vulnerability.

Our assumption is the attribution techniques improving our evaluation, and we investigate in this assumption as follows.

[2]There may be multiple lengths since a declaration may consists of multiple array declarators each specifying another length. In this thesis, we consider each declarator as a separate declaration in accordance to Clang's AST representation.

RQ4 How much do the gradient-based attribution techniques and their resulting overlays contribute to understanding an ANN's prediction? What is their time consumption, especially with respect to using them in an IDE?

Until here, the RQs were only about appraising our static analyzer itself. Thus, our last RQ is about the comparison with static analyzers from the literature. To enable a fair comparison, both our results and results from the literature must have been achieved on the same dataset. In order to profit from our previous experience with the Clang compiler frontend, we decide to only select datasets containing source code snippets written in C or C++.

Russell et al. performed benchmarks on the Juliet Test Suite for C/C++. This synthetic dataset consists of vulnerable and non-vulnerable source code snippets for 118 weaknesses. They also crawled their own dataset from Debian and GitHub repositories. Unfortunately, the published version of the latter only provides incomplete source code snippets. This renders them semantically ill-formed such that we cannot use them with our AST-based feature representation. We therefore identified only the Juliet Test Suite as this thesis' first dataset. We compare our results to Russell et al.'s results on that dataset.

Goseva-Popstojanova and Perhinschi used the same dataset for comparison of three non-ANN-based static analyzers, which makes their results the second results, we compare ours to.

Choi et al. synthetically created their own dataset in their paper about detecting vulnerabilities of a buffer overflow weakness with an ANN variant. The dataset is this thesis' second dataset and Choi et al.'s results on their dataset serve as the third results, we compare ours to.

Other datasets' source code snippets are neither written in C or C++ nor semantically well-formed [Li+18]. Others are not publicly available [Mou+16]. Unfortunately, this results in all our datasets' source code snippets being synthetic. In comparison to natural source code snippets written by humans, those snippets are simpler and less versatile [Nat12]. As a consequence, statements made in this thesis are limited to synthetic source code snippets and cannot necessarily be transferred to natural source code snippets found in the real world.

The following is our last RQ.

RQ5 How does our static analyzer's vulnerability detection power compares to the results by Russell et al., Goseva-Popstojanova and Perhinschi, and Choi et al.?

This thesis' objective is formed by our RQs. We summarize it into the following sentence:

The objective is the construction and evaluation of a NALU- and LSTM-based static analyzer for vulnerability detection which utilizes an AST-based feature representation and an attribution-based explanation of its prediction.

1.2 Structure of This Thesis

Analogously to this section giving an outline of this thesis' structure, chapter 2 provides an overview of our static analyzer's components and pipelines. The subsequent chapter 3 describes the work done by other researchers and discusses connections and differences to the goal of this thesis. Chapter 4 introduces fundamentals of an ANN's structure, training, and evaluation metrics. It further considers the two attribution techniques used for explaining our ANN's predictions. The datasets utilized for training and evaluation of our static analyzer's ANN are described in chapter 5. That chapter also explains the ASTs on which our feature representations are based. At that point, we described our static analyzer's inputs and functionality in theory.

The subsequent chapter 6 shows how our implementation transfers the theory into practice. The implementation is the foundation of our evaluation, which follows in chapter 7. That chapter contains our RQ-based experiment design and the analysis of the experiments' results. It closes with a discussion of the threats to the validity of our results. Lastly, chapter 8 summarizes this thesis' conclusions and provides an outlook to further work.

For completeness, we subsequently list the abbreviations and the bibliography that we use in this thesis. The following appendix contains verbose versions of our experiment results. Finally, we give an overview of the figures, listings, and tables that are present in this thesis.

Chapter 2

Overview

This chapter gives an overview of our static analyzer's components and of this thesis' two pipelines, which utilize them. For each component, we mention their relevant chapters and sections. Therefore, this chapter serves as a kind of index.

For brevity, we refer to our static analyzer as our *tool*. The term *tool configuration* denotes a specifically parameterized variant of our tool. Such a variant is described by a specific combination of values of our tool's RNN's HPs, of its RNN's weights, and of its training process' HPs. We abbreviate source code snippet by SCS.

Figure 2.1 visualizes the two pipelines in our tool. The first pipeline is drawn on the upper half in solid black. It trains an RNN model based on one of the datasets and tests the trained model's performance. The lower half depicts the second pipeline in dashed red. It is based on the first pipeline as it requires an already trained model. For a given SCS the pipeline allows the classification in terms of its vulnerability and the placement of an explanatory overlay over the snippet. Such an overlay tries to explain how much each source code entity contribute to the model's prediction. The following sections 2.1 and 2.2 describe both pipelines as well as their components and data flow in more detail.

2.1 Training and Testing Pipeline

The upper pipeline receives a dataset as input and outputs a model trained on the dataset together with scores quantifying the success of the training in terms of generalization.

Each **dataset** is a collection of pairs of SCSs and their associated labels. The label indicates whether its associated SCS is vulnerable and if so, to which weakness it belongs. The SCSs are written in C or C++ and usually consist of a function definition

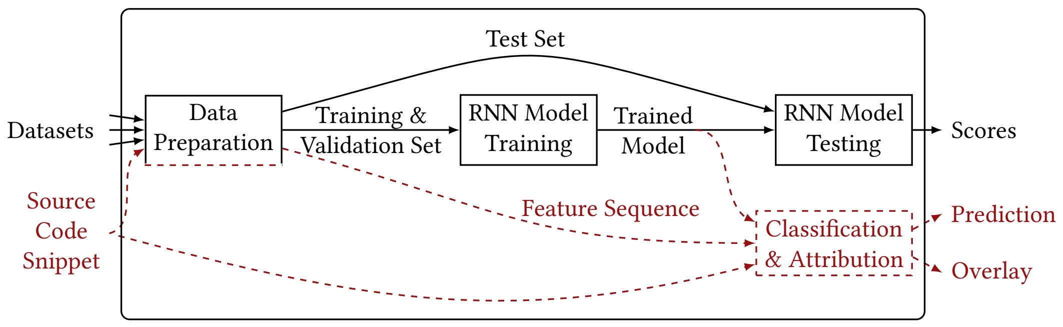


Figure 2.1: Components (rectangles) of our tool (rounded rectangle) and pipelines (arrows) utilizing these components. Solid black denotes the pipeline for training an RNN model and testing its performance. Dashed red indicates the pipeline for classification of an SCS and generation of its attribution overlays.

together with its callee functions' definitions. Section 5.1 describes the datasets utilized in this thesis and their containing SCSs in more detail.

The **data preparation** component converts a dataset to the training, validatation, and test sets, which are consecutively used for training and testing an RNN model. This component is also capable of preparing a single SCS such that it can be input to the classification & attribution component described below. We base our use of the term *data preparation* on the book by Pyle [Pyl99]. The sub-components are data discovery, data representation, and data segmentation, which we further describe in the following paragraphs.

The datasets are given as a nested directory structure of both source code files and files containing corresponding labels. The data discovery component traverses the directory structure, extracts SCSs from source code files, matches them with labels, and provides them in a uniform, dataset-independent way for our feature representations. This is done by forming a set of pairs consisting of an SCS and its associated label. Section 6.1 describes the data discovery in more detail.

During data representation, each SCS is parsed to an Abstract Syntax Tree (AST), which in turn is represented by a feature sequence, and which in turn consists of a feature vector for each AST node. These feature sequences and their associated labels are the output of the data representation component. Section 5.2 presents our two AST-based feature representations which we employ for data representation.

Before leaving the data preparation component, the data segmentation component splits the set of feature-sequence-label pairs intro three disjoint training, validation, and test sets. The training and validation sets are the inputs for the training of the tool's RNN while the test set serves for testing the trained model's performance.

Since we do not use the same split for all datasets, we describe the different split approaches in the respective dataset sections 5.1.1 to 5.1.3.

The **RNN model training** component trains the tool's RNN on the training set. For this, the component repeatedly adjusts the model's weights with the goal to successfully predict the training feature sequences' vulnerabilities. During the training process, we occasionally use the model to detect vulnerabilities in the validation set's feature sequences and thus measure its performance on that set. The training component's output is not the training's final model – which usually performs best on the training data – but the model that achieves the best performance on the validation set. Through this we avoid outputting a model that only memorized the training data without learning generalized rules. We explain the theoretical foundations for training RNNs – and ANNs in general – in section 4.2. Section 6.2 additionally describes the implementation of our tool's training process.

Since we used both the training set and the validation set for the selection of the training's best model, we use the **RNN model testing** component to access the trained model's performance on data which the model has never seen before. We employ multiple metrics and use the resulting scores[1] to compare different tool configurations to each other and to tools from the literature. Section 4.2.6 describes the methodology of these metrics. The evaluation of different tool configurations and their comparison with the literature is located in chapter 7.

2.2 Classification and Attribution Pipeline

The second pipeline receives a single SCS and outputs a prediction in terms of its (non-)vulnerability. Besides, it generates an overlay indicating the SCS's entities' contribution to the prediction.

A trained model is not only used for testing its performance but to also classify a given feature sequence in terms of its (non-)vulnerability. This is done in the **classification & attribution** component. A feature sequence is the output of the above described data preparation component for a single SCS. Next to classification, the classification & attribution component utilizes two attribution techniques to try to explain which entities of the SCS highly contribute to the trained model's prediction. To visualize these contributions by placing an overlay over the SCS, the component also receives the SCS as input. Section 4.5 introduces the two attribution techniques. We evaluate their resulting overlays and their time consumption during our experiments' analysis in section 7.2.

[1]We use the term *metric* to denote a method of measurement and the term *score* to denote a specific value computed by such a method.

Chapter 3

Related Work

This chapter gives an overview of other researchers' work which is related to this thesis but not in its central focus.

3.1 Natural Language Processing

The goal of *Natural Language Processing* (NLP) is to convert natural language, which is written or spoken by humans, to a representation that allows easy further processing by machines. This brief overview is based on Collobert and Weston's paper [CW08]. Sub-tasks of NLP are – among other – tagging each word with its syntactic role or building language models, which allow the prediction of the next word given a sequence of words. Collobert and Weston train a CNN that is capable of jointly performing these and other sub-tasks. Similar to this thesis, they use embeddings to allow the training algorithm to autonomously learn similar feature vector representations for similar words.

While both natural languages and programming languages exist to communicate and to exchange information, their respective communication partners differ [All+18]. In case of the former humans interact. The latter are designed to allow communication between humans and machines, and they are therefore more structured and non-ambiguous. Thus, we refrain from directly applying NLP approaches in this thesis.

Huo, Li, and Zhou bridge the gap between NLP and programming languages in their approach [HLZ16]. It uses a bug report written in natural language to localize corresponding source code files with the goal to support software maintainers in their work. Their ANN learns both tasks in a unified way and outperforms state-of-the-art bug localization techniques.

3.2 Clone Detection

The goal of *clone detection* is to detect two SCSs which implement the same functionality given some similarity measure. This paragraph briefly describes clone detection according to Roy, Cordy, and Koschke [RCK09]. Usually the SCSs are present in the same software project and the detection of clones is motivated by the removal of redundancy. There are four types of clones which are increasingly hard to detect. Type 1 clones are token-wisely equal while type 4 clones only share the same semantic functionality.

If a vulnerable SCS is given, detecting clones of this SCS corresponds to the detection of similar vulnerabilities. Kim et al. successfully apply this approach to type 1 and type 2 clones [Kim+17]. Vice versa, if two SCSs have vulnerabilities of the same weakness, they will not necessarily implement the same functionality. In general, vulnerabilities do not correlate with the overall functionality of their containing SCS.

There are multiple clone detection approaches using an AST-based feature representation and/or employing LSTM-based RNNs [SK16; WL17; Bax+98]. The foundations of their feature representations are similar to ours but their ANN structures are more complicated since they need to input two SCS at the same time.

In summary, their goal and techniques do not align well enough with the goal of this thesis.

3.3 Detection of Syntax Errors

Gupta et al. create an RNN-based tool for detection of syntax errors in SCSs [Gup+17]. They use a token-based feature representation and utilize common compilers to generate their training data since these reliably detect syntax errors. In addition, their tool is capable of localizing and repairing these errors. Their goal differs from our goal since they completely ignore semantic errors. Thus, their approach does not allow the detection of vulnerabilities.

3.4 Vulnerability Detection

The most related work is done in the field of vulnerability detection. As mentioned above, we exclude this thesis' central papers by Russell et al. and Choi et al. here [Rus+18; Cho+17].

Mou et al. use an AST-based feature representation but only incorporate the AST nodes' kinds [Mou+16]. In contrast to our tool, they directly feed the ASTs to their ANN without sequencing it first. This is possible through their special tree-based, convolutional ANN, which they abbreviate as TBCNN. By using convolutional layers they also make advantage of the layers' translation invariance, since vulnerabilities usually do not depend on their position in the SCS. They crawl a dataset of SCS written in C and their tool achieves good results on that dataset. Since we are unable to retrieve the dataset we do not compare our results with theirs.

Pradel and Sen use a name-based approach by assuming that identifier names contain information about their corresponding variable's or function's semantic purpose [PS18]. They train an ANN that is capable of detecting mismatches between a function parameter's actual purpose and the suspected purpose of the identifier passed as argument. The results show their approach achieving high accuracies on their synthetic dataset. Since in C and C++ identifiers can be renamed without changing the SCS's behavior we do not consider their approach for our tool.

Similar to this thesis, Wang, Liu, and Tan utilize ASTs for their feature representation that contains information about names, strings, and integers [WLT16]. They employ a variant of an ANN as a classifier. They achieved good results and also compared their AST-based feature representation to 20 traditional features available for the PROMISE dataset. These traditional features include "lines of code, operand and operator counts, number of methods in a class, the position of a class in inheritance tree, and McCabe complexity measures" [WLT16]. They found their AST-based feature representation to perform better in almost all cases. We do not compare our results with theirs since they utilize SCSs from the PROMISE dataset written in Java for their evaluation.

Sestili, Snavely, and VanHoudnos unsuccessfully tried to apply the memory network approach by Choi et al. on the buffer overflow related CWEs of the Juliet dataset [SSV18]. They state that the dataset is too complex – in terms of functional variant count – and too small – in terms of SCSs per functional variant – for the training of the memory network to converge. This shows our tool more flexible than the memory network tool since we are able successfully detect vulnerabilities in both the Juliet dataset and the MemNet dataset.

3.5 Attribution

Gradient-based attribution techniques are successfully used in the field of image classification, in which images are classified based on the object they are showing. Zhou et al. create so called Class Activation Maps (CAM), which visualize the pixels' contributions to the predicted output's neuron's activation [Zho+16]. Based on these they are able to localize the predicted object in the image. Similar, Simonyan, Vedaldi, and Zisserman use saliency maps to segment the image into object and background [SVZ13]. Besides the already mentioned work by Russell et al., we did not find other approaches that utilize attribution techniques in the field of vulnerability detection.

Chapter 4

Fundamentals

This chapter describes the theoretical foundations on which this thesis bases. Since ANNs are this thesis' central method, all foundations are related to them.

The first sections 4.1 and 4.2 explain the structure of ANNs and how they can be trained to a specific task, respectively. We base these explanations on the simple Feed-forward ANNs (FNNs). Afterwards, section 4.3 demonstrates why they work poorly in arithmetic contexts, and it presents Neural Arithmetic Logical Units (NALUs) as solution.

Since NALUs – and FNNs in general – are not capable of receiving sequential data such as an SCS as input, we introduce Recurrent ANNs (RNNs). Basic RNNs, however, suffer from vanishing or exploding gradients, which motivates the need for the Long Short Term Memory (LSTM) technique. We describe RNNs and the LSTM technique in section 4.4.

At that point, all presented methods and techniques allow us to train our tool such that it can successfully predict vulnerabilities. However, one drawback of ANNs is the poor understanding of their predictions, as the learned rules are usually not intuitively understandable by humans. To overcome this, section 4.5 introduces attribution techniques, which allow our RNN to explain and visualize each of its predictions.

We base our following explanations on Bishop et al. and Gardner and Dorling, unless otherwise noted [Bis+95; GD98]. We unified and adjusted their notation to achieve a consistent notation within this thesis.

4.1 Structure of ANNs

An ANN is a machine learning model in form of a directed graph which can be used for approximation of a target function *F*. Usually, there is no analytical definition of

F available. Instead, only its conceptual behavior as well as a large set of example inputs and associated outputs are given. A conceptual behavior could be for example "Correctly map a picture to labels `cat` and `dog`", "Correctly map an English sentence to its German translation", or "Correctly map an SCS to labels `vulnerable` and `non-vulnerable`".

ANNs are inspired by the natural neural networks in the human brain. Therefore, the directed graph's nodes are called neurons or units. In an ANN these are usually organized in layers and connected by weighted edges which are sometimes called synapses. The weights together with the ANN's HPs form the function $\hat{F}$ realized by the ANN.

At this point, we describe the structure of FNNs, whose graph is acyclic such that the FNN's input is fed forward through the network. In contrast, RNNs have circles in their graph resulting in neurons receiving their own past output as input. This corresponds to neurons having some kind of memory which allows RNNs to extract time or spatial information from sequential input data more easily. The following descriptions of FNNs mostly also apply to RNNs. We show the differences to FNNs in the already mentioned section 4.4.

One of the most widely used form of an FNN is the so-called *MultiLayer Perceptron* (MLP) [Gra08]. To start with a rough overview, figure 4.1 shows an MLP with its neurons depicted as circles. The labeled arrows represent the MLP's weighted edges where each label denotes the real-valued weight of its associated edge. The MLP's neurons are organized in *layers* such that each neuron belongs to exactly one layer. The layers are arranged sequentially from left to right. Furthermore, MLPs are defined to be *Fully-Connected* (FC), i.e., each neuron in a given layer is connected with all neurons in the following layer. An exception to this are the so-called bias neuron in the first row.

Figure 4.1 shows the first and the last layer different from the ones in between. The first layer is called *input layer* and its neurons, the *input neurons*, have a simple structure. Each of them holds and outputs a component of the MLP's overall input $\vec{x} \in \mathbb{R}^M$. The uppermost input neuron corresponds to $\vec{x}$'s first component x_1. The lowest of the M input neurons corresponds to x_M. In the figure $M = 3$ holds.

Analogously, the MLP's last layer's neurons' outputs form the overall output $\vec{y} \in \mathbb{R}^N$. In the figure 4.1 $N = 2$ holds. All other layers are called *hidden layers* because they are neither exposed as the MLP's input nor its output. An MLP consists of at least one hidden layer.

$\hat{F_{\underline{W}}} : \mathbb{R}^M \to \mathbb{R}^N, \hat{F_{\underline{W}}}(\vec{x}) = \vec{y}$ denotes the function realized by the MLP. It maps the MLP's input $\vec{x}$ to its output $\vec{y}$ depending on the MLP's HPs and its weights $\underline{W}$[1].

[1]We denote matrices with an underscore to distinguish them from scalars.

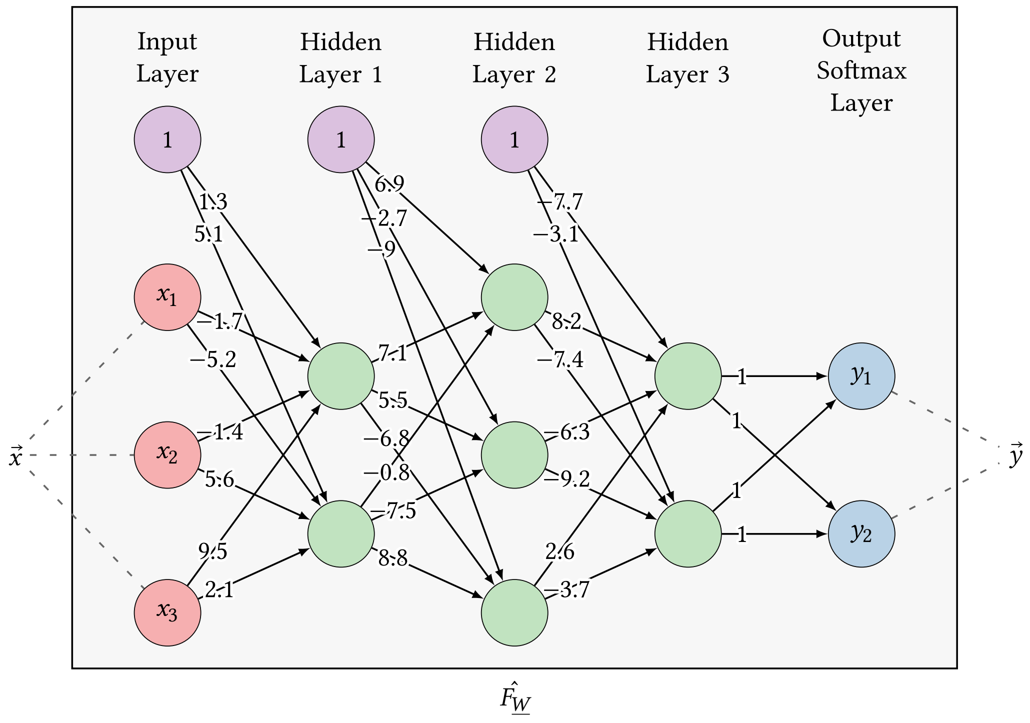


Figure 4.1: Structure of an MLP with three hidden layers. Input, softmax output, hidden, and bias neurons are depicted by red, blue, green, and purple circles, respectively. Black arrows denote weighted edges.

We now describe how each input propagates through the MLP to the back. Real-valued signals flow along the directed edges through the MLP starting at the input neurons and ending at the output neurons. During this process, which is called *forward pass*, the edges as well as the neurons interact with the signals.

When a signal traverses an edge, the signal is multiplied by the edge's weight resulting in a possibly changed signal.

The neurons are the second point of interaction. Figure 4.1 contains four different kinds of neurons which are present in our tool's ANN. These are input, hidden, bias, and softmax output neurons. We further describe them below.

Except for input and bias neurons, each neuron receives multiple incoming real-valued signals from the preceding layer's neurons. Except for output neurons, each neuron outputs a single real valued signal, which is called the neuron's *activation*. The activation flows to all neurons of the succeeding layer (except for bias neurons).

- As mentioned above, each **input neuron** holds one component of the MLP's overall input. To be more precise, the k-th input neuron holds the k-th component x_k of the MLP's overall input $\vec{x}$ and outputs it to all outgoing edges.

- A **hidden neuron** performs calculations to determine its activation based on its input. Figure 4.2 shows a hidden neuron n. The neuron n's incoming signals are denoted as $\vec{x^n}$ and weighted element-wise with $\vec{w^n}$. As a result $x_i^n \cdot w_i^n \in \mathbb{R}$ is the real-valued signal coming in on the i-th edge. M^n denotes the count of n's incoming edges, which equals the lengths of $\vec{x^n}$ and $\vec{w^n}$. The incoming signals are processed in three steps.

  Firstly, the neuron n's weighted input vector $\vec{x^n} \star \vec{w^n}$ is fed to its *input function* $\sigma^n : \mathbb{R}^{M^n} \to \mathbb{R}$. $\star$ is the Hadamard product, which performs element-wise multiplication. σ^n aggregates the incoming weighted signals into one real-valued signal. Usually, the sum function is utilized for this aggregation, i.e., $\sigma^n\left(\vec{x^n} \star \vec{w^n}\right) = \sum_{i=1}^{M^n} x_i^n \cdot w_i^n$. For simplicity, we assume this from here unless otherwise noted.

  Secondly, after aggregating the incoming signals, the single aggregate value is passed to the neuron's *activation function* $f^n : \mathbb{R} \to \mathbb{R}$. Non-linear functions, as for example the sigmoid function[2] plotted in figure 4.3b, are a usual choice to allow the MLP to approximate non-linear target functions. f^n's output is the neuron's overall activation y^n.

  Thirdly, y^n is sent to all N^n outgoing edges.

  To sum up, a hidden neuron n can be written as $y_j^n = y^n = f^n\left(\sigma^n\left(\vec{x^n} \star \vec{w^n}\right)\right)$.

  This paragraph briefly motivates why sigmoid is used as activation function. Actually, the Heaviside step function as plotted in figure 4.3a is the desired activation function as its behavior corresponds to the one in the human brain [Ros61; Blo62]. In the human brain, a natural neuron *fires* if its accumulated input exceeds a certain threshold as with the Heaviside function. However, for the training algorithm to work, the neuron's complete function must be differentiable. Thus, the differentiable sigmoid function, which approximately looks like a smooth Heaviside function, is used.

- **Bias neurons** are similar to input neurons. Instead of holding and outputting a component of the MLP's input, they hold the value one and output it to all outgoing edges.

  They exist because it is often desirable to shift the activation functions along the x-axis. Bias neurons allow such shifts, since each outgoing edge 's weight of a bias is added to the argument of a neuron's activation function. As a result, the mapping between specific incoming weighted signals and the neuron's activation is not fixed. Instead, the training algorithm can control the shift during the training process by adjusting the bias' outgoing edges' weights.

[2]In this thesis, “sigmoid function” refers to the sigmoidal logistic function $f(x) = \frac{1}{1+\exp(-x)}$.

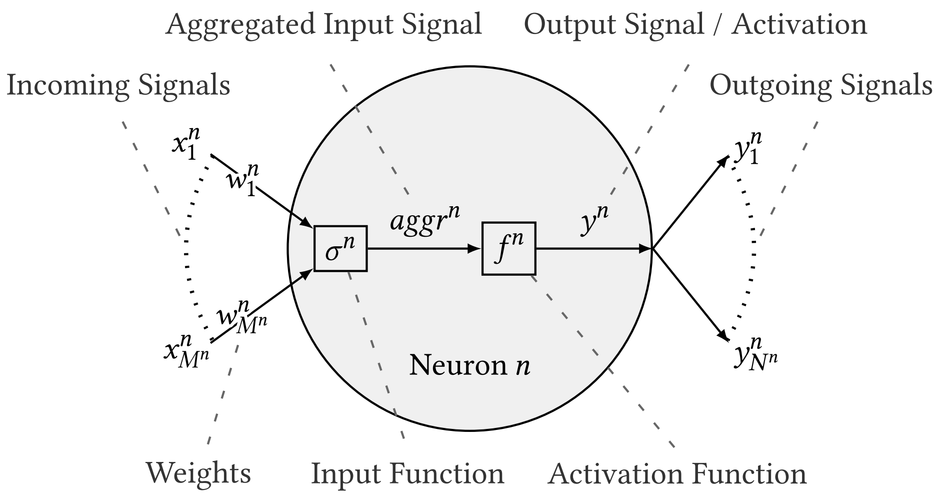
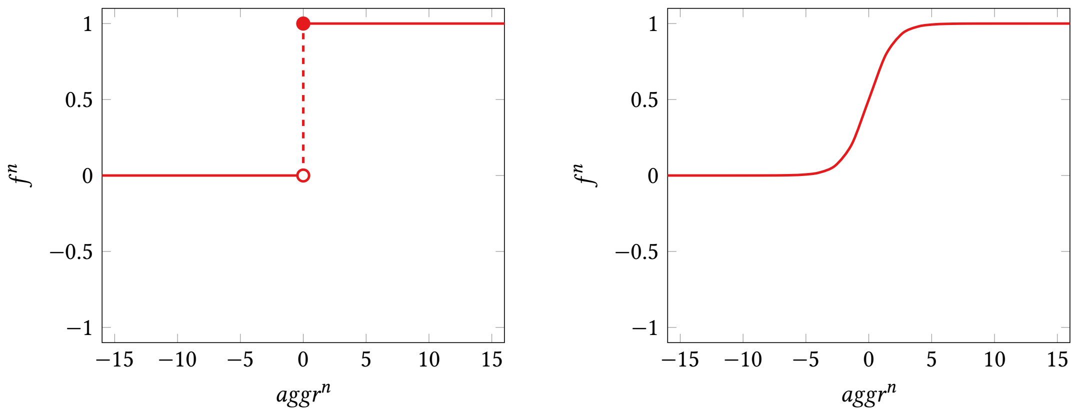


Figure 4.2: Structure of a basic hidden neuron n.



(a) Heaviside step function.

(b) Sigmoid function.

Figure 4.3: Heaviside and sigmoid functions.

- **Softmax output neurons** are the common output neuron in case of multi-class classification, as in this thesis.

  In multi-class classification with K classes C_k $\forall k \in 1, ..., K$, each target output $\vec{y}$ is a *one-hot encoded* vector, which represents exactly one class. For representation of class C_k the one-hot encoded vector's components are all zero except for the k-th component whose value is one. As a consequence, the FNN has exactly K output neurons, i.e., $N = K$. As an example, $(0, 1, 0, 0)$ would be the one-hot encoded output representation for class C_2 with $K = N = 4$.

  In such situation, Cross Entropy (CE) is usually employed as cost function as further described in section 4.2. To use CE, the FNN's output must be interpretable as a discrete probability distribution such that each output neuron's activation is in the interval $[0, 1]$ and all activations sum up to one. A last layer that has one softmax neuron for each preceding layer's neuron ensures these requirements. As a result, both the target one-hot encoded output and the predicted, almost one-hot encoded softmax output are discrete probability distribution and can be input to CE.

A softmax layer's incoming weights are restricted to 1 and each softmax neuron uses the special softmax function. Let n_k be the k-th softmax neuron. Then, n_k can be written as the softmax function

$$y^{n_k} = \frac{\exp\left(x_k^{n_k}\right)}{\sum_{k'=1}^{K} \exp\left(x_{k'}^{n_k}\right)}. \tag{4.1}$$

Each softmax neuron normalizes the activation of its corresponding neuron of the preceding layer to the interval $]0, 1[$ depending on how large that activation is compared to the preceding layer's other activations. As the softmax function is monotonic, it does not change the output values' order. Note that a softmax layer cannot generate a perfectly one-hot encoded output as the nominator of the softmax function in equation (4.1) is always greater than zero.

Since the softmax layer is the FNN's last layer, the k-th softmax neuron's output equals the k-th component of the FNN's overall output $\vec{y}$, i.e., $y^{n_k} = y_k$. Thus, for a given input $\vec{x}$, y^{n_k} can be interpreted as the probability of $\vec{x}$ belonging to the k-th class C_k according to $\hat{F}_{\underline{W}}$. For example, if $(3.4, 5.2, 0.1, -3)$ is the output of the layer preceding the softmax layer, the softmax layer's output will be $(0.14, 0.85, 0.00, 0.00)$ resulting in class C_2 being the predicted output.

In summary, the hidden neurons are the central part of an MLP and perform most of the calculations. Bias neurons help the ANN to be flexible. Softmax neurons convert their preceding activations to a discrete probability distribution.

Summarizing the whole MLP, it can be written as a function $\hat{F}_{\underline{W}} : \mathbb{R}^M \to \mathbb{R}^N, \hat{F}_{\underline{W}}(\vec{x}) = \vec{y}$, of its input $\vec{x}$, output $\vec{y}$, and weights $\underline{W}$. This notation is also valid in general for FNNs.

4.2 Training of FNNs

Roughly, the training of an ANN has the goal to adjust the weights of the ANN's edges with the result that $\hat{F}$ treats the example inputs similarly or identically to F. Unfortunately, analytically determining the best combination of weights is not possible in general. Therefore, other methods such as numerical optimization are used.

Not overfitting on the example inputs, i.e., not memorizing each input's output, is the decisive factor here. Hence, generalized rules for predicting the outputs based on the inputs' properties should be learned. If this generalization took place, the ANN would be able to predict the correct output (according to *F*) even for unknown inputs. To assess this, some input-output pairs are held back during training and then used for testing the ANN's performance on these unseen pairs. Since the ANN can make different kinds of errors during this, multiple metrics for quantifying the performance exist. We describe three such metrics in section 4.2.6.

The following paragraphs roughly describe how the training achieves its just mentioned goal. The subsequent sections describe some aspects of the training process in more detail.

As visualized in figure 2.1, the training process receives a *training set* and *validation set* and outputs a trained ANN model. The former are disjoint sets of pairs of *F*'s inputs and associated outputs. We assume them to be randomly drawn from the distribution specified by *F*. The latter is an FNN whose weights allow it to correctly predict outputs for the training and validation sets' inputs in most cases.

The training process consists of multiple steps and starts with our ANN having initial weights. Section 4.2.3 introduces the method that we utilize for initializing the weights. For each training pair, i.e., each pair of *F*'s input and associated output in the training set, the *training algorithm* executes the following steps.

1. It places the input in the ANN's input layer and performs a *forward pass*, i.e., it layer-wisely calculates all neurons' activations starting from the input layer. It retrieves the output activations from the ANN.

2. It uses the *loss function*, which we describe in detail in section 4.2.1, to quantify the error of the ANN's output being different from *F*'s target output. This is done by inserting the current weights, the actual output, and the target output into the loss function.

3. It calculates the derivative of the loss function with respect to each weight. As a result it obtains the influence of each weight on the error. Section 4.2.2 shows how those derivations can be deduced from the loss function.

4. It decreases each weight by its derivative with the goal that the error will be lower with the adjusted weights. When considering all weights together, the vector of all weights' derivatives is the gradient of the loss function. Adjusting each weight therefore corresponds to updating the weights in the direction of the negative gradient. Each update is called a training *iteration*.

After employing each training pair for adjusting the weights, one *epoch* passed. Updating the weights after each training pair corresponds to Stochastic Gradient Descent (SGD). Other variants aggregate the error over multiple training pairs before updating the weights. The number of training pairs in such a batch is called *batch size*. Section 4.2.2 describes SGD and other variants in more detail.

A training usually consists of multiple epochs. The validation set is incorporated to select the epoch whose model generalized best. After each epoch the training algorithm measures the model's performance on the training set and on the validation set. After the training, the training algorithm returns the model, which performed best on the validation set.

To exemplify, figure 4.4 shows curves which may occur when plotting training and validation performances over the epochs. In the figure, we used the F_1 metric, whose scores are between 0 and 1 with higher values corresponding to better performance. We discuss this and other scores in section 4.2.6. With the above steps the training algorithm focuses on reducing the loss function based on the training data. In the beginning, this result in an increasing training and validation score since the model starts to fit the training set by building generalized rules over the patterns in the input. Up to a certain point, these rules also allow predicting the validation pairs, i.e., the pairs of F's inputs and associated outputs in the validation set.

After some epoch – epoch 32 in the figure – the training score continues increasing while the validation score starts decreasing. The model continues to fit on the training data by building more specialized rules. In other words, it starts to memorize the training pairs by scarifying generalization. As a result these less general rules do not apply for the validation pairs anymore, which results in a worse validation score. This phenomenon is called *overfitting*. We further define a ratio for measuring it and ways of tackling it in section 4.2.5.

The validation score allows assessing the model's grade of training generalization. However, it is not an unbiased estimation of the model's generalization with respect to entirely unrelated data. This is the reason for also holding an additional *test set* back from the training. The model that the training algorithm selected based on the validation set is then evaluated on the test set. If the score achieved on the test set is worse than the one reached on the validation set, the training algorithm overfitted on the validation set.

For example, in case of noisy training and validation scores there may be multiple weight combinations having the same training score but different validation score. By selecting the model with the best validation score, the training algorithm optimizes in terms of validation score, and is thus able to overfit on that score.

In summary, the training algorithm iteratively adjusts the model's weights based on the training set and utilizes the validation set to select the model that generalized

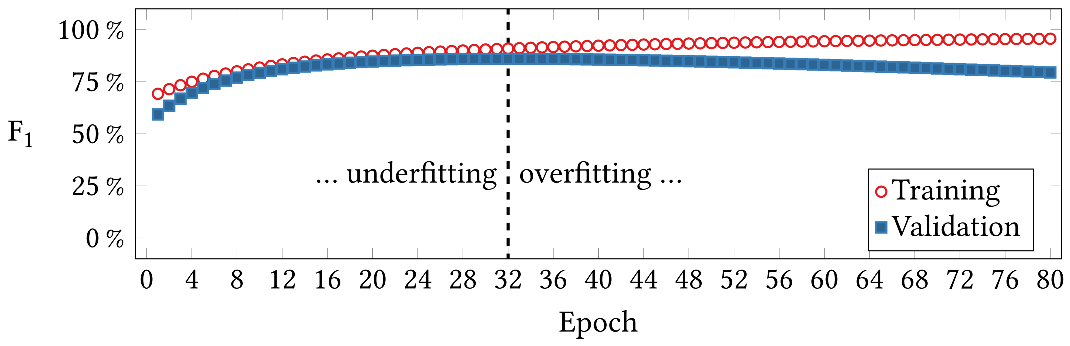


Figure 4.4: Example curves showing training and validation scores over training epochs. The dashed, vertical line marks the epoch with the best validation F_1.

best. The selected model's score on the test set estimates the performance on unseen data. Overfitting is possible both on the training set and on the validation set.

Since an ANN is not only defined by its weights, section 4.2.4 presents techniques to optimize the ANN's HP such as the training algorithm, layer counts, and the neurons' activation functions.

4.2.1 Loss Function

This section derives the *Cross Entropy* (CE) loss, which is usually used for multi-class classification. We also utilize it for our tool.

The *loss* is a measurement of the approximation quality of an FNN represented by $\hat{F}_{\underline{W}}$ with respect to a given set T_F of input-output pairs. It is defined as

$$E\left(\hat{F}_{\underline{W}}, \mathrm{T}_F\right) = \sum_{\left(\vec{x},\vec{y}\right)\in\mathrm{T}_F} \epsilon\left(\hat{F}_{\underline{W}}, \vec{x}, \vec{y}\right), \tag{4.2}$$

which is the sum of all single costs ϵ of each $\hat{F}_{\underline{W}}\left(\vec{x}\right)$ being the predicted output instead of the pair's target output $\vec{y}\left(= F\left(\vec{x}\right)\right)$. Based on equation (4.2), $E\left(\hat{F}_{\underline{W}}, \mathrm{T}_F^{train}\right)$ is the *training loss*.

The cost function ϵ will be greater than zero iff $\hat{F}_{\underline{W}}\left(\vec{x}\right) \neq \vec{y}$ and will be equal to zero otherwise. CE is the cost function usually used for multi-class classification to score the difference between the target, one-hot encoded output and the predicted, almost one-hot encoded softmax output. CE was established based on the maximum likelihood principle and can be computed as

$$\epsilon_{CE}\left(\hat{F_{\underline{W}}}, \vec{\ddot{x}}, \vec{\ddot{y}}\right) = \sum_{k=1}^{K} \ddot{y}_k \cdot \ln\left(\hat{F_{\underline{W}}}\left(\vec{\ddot{x}}\right)_k\right).$$

$\hat{F_{\underline{W}}}\left(\vec{\ddot{x}}\right)_k$ is the k-th component of the FNN's output for input $\vec{\ddot{x}}$. In conclusion,

$$E_{CE}\left(\hat{F_{\underline{W}}}, \mathrm{T}_F\right) = \sum_{\left(\vec{\ddot{x}},\vec{\ddot{y}}\right)\in \mathrm{T}_F} \sum_{k=1}^{K} \ddot{y}_k \cdot \ln\left(\hat{F_{\underline{W}}}\left(\vec{\ddot{x}}\right)_k\right)$$

summarizes our overall loss function E_{CE}.

4.2.2 Gradient Descent

This section describes the process of updating weights to descend the loss function in the negative direction of the gradient. It also visualizes the influence of the learning rate, and it shows the difference between stochastic and mini-batch gradient descent.

Since the training pairs T_F^{train} are constant, E_{CE} is a function with a variable for each weight w_{ij} in $\underline{W}$, i.e., $E_{CE} : \mathbb{R}^{\Psi} \rightarrow \mathbb{R}$ with weight count Ψ. We search for the weights $\underline{W'} = \left(w'_{ij}\right)$ that minimize the training loss, i.e.,

$$\underline{W'} = \arg\min_{\underline{W}} E_{CE}\left(\hat{F_{\underline{W}}}, \mathrm{T}_F^{train}\right). \tag{4.3}$$

Equation (4.3) can be rephrased into the necessary condition

$$\vec{\nabla}_{w_{ij}} E_{CE} = \vec{0} \tag{4.4}$$

where $\vec{\nabla}_{w_{ij}}$ is the gradient with respect to each weight w_{ij}.

$\hat{F_{\underline{W}}}$ is a differentiable composition of sums, multiplications, (non-)linear input and differentiable activation functions. Thus, E_{CE} is also differentiable, such that the left side of equation (4.4) can be calculated analytically. However, analytically solving the system of non-linear equations in equation (4.4) itself is not possible in general.

To overcome this limitation, equation (4.4) is solved by employing the *Gradient Descent* (GD) algorithm. This algorithm repeatedly updates the weights $\underline{W}$ by a fixed step in the direction of $E_{CE}\left(\hat{F_{\underline{W}}}, \mathrm{T}_F^{train}\right)$'s negative gradient. This hopefully converges to the global minimum or at least to a "good" local minimum. As mentioned above, each of these weight updates is named *iteration*.

Each iteration t's fixed step is defined as

$$\underline{\Delta W^t} = -\alpha * \left(\frac{\partial E_{CE}\left(\hat{F_{\underline{W}}}, \mathrm{T}_F^{train}\right)}{\partial w_{ij}} \right) \left(\underline{W^t}\right) \tag{4.5}$$

where α is the *learning rate*, $\underline{W}$ the matrix of weight variables, $\underline{W^t}$ the matrix of concrete weight values at iteration t, and (...) $\left(\underline{W^t}\right)$ the training loss derivative's value at the point $\underline{W^t}$.

Figure 4.5a exemplifies the one-dimensional case where only one weight is present. Each solid arrow denotes one iteration's step.

With equation (4.5),

$$\underline{W^{t+1}} = \underline{W^t} + \underline{\Delta W^t}$$

is the complete weight update formula used in each iteration.

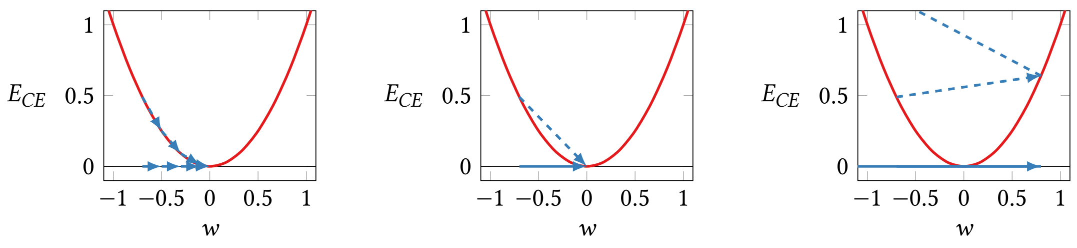
The learning rate α's concrete value is essential for the training process' convergence and efficiency [Roj96]. In case of a too small learning rate, the training process will be slow and chances of getting stuck in a local minimum that is much greater than E_{CE}'s global minimum, are high. Figure 4.5a exemplifies this. Vice versa, if the learning rate is chosen too big, the probability of the training process oscillating and/or diverging will increase. Figure 4.5c visualizes this.

Stochastic Gradient Descent (SGD) is a variation of GD where not the whole training set but only one randomly drawn (without replacing) input-output pair is used per iteration. As mentioned above, after each training pair has been used for weight update, i.e., after $\left|\mathrm{T}_F^{train}\right|$ iterations, one *epoch* is done. LeCun et al. describes numerous advantages of SGD compared to GD [LeC+12].

Compared to GD, more iterations can be done in a given time span usually resulting in an earlier convergence. This would be true especially if T_F^{train} contains similar training pairs.

Another advantage arises from the fact that the gradients computed with SGD are noisy and do not move down $E_{CE}\left(\hat{F_{\underline{W}}}, \mathrm{T}_F^{train}\right)$'s gradient exactly. The resulting noisy weight updates allow the SGD algorithm to escape from non-global minima since the minima of $\underline{\Delta W^t}$ will be different for each iteration. However, this also leads to problems when the SGD algorithm descends to E_{CE}'s global minimum since it continues fluctuating around the global minimum instead of converging. Decreasing the learning rate over time is a countermeasure. Another countermeasure is the combination of GD and SGD through the use of *mini-batches* per iteration.

Mini-batch Gradient Descent (MGD) is a mixture of GD with SGD [GBC16, pp. 274-277]. Instead of drawing one training pair per iteration, multiple training pairs are



(a) Too small learning rate. (b) Optimal learning rate. (c) Too large learning rate.

Figure 4.5: One-dimensional example of gradient descent for different learning rates. Solid arrow denote iteration steps. Corresponding dashed arrows visualize the gradient descent. Figure based on LeCun et al. [LeC+12].

drawn without replacing. *Batch size* is the number of pairs in each of these resulting mini-batches (except for one mini-batch iff $\left|T_F^{train}\right|$ is not a multiple of the batch size). The batch size is most commonly chosen $\gg 1$ depending on the dataset and also limited by hardware constraints as discussed in section 7.1.1. Additionally, to further increase variability in the training process, T_F^{train} is shuffled after each epoch.

4.2.3 Weight Initilization

Selecting appropiate initial weight values $\underline{W^0}$ is another important subject in terms of a good training process. For example, starting with weight values close to the weight values of E_{CE}'s global minimum is advantageous in general. Usually, $\underline{W^0}$ is drawn from some probability distribution. In this thesis, weights of edges originating from a bias neuron are set to zero. Each other weight w_{ij} is drawn from the uniform distribution in the interval $\left[-\frac{1}{\sqrt{\mu}}, \frac{1}{\sqrt{\mu}}\right]$, i.e., $w_{ij} \sim \mathcal{U}\left(-\frac{1}{\sqrt{\mu}}, \frac{1}{\sqrt{\mu}}\right)$, where μ is the number of neurons in the preceding layer. This heuristic is named *Glorot Initialization* since it has been proposed by Glorot and Bengio, and it has proven itself in practice to speed up the training algorithm's convergence [GB10].

4.2.4 HP Optimization

The above sections describe the training process, which determines the best specific weights of the ANN. However, an ANN is not only described by its weights but also by its HPs such as layer counts, neuron counts, activation function, training algorithm, etc. This section briefly introduces two approaches on the basis of which appropriate HP values can be chosen.

Grid search is a technique that exhaustively explores all combinations of given HPs' values. Based on the resulting list of combinations, tool configurations are created and trained. The advantage of grid search is the fact that for each tool configuration and each HP another tool configuration is trained which only differs in that HP. This is convenient if comparisons of tool configurations are desired. On the downside, performing experiments for each combination of HP values takes time exponential in the number of HP values. Since the training processes of the tool configurations are independent of each other, parallelization allows to mitigate this problem.

Mathematically, let $\vec{\pi} = (\pi_1, ..., \pi_\Phi)$ be the tuple of the Φ HP variables π_ϕ, and let $\rho(\pi_\phi)$ be the finite set of the HP variable π_ϕ's possible values. Then

$$\Pi = \{(\overline{\pi_1}, ..., \overline{\pi_\Phi}) \,|\, \overline{\pi_1} \in \rho(\pi_1), ..., \overline{\pi_\Phi} \in \rho(\pi_\Phi)\}$$

is the cartesian product of all HP values $\rho(\pi_\phi)$. Π can be imagined as a Φ-dimensional grid spanned by the HP values, which is giving grid search its name. Figure 4.6 exemplifies the resulting grid given the two HPs *Learning Rate* (π_1) and *Hidden Neuron Count* (π_2) with their possible values assumed as $\rho(\pi_1) = \{0.1, 0.001\}$ and $\rho(\pi_2) = \{2, 16, 8, 32\}$, respectively.

Determining a baseline combination first is an alternative. Starting from such a baseline, the HPs are separately modified. This takes time linear in the number of HP values. One drawback is that the determination of a baseline usually requires knowledge about the HPs relationship. Another drawback is that this technique makes assumptions about the HPs independence. The first drawback can be mitigated

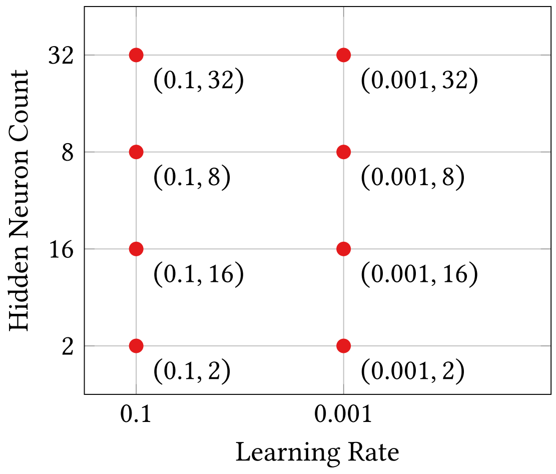


Figure 4.6: Exemplary combinations of HP values Π (red dots) resulting from grid search for the two HPs *Learning Rate* (π_1) and *Hidden Neuron Count* (π_2) with their possible values assumed as $\rho(\pi_1) = \{0.1, 0.001\}$ and $\rho(\pi_2) = \{2, 16, 8, 32\}$, respectively.

by combining both approaches. Firstly, grid search is utilized on a coarse grid to obtain a baseline. Secondly, from that baseline HPs can be modified in smaller steps.

For the above example, grid search could have identified $(0.1, 16)$ as baseline. Then, learning rates of 0.075 and 0.125 could be utilized while keeping 16 hidden neurons. Afterwards, tool configurations with 12 and 24 hidden neurons could be evaluated together with a learning rate of 0.1.

4.2.5 Overfitting and Regularization

This section defines a straightforward ratio for measuring overfitting. Besides, we introduce regularization strategies, whose goal is the reduction or prevention of overfitting.

Overfitting Ratio We did not find a simple measure for overfitting on the validation set in the literature. We therefore define the *overfitting ratio* as the ratio between the model's validation score and its test score. The ratio's ideal value is 1.0. This corresponds to the situation in which the learned rules of the model that the training algorithm selected based on the validation set equally apply to the test set. A value greater than 1.0 indicates overfitting. If the model randomly predicts the correct output for training set inputs, values smaller than 1.0 may occur.

Regularization The goal of regularization is the reduction and prevention of overfitting [Bis06]. Keeping the model's complexity low by decreasing layer and neuron counts is the simplest regularization strategy. Due to this, the model is forced to learn simpler, generalized rules instead of complex rules exactly fitting each training pair. However, too few layers or neurons result in underfitting. Grid search, which we described in section 4.2.4, is one approach to identify well-fitting model HPs.

Early stopping is another regularization strategy which we employ in this thesis. It stops the training as soon as the validation score starts decreasing [Bis06]. For the above example in figure 4.4 this corresponds to stopping after epoch 33 and returning the model achieved after epoch 32. However, usually the training and validation score curves are noisy. In this case, after observing a decreasing validation score, the training is continued for a given count of epochs. If the validation score does not increase again during those epochs, early stopping terminates the training and returns the model with the best validation score. Otherwise, early stopping is continued with the new best validation score.

4.2.6 Metrics

This section describes metrics for assessing an ANN model's prediction power on a given set of input-output pairs. There are four possible outcomes when predicting an input's output. The *confusion matrix* in table 4.1 shows these in the context of binary classification of vulnerabilities, where vulnerable is considered the positive class.

Hereafter, TP, TN, FP, and FN denote the counts of the respective predictions of the model on the given set of SCS. All metrics introduced in this section produce scores in the interval $[0, 1]$.

Accuracy The *accuracy* metric counts the number of correctly predicted inputs and divides the result by the count of all given inputs. Mathematically, it is defined as

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}.$$

Its disadvantage is its lack of expressiveness in case of unbalanced classes, i.e., if the number of vulnerable SCS vastly differs from the number of non-vulnerable SCS. For example, given one vulnerable and nine non-vulnerable SCS, the accuracy of a model that guesses non-vulnerable for each SCS would be 90 %. While 90 % indicates a high prediction power, the model did not learn generalized rules based on the SCS.

F_1 The F_1 metric is the harmonic mean of *precision* and *recall* which are defined as

$$precsion = \frac{TP}{TP + FP} \quad \text{and} \quad recall = \frac{TP}{TP + FN},$$

respectively. A low FP and a low FN count corresponds to high precision and recall, respectively. Thus, the F_1 metric

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

has the highest possible value of 100 % in the ideal case of no FPs and FNs.

AuROC The *Area under Receiver Operating Characteristic curve* (AuROC) is another metric often used in the literature for binary classification [Met78]. The above-mentioned metrics measure the performance of a model with a specific threshold. The threshold governs, which strength of activation an output neuron must produce

Table 4.1: Elements of the confusion matrix in the context of vulnerability detection.

		Predicted	
		Vulnerable	**Non-vulnerable**
Actual	**Vulnerable**	*True Positive* (TP)	*False Negative* (FN)
	Non-vulnerable	*False Positive* (FP)	*True Negative* (TN)

such that its corresponding class is predicted. In constrast, AuROC varies the model's threshold and aggregates the performances for all thresholds by computing the area under the receiver operating characteristic curve that is formed by plotting true positive rates against false positive rates for various thresholds. Further elaboration and visualization of these are outside this thesis' scope and space constraints. See the paper by Metz for details [Met78].

As a consequence, AuROC measures how versatile a model is. For example, the AuROC score of a model achieving moderate performance with many threshold may be higher than the AuROC score of a model achieving superb performance for a single threshold and poor performance for the others. However, to compare a model's performance to others or to use it productively, one specific threshold must be chosen. As a result, AuROC often overestimates the actual performance of a model. Lobo, Jiménez-Valverde, and Real describe this issue in more detail [LJR08]. For the above reasons, we refrain from using the AuROC metric in this thesis.

4.3 Neural Arithmetic Logical Unit

An important ability necessary for detecting vulnerabilities in source code is basic math. For example, if a C++ array is declared with length eight and its ninth element is later accessed inside a loop, an overflow occurs. In general there may be a vulnerability if the difference between the array's declaration length and the greatest index used for accessing the array is greater than or equal to zero.

As can be seen from this example, ANNs need to be able to perform a subtraction and a comparison to detect this specific vulnerability. However, according to Trask et al., ANNs are usually not able to generalize basic arithmetic operations well [Tra+18]. In particular, extrapolation of the operations to the range outside the values encountered during training works poorly. As a countermeasure Trask et al. developed a new kind of neurons called Neural Arithmetic Logic Unit (NALU).

Addition and subtraction are also possible without NALUs. Figure 4.7 shows an MLP performing the subtraction of the MLP's third input component from the first input

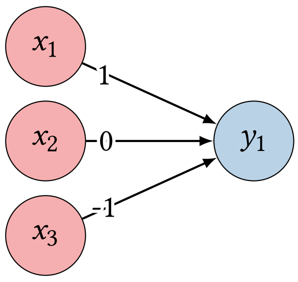


Figure 4.7: MLP realizing subtraction of two of its inputs, i.e., $\hat{F_{\underline{W}}}(\vec{x}) = 1 \cdot x_1 + 0 \cdot x_2 + -1 \cdot x_3 = x_1 - x_3$

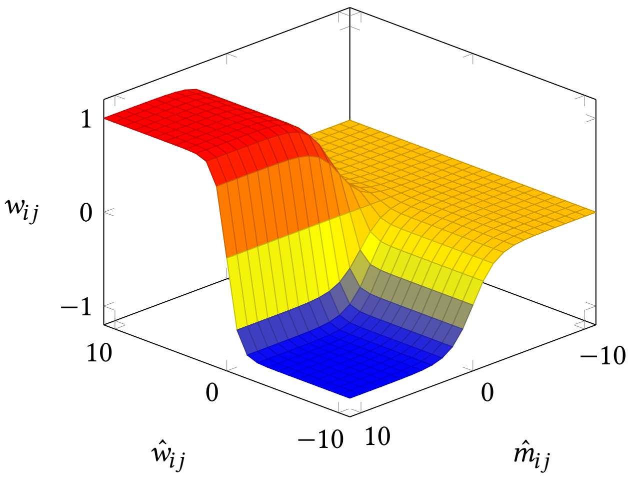


Figure 4.8: $w_{ij} = \tanh\left(\hat{w}_{ij}\right) \cdot \operatorname{sigmoid}\left(\hat{m}_{ij}\right)$ with stationary values -1, 0, and 1.

component: $\hat{F_{\underline{W}}}(\vec{x}) = 1 \cdot x_1 + 0 \cdot x_2 + -1 \cdot x_3 = x_1 - x_3$. The weight 0 corresponds to ignoring the input component as for the MLP's second input component in the example. However, since the weights are adjusted in a continuous way during training it is difficult to restrict them to only take integer values and especially to restrict them to only take the values -1, 0, or 1.

This is where the NALU technique begins. Instead of restricting the weights $\underline{W}$ during training, Trask et al. replace them by a special function of two other weight matrices $\underline{\hat{W}}$ and $\underline{\hat{M}}$, which are trained in the usual way. This special function is shown in figure 4.8 and applied element-wise to $\underline{\hat{W}}$ and $\underline{\hat{M}}$ as

$$\underline{W} = \tanh\left(\underline{\hat{W}}\right) * \operatorname{sigmoid}\left(\underline{\hat{M}}\right).$$

As a result, $\underline{W}$'s single weights are close to -1, 0, or 1 for most values of $\underline{\hat{W}}$ and $\underline{\hat{M}}$. Based on this, they create a layer of neurons capable of adding and subtracting its input's component in different combinations as

$$\vec{a^n} = \underline{W^n} \cdot \vec{x^n}. \tag{4.6}$$

They name such a layer *Neural Accumulator* (NAC) and the layer's output $\vec{a^n}$ consists of the results of adding/subtracting $\vec{x^n}$'s components as specified by $\underline{W^n}$. The number of additions/subtractions performed equals the length of $\vec{a^n}$.

To exemplify, imagine a single NAC having two inputs and three outputs. Thus, it performs three calculations at the same time. Then, $\underline{W^n}, \underline{\hat{W^n}}, \underline{\hat{M^n}}$ are $\in \mathbb{R}^{3\times2}$. Each component of the NAC's output, i.e., each component a^n_i of $\vec{a^n}$, holds the results of adding/subtracting x^n_1 and x^n_2 in some way. $\underline{W^n}$ decides how this is done through $a^n_i = w_{i1} \cdot x^n_1 + w_{i2} \cdot x^n_2$ where w_{ij} – as said before – are mostly either -1, 0, or 1.

In order to multiply and divide input components with each other, a detour via the logarithmic space is taken. Roughly, three steps are carried out for this. Firstly, the input $\vec{x^n}$ is transformed into logarithmic space. Afterwards, the usual NAC addition/subtraction operation is performed. Lastly, the result is transformed back to linear space. These steps correspond to multiplying/dividing the input components with each other because of

$$\ln(x) + \ln(y) = \ln(x \cdot y) \Rightarrow x \cdot y = \exp(\ln(x) + \ln(y)).$$

Whether a multiplication or division is performed between input components or whether an input component is ignored, is – as before – controlled by $\underline{W}$. In greater detail, positive values must be ensured for $\vec{x^n}$ before applying ln. This is done by using absolute values and adding some small values $\vec{\epsilon}$ to avoid $\ln\left(\vec{0}\right)$. As a consequence for the NALU, a negative value cannot be multiplied/divided by a positive value or vice versa. Summarized, multiplication/division is conducted according to

$$\vec{m^n} = \exp\left(\underline{W} \cdot \ln\left(\left|\vec{x^n}\right| + \vec{\epsilon}\right)\right) \tag{4.7}$$

with $|\cdot|$ denoting the element-wise absolute value. $\vec{m^n}$ consists of the results of multiplying/dividing $\vec{x^n}$'s components as specified by $\underline{W}$.

Having NACs for addition/subtraction and the just mentioned trick for multiplication/division, NALUs can be constructed by combining both. For allowing NALUs to learn which of the math operations to perform, another trainable weight matrix $\underline{G}$ is introduced. Then,

$$\vec{g^n} = \text{sigmoid}\left(\underline{G} \cdot \vec{x^n}\right) \tag{4.8}$$

holds the NALU's gate activations $g^n_i \in \,]0, 1[$ governing which kind of math operations should be performed. Summarized,

$$\vec{y^n} = \vec{g^n} * \vec{a^n} + \left(\vec{1} - \vec{g^n}\right) * \vec{m^n}$$

describes the overall NALU based on equations (4.6) to (4.8). Figure 4.9 visualizes the structure and data flow of a NALU. Different colors mark the components realizing addition/subtraction, multiplication/division, and the gating mechanism.

For a single output y^n_j, a gate activation of one corresponds to the NALU behaving like a NAC. Vice versa, a gate activation of zero indicates the NALU performing multiplication/division. Gate activations in between result in a mixture of both.

In cases where only NAC operations are desired, an FNN with NACs outperforms one with NALUs regarding training time as well as extrapolation [Tra+18]. This

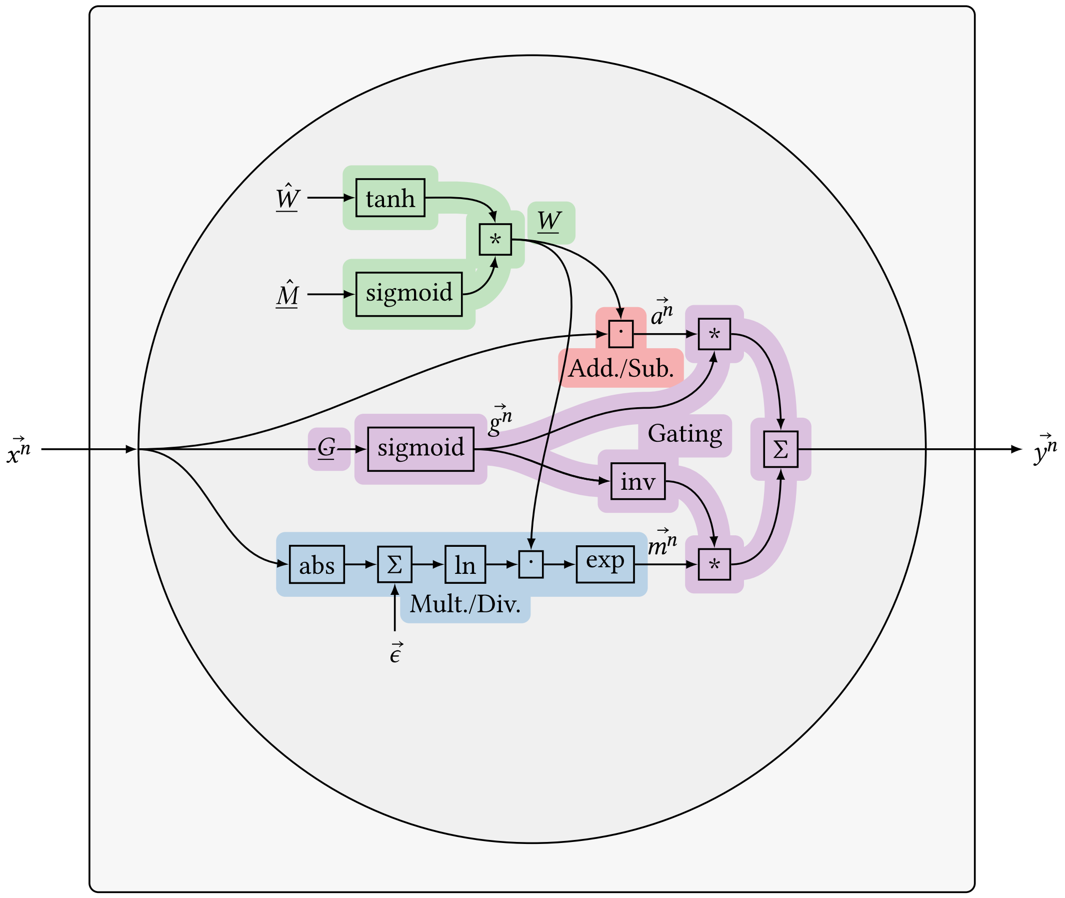


Figure 4.9: Structure of a NALU layer with a single NALU. Σ denotes the normal, element-wise addition. · is the normal matrix multiplication while ⋆ is the Hadamard product (element-wise multiplication). inv denotes element-wise $1 - \cdot$. sigmoid, tanh, ln, exp, and abs are applied element-wise. $\vec{x} \xrightarrow{\underline{W}}$ denotes the multiplication of $\underline{W}$ with the signal $\vec{x}$, i.e., $\underline{W} \cdot \vec{x}$. $\xrightarrow{\text{abc}}$ denotes a signal which is just labeled “abc”, i.e., no interaction with the signal occurs.

result is not surprising, as of the extra complexity of a NALU it is easier for a NAC to become high-performing than for a NALU to become a high-performing NAC.

4.4 RNNs and Long Short-Term Memory

In case of sequentially ordered data, FNNs lack possibilities to represent the information introduced by that ordering. An example for such an order is the spatial relationship between source code tokens as they are parsed from top to bottom and from left to right by the compiler. Another example are temporal relationships between temperature measurements for weather forecasting. FNNs cannot handle these relationships because of their strict feed-forward behavior, where each input is processed individually without knowledge of previous inputs.

This loss of knowledge can be circumvented by not only using the current input for computing each neurons output, but by also referring to previous inputs. Doing this leads to feedback loops inside the ANN as in figure 4.10. ANNs that have such a cyclic graph allowing information to also flow backwards, are called *Recurrent ANNs* (RNNs). These are described according to Graves unless otherwise noted [Gra08]. We adjusted their notation for consistency with this thesis' notation.

Due to the additional dimension introduced by the ordering information, the RNN's input $\underline{x}$ is not one-dimensional, but a sequence of one-dimensional sub-inputs. This can be represented as a two-dimensional matrix, i.e., $\underline{x} \in \mathbb{R}^{M \times U}$. U is the sequence length and M specifies the length of each sub-input. *Feature vector* is a term often used for such a sub-input on the basis of which $\underline{x}$ is also referred to as *feature sequence*. Section 5.2 elaborate both terms in more detail. The length M of the feature vectors is assumed to be constant, but the sequence length may vary between multiple sequences as visualized for two example inputs

$$\underline{x}^1 := \begin{bmatrix} 1 & 2 & 1 & 5 \\ 2 & 4 & 2 & 5 \\ -1 & 20 & -1 & 3 \end{bmatrix} \quad \hat{=} \left(\begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 \\ 4 \\ 20 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}, \begin{pmatrix} 5 \\ 5 \\ -3 \end{pmatrix} \right)$$

and

$$\underline{x}^2 := \begin{bmatrix} 2 & 1 \\ 2 & 4 \\ 4 & 5 \end{bmatrix} \quad \hat{=} \left(\begin{pmatrix} 2 \\ 2 \\ 4 \end{pmatrix}, \begin{pmatrix} 1 \\ 4 \\ 5 \end{pmatrix} \right).$$

$\underline{x}^1$'s sequence length $U^{\underline{x}^1} = 4$ differs from the second example sequence's length $U^{\underline{x}^2} = 2$ while $M = 3$ is identical for both.

In this thesis, classification of such a sequence is the desired task, i.e., predicting the weakness or non-weakness for a sequence of AST nodes. Therefore, the RNN's

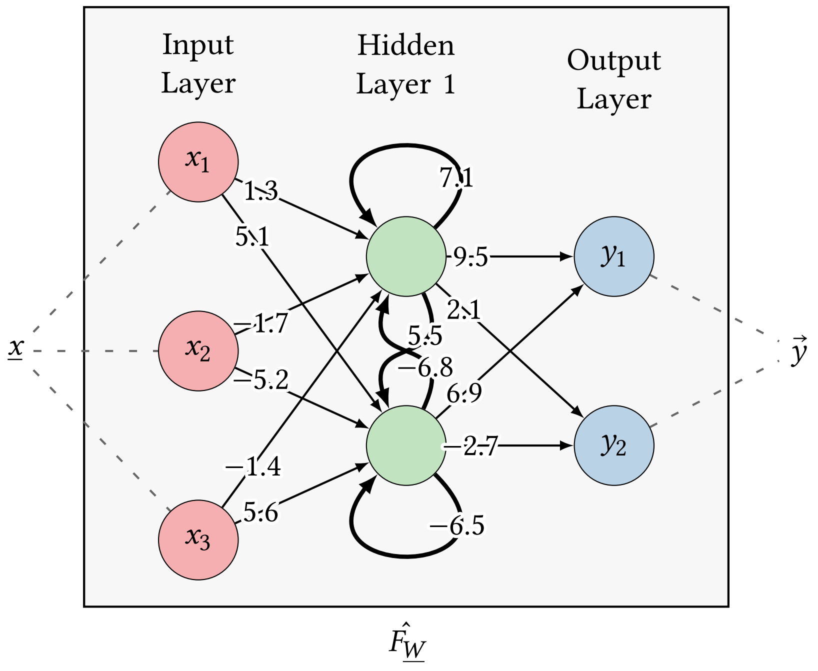


Figure 4.10: RNN with one recurrent hidden layer.

output is a one-dimensional one-hot-encoded vector of class-membership as before with FNNs.

The forward pass looks similar to the one in an FNN as described in section 4.1. The only difference is a hidden neuron's vector of incoming signals which additionally contains signals from the previous timestep. To be more precise, it contains the previous activations of all hidden neurons in its layer.

One approach to train an RNN is *Backpropagation Through Time* (BPTT), which unrolls the RNN along the temporal or spatial dimension for certain timesteps and then uses the common backpropagation approach [WZ95]. Unrolling the RNN duplicates weight variables, whose values may differ after backpropagation. When rolling the RNN back up, the weights' values are aggregated, e.g., by averaging them.

The unrolled RNN easily becomes deep with the result that gradients propagate over many layers. This leads to vanishing or even exploding gradients [GBC16, p. 396ff.]. These problems arise from the fact that the unrolled RNN's function consists of deeply nested activation functions. During the gradient calculation of each training pair's error, the chain rules converts the nested activation functions to many multiplicative terms. In case of the sigmoid activation function, its derivation maps to the interval $]0, 0.25]$. As a consequence, many small values are multiplied during gradient calculation. This results in so-called *vanishing gradients* since the multiplications' result is close to or equal to zero. In case of other activation functions, the opposite of exploding gradients is also possible. In both cases long-term relationships between sub-inputs cannot be successfully learned. This is especially disadvantageous in the context of this thesis since a program's semantically related parts often lie far apart in the source code.

One approach to overcome this problem is the *Long Short-Term Memory* (LSTM) technique invented by Hochreiter and Schmidhuber [HS97]. It addresses that problem by employing an internal *memory state* that flows to the next time step without being activated by an activation function. The authors name it *constant error carrousel.* As a result, the gradients for this memory state do not suffer from vanishing or exploding gradients, and an RNN consisting of *LSTM neurons* is able to capture long-distance relationships between an RNN's sub-inputs.

The original LSTM neuron by Hochreiter and Schmidhuber had only two gates, namely input gate and output gate [HS97]. More sophisticated LSTM neurons with forget gates according to Gers, Schmidhuber, and Cummins are most commonly used today and implemented in TensorFlow [GSC99]. We therefore use and describe these in this thesis.

An LSTM layer consists of multiple LSTM neurons which are fully-connected in a recurrent way as in figure 4.10. Let H be the number of neurons in the LSTM layer and let M be the count of incoming signals. Each LSTM neuron n_k outputs a signal $y^{n_k,t} \in \mathbb{R}$. Since all LSTM neurons perform the same calculations, they can be condensed into a single LSTM neuron outputting a vector of signals $\vec{y^{n_k,t}} \in \mathbb{R}^H$. As the condensed notation corresponds to the one in the literature and because an LSTM layer consisting of one neuron can be more easily visualized, it is used in this thesis. As a result, the weights $\underline{W}$ and $\underline{V}$ are matrices below.

Figure 4.11 visualizes the structure as well as the data flow in an LSTM layer with M inputs and H outputs. Three gates govern the information flow inside an LSTM neuron: a forget gate, an input gate, and an output gate. For each gate, a gate activation $\in]0,1[$ is constructed based on the neuron's current input signals $\vec{x^{n,t}}$, its bias, and the layer's outputs signals $\vec{y^{n,t-1}}$ from the previous timestep. Each signal is weighted with a different, trainable weight. As a result, an LSTM neuron can learn how much of previous memory to forget, how much of each part of the input to remember, and how much of the current memory to output. All of these "how much"s are determined based on the current input sequence and the neuron's weights.

Firstly, the *forget gate*'s activation $\vec{\phi^{n,t}}$ controls how much of the previous internal memory $\vec{c^{n,t-1}}$ should be forgotten. However, to be precise, it should actually be called *keep gate* because a higher activation has the effect of forgetting less. For consistency, we stick with *forget gate* as in the literature.

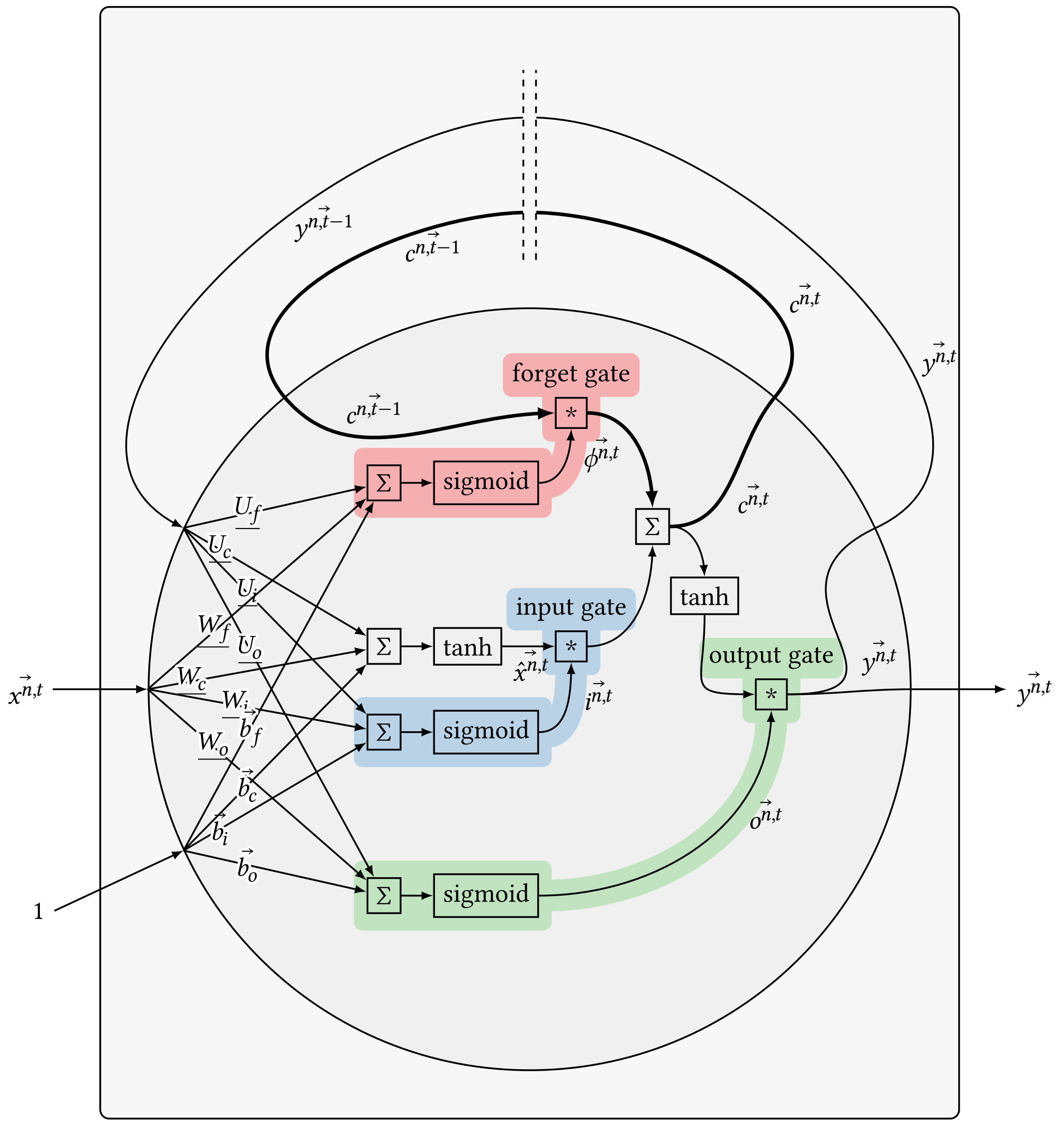


Figure 4.11: Structure of an LSTM layer with a single LSTM neuron. The constant error carrousel is marked with thicker arrows. Otherwise, the notation is identical to figure 4.9.

Afterwards, the *input gate*'s activation $\vec{i}^n_t$ regulates how much of the currently available input information $\hat{\vec{x}}^{n,t}$ should be added to the internal memory. $\hat{\vec{x}}^{n,t}$ is computed by weighting and biasing the current input and previous output, followed by tanh squashing the result to $]-1,1[$. Together, the forget and input gates' output form the new internal memory $\vec{c}^{n,t}$, which flows to the next timestep.

At the third gate, the output gate's activation $\vec{o^{n,t}}$ controls how much of the squashed, new internal memory $\tanh\left(\vec{c^{n,t}}\right)$ is used as the LSTM neurons output. As usual with RNNs, the neuron's output flows both back to itself in the next time step and forward to the succeeding layer.

The above-discussed behavior is reflected in equations (4.9) to (4.14). For each forget gate $\because$, its activation's input weights $\underline{W^n_{\because}}$ are $\in \mathbb{R}^{M\times H}$. Analogously, the activation's weights $\underline{V^n_{\because}}$ for the previous neuron outputs are $\in \mathbb{R}^{H\times H}$. The biases are $\in \mathbb{R}^{H}$.

Gate activations:

$$\vec{\phi^{n,t}} = \operatorname{sigmoid}\left(\underline{W^n_\phi} \cdot \vec{x^{n,t}} + \underline{V^n_\phi} + \cdot \vec{y^{n,t-1}} + \vec{b^n_\phi}\right) \in \mathbb{R}^H \tag{4.9}$$

$$\vec{i^{n,t}} = \operatorname{sigmoid}\left(\underline{W^n_i} \cdot \vec{x^{n,t}} + \underline{V^n_i} + \cdot \vec{y^{n,t-1}} + \vec{b^n_i}\right) \in \mathbb{R}^H \tag{4.10}$$

$$\vec{o^{n,t}} = \operatorname{sigmoid}\left(\underline{W^n_o} \cdot \vec{x^{n,t}} + \underline{V^n_o} + \cdot \vec{y^{n,t-1}} + \vec{b^n_o}\right) \in \mathbb{R}^H \tag{4.11}$$

Weighted, biased, and squashed neuron input:

$$\vec{\hat{x}^{n,t}} = \tanh\left(\underline{W^n_{\hat{x}}} \cdot \vec{x^{n,t}} + \underline{V^n_{\hat{x}}} + \cdot \vec{y^{n,t-1}} + \vec{b^n_{\hat{x}}}\right) \in \mathbb{R}^H \tag{4.12}$$

New memory state:

$$\vec{c^{n,t}} = \vec{c^{n,t-1}} * \vec{\phi^{n,t}} + \vec{\hat{x}^{n,t}} * \vec{i^{n,t}} \in \mathbb{R}^H \tag{4.13}$$

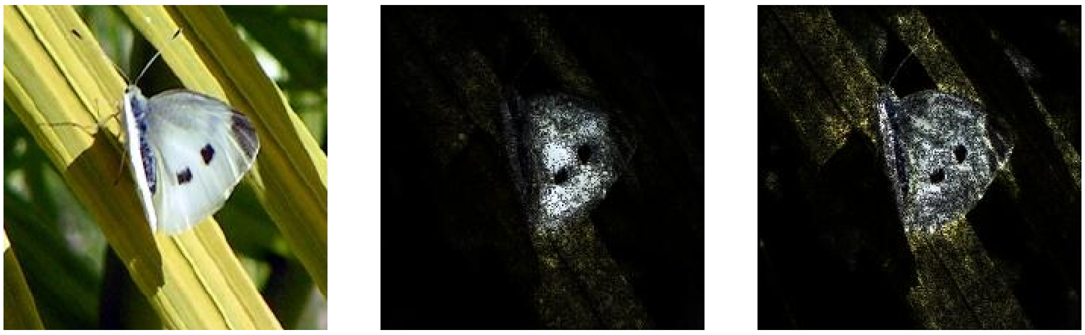
Neuron output:

$$\vec{y^{n,t}} = \tanh\left(\vec{c^{n,t}}\right) * \vec{o^{n,t}} \in \mathbb{R}^H \tag{4.14}$$

4.5 Attribution

As described in the introductory chapter 1, ANNs are usually described as black boxes whose internal behavior is hard to understand for humans. To be more precise, there is no obvious way to tell which generalized rules an ANN learned. Such desired rules would be of the form "Look at input x_1 and if its value is greater than 10 continue with rule R_2".

One way of improving this situation are so called *attribution* techniques. For a specific input, they compute and visualize the contribution of each input feature to each ANN's output [STY17]. However, attribution techniques do not expose the underlying rules on the basis of which an input feature is connected to the ANN's output.



(a) Original image. (b) GradInput. (c) IntGrad.

Figure 4.12: Visualization of GradInput (b) and IntGrad (c) attributions for an image (a). Images copied from Sundararajan, Taly, and Yan's paper [STY17].

Formally, we transfer Sundararajan, Taly, and Yan's attribution definition to multi-class RNNs as follows [STY17]. We adjusted the notation for consistency with this thesis' notation. Suppose we have an RNN $\hat{F_{\underline{W}}} : \mathbb{R}^{M \times U} \to \mathbb{R}^N$, a specific input matrix $\underline{x} = \left(x_{i,j}\right)$, and a baseline input matrix $\underline{b} = \left(b_{i,j}\right)$ with $\underline{x}, \underline{b} \in \mathbb{R}^{M \times U}$. For the k-th output neuron, an *attribution* of that neuron's output $\left(\hat{F_{\underline{W}}}\right)_k$ for input $\underline{x}$ relative to the baseline $\underline{b}$ is a matrix

$$A_k^{\hat{F_{\underline{W}}}}(\underline{x}, \underline{b}) = \left(a_{i,j}\right) \in \mathbb{R}^{M \times U} \tag{4.15}$$

where $a_{i,j}$ is the *contribution* of $x_{i,j}$ to the prediction $\left(\hat{F_{\underline{W}}}\right)_k$ of the k-th output neuron. In this thesis, for a specific input there is one attribution for each weakness and non-weakness.

The baseline $\underline{b}$ should be chosen such that the model's prediction for the baseline is neutral. As a result, each input feature's contribution is relative to the neutral contribution of the corresponding baseline feature. Sundararajan, Taly, and Yan suggest the black image as baseline for image processing ANNs and the zero embedding vector for embedding-based text processing. For both our feature representation, which we describe in section 5.2, we use zero for our features and the zero vector for our embeddings as baseline.

Section 4.5.1 introduces the simple *Gradient*Input* (GradInput) approach, which multiplies the gradients w.r.t. the input with the input itself. Based on this, *Integrated Gradients* (IntGrad) is the second approach, which calculates additional gradients to improve the results. It is described in section 4.5.2.

Figure 4.12 shows the resulting attributions in the context of image classification. Listing 8.1 in the appendix visualizes our utilization of these techniques on SCSs. We utilize both attribution techniques in our implementation to generate attribution overlays. Section 6.3 describes this process. Further visualizations and discussions of the results are located in sections 7.2.1 and 7.2.5.

4.5.1 Gradient*Input

The GradInput attribution technique computes the gradient of the outputs with respect to each input feature and multiplies each resulting gradient matrix with the input itself. Since no baseline is explicitly incorporated, it is implicitly assumed to be the zero matrix.

To explain the reason for multiplying with the input, imagine a single input feature whose value is zero. The input feature itself may still being connected to the model's output in a way that it highly contributes in general. However, since the input feature's value is zero in this case, the input feature does not contribute to output. By multiplying with the input, this is reflected in the attribution.

For brevity, the following definition is based on equation (4.15) and defines each input feature's contribution to the k-th output neuron as

$$a_{i,j} = x_{i,j} \cdot \frac{\partial \left(\hat{F_{\underline{W}}}\right)_k}{\partial X_{i,j}} (\underline{x}) \in \mathbb{R}.$$

$X_{i,j}$ denotes the i, j-th input dimension and $\left(\hat{F_{\underline{W}}}\right)_k$ the model function of the k-th output neuron. The gradient function is evaluated at position $\underline{x}$.

4.5.2 Integrated Gradients

Sundararajan, Taly, and Yan define two fundamental axioms, which every attribution technique should satisfy. They call them *Sensitivity(a)*[3] and *Implementation Invariance.*

Sensitivity(a) is defined as follows. If one input and the baseline differ in one feature *and* result in different predictions then the differing feature's contribution should be non-zero. GradInput does not satisfy this in general. For example in the one-dimensional case, suppose an ANN is given as $\hat{F_{\underline{W}}}(x) = 1 - \mathrm{ReLU}(1 - x)$. ReLU is the *Rectified Linear Unit* activation function $\mathrm{ReLU}(x) = \max(x, 0)$. The baseline $b := 0$ and an input $x^p := 2$ differ in their single feature and in their predictions of 0 and 1, respectively. However, the input's GradInput contribution is $2 \cdot 0 = 0$.

[3]They further define a part (b) which is not a part of their fundamental axioms.

An implementation invariant attribution technique produces identical attributions for functionally equivalent ANNs. Sundararajan, Taly, and Yan show that many other attribution techniques from the literature break implementation invariance [STY17].

As a solution, they propose their IntGrad attribution technique that satisfies both sensitivity(a) and implementation invariance. Instead of only computing gradients for the input, they compute gradients for all points on the straight path between the baseline and the input. Then, they accumulate the separate gradients to achieve a more robust and focused attribution.

As above, the following definition is based on equation (4.15) and defines each input feature's contribution to the k-th output neuron as

$$a_{i,j} = \left(x_{i,j} - b_{i,j}\right) \cdot \int_{\gamma=0}^{1} \frac{\partial \left(\hat{F_{\underline{W}}}\right)_k}{\partial X_{i,j}} \left(\underline{b} + \gamma \cdot \left(\underline{x} - \underline{b}\right)\right) d\gamma.$$

In contrast to GradInput, for each final attribution, gradients w.r.t. all baseline-input combination that are described by the straight path between the baseline and the input are integrated. In practice, the integral is replaced by a finite sum. This corresponds to taking discrete steps on the straight path from the baseline to the input.

Chapter 5

Data

This chapter is about the SCS data and their representation for feeding them to an ANN. The first section 5.1 introduces the different datasets on the basis of which we evaluate our tool. Additionally, it describes the dataset's statistical properties and their strengths and deficiencies. Section 5.2 explains which information we extract from the datasets and how we represent these information with real-valued numbers required by our tool.

5.1 Datasets

This section describes the three datasets utilized in this thesis. Each dataset is a collection of pairs of SCSs and their associated labels. The label indicates whether its associated SCS is vulnerable and if so, to which weakness it belongs. The SCSs are written in C or C++ and usually consist of a function definition together with its callee functions' definitions.

We use the *Common Weakness Enumeration* (CWE) for identification of weaknesses. According to the CWE's website[1] the CWE is a "community-developed list of common software security weaknesses" serving as a "baseline for weakness identification". The CWE should not be confused with CVE, which is an acronym for *Common Vulnerabilities and Exposures.* The CVE list contains known vulnerabilities in software systems and each of these vulnerabilities is an instance of at least one weakness from the CWE.

The first and biggest dataset is the Juliet Test Suite for C/C++. We abbreviate its name as *Juliet* and describe it in section 5.1.1. Section 5.1.2 introduces the second dataset, which we name *MemNet.* For deeper evaluation of the extrapolation capabilities

[1]https://cwe.mitre.org/ (retrieved on July 14, 2019)

of NALUs, we also generated a simple dataset with buffer overflow vulnerabilities ourselves. We call this dataset *ArrDeclAccess* and describe it in more detail in section 5.1.3.

All datasets are synthetic, i.e., its SCSs were artificially generated by a generator tool following some rules. In comparison to natural SCSs written by humans, those SCSs are simpler and less versatile [Nat12]. As a consequence, statements made in this thesis are limited to synthetic SCSs and cannot be simply transferred to natural SCSs found in the real world. Section 7.3 gives more information about this and other threats to validity.

5.1.1 Juliet

We use the Juliet Test Suite for C/C++ in version 1.3 [Nat12][2]. It contains 64 099 *test cases* for 118 CWE weaknesses [Bla18]. Each test case consists of a C or C++ SCS having a vulnerability of a given weakness and usually of one or two fixed, non-vulnerable versions of that SCS. According to the dataset's user guide, the "test cases cover 11 of the 2011 CWE/SANS Top 25 Most Dangerous Software Errors" with the other 14 CWEs mainly being design issues [Nat12]. Another reason why we chose this dataset is that many other researchers use the dataset to evaluate the prediction power of their static analyzers [Rus+18; GP15; SSV18].

The Center for Assured Software (CAS) at the United States National Security Agency (NSA) created the Juliet dataset with the goal to assess the capabilities of static analyzers. It is publicly available on the website[3] of the United States' National Institute of Standards and Technology (NIST). The ZIP archive from the website contains documentation files, supporting source code files, and a directory with corresponding test cases for each CWE weakness.

In summary, the Juliet dataset contains 64 099 test cases. Table 5.1 shows that the smallest and biggest number of test cases for a CWE weakness are 1 and 5922, respectively. On average, for each CWE weakness 543 test cases are present in the Juliet dataset. Per-CWE-weakness statistics are located in the appendix in table 8.2. 22 287 test cases require a Microsoft Windows environment for successful parsing. This is described in more detail in the context of functional variants below as well as in the context of our parsing implementation in section 6.1.2.

Each test case consists of one or multiple C or C++ source code files. These source code files contain – among others – one function that is vulnerable to the weakness

[2]There exists no user guide document for version 1.3. Instead, we reference and use the user guide for version 1.2 and the changelog document as advised by the NIST at https://samate.nist.gov/SRD/around.php#juliet_documents (retrieved on July 14, 2019).

[3]https://samate.nist.gov/SRD/testsuite.php#standalone (retrieved on July 14, 2019)

Table 5.1: Aggregations of test case statistics over separate CWEs of the Juliet dataset. Parenthesis denote the number of the non-parenthesised test cases that require a Microsoft Windows Environment. Statistics for each separate CWE are located in the appendix in table 8.2.

	Test Cases Count (Need Windows)					
	Overall		**Not Parsable**		**Parsable**	
Sum	64 099	(22 287)	2472	(2472)	61 627	(19 815)
Min	1	(0)	0	(0)	1	(0)
Max	5922	(3840)	1220	(1220)	5922	(2620)
Mean	543	(189)	21	(21)	522	(168)
SD	1085	(501)	122	(122)	1039	(414)

identified by the test case's parent CWE directory. In most cases the source code files also contain fixed functions which do not have a vulnerability of that weakness. A list of the CWE IDs and corresponding CWE names that are present in the Juliet dataset is located in the appendix in table 8.1.

For example, the CWE directory `CWE124_Buffer_Underwrite` contains test cases which have a vulnerability of the buffer underwrite weakness with the CWE ID 124. `CWE124_Buffer_Underwrite__char_declare_cpy_17.c` is an example for a test case's single source code file. It contains the primary bad and primary good functions `void CWE124_Buffer_Underwrite__char_declare_cpy_17_bad()` and `void CWE124_Buffer_Underwrite__char_declare_cpy_17_good()`, respectively. Both are shown in listing 5.1. The primary good function only calls the secondary good function `static void goodG2B()`. The other secondary good function would be called in the same way. Apart from that, the only difference is line 8, in which the pointer `data` is set to before the allocated memory in the bad function. Due to this, `'C'` characters are written to the stack outside the allocated range resulting in undefined behavior. In the good function, the pointer is set to the start of the allocated memory leading to well-defined behavior.

Multiple test cases for the same weakness differ in their *functional variant* and *flow variant* [Nat12]. The functional variant indicates which specific language constructs are utilized to yield a vulnerable SCS. This contains the use of different variable types, different ways of gathering data, and different ways of triggering the weakness. Then, for a concrete functional variant the flow variant specifies how the vulnerable source code lines are obfuscated. The obfuscation does not change the program behavior but increases the difficulty of identifying the vulnerable lines. This also simulates that vulnerable lines rarely occur in isolation in real world. Both functional and flow variants are also applied to the non-vulnerable versions of each SCS.

```c
void CWE124_Buffer_Underwrite__char_
  ↪ declare_cpy_17_bad(){
  int i;
  char * data;
  char dataBuffer[100];
  memset(dataBuffer, 'A', 100-1);
  dataBuffer[100-1] = '\0';
  for(i = 0; i < 1; i++) {
    /* FLAW: Set data pointer to
      ↪ before the allocated memory
      ↪ buffer */
    data = dataBuffer - 8;
  }
  char source[100];
  memset(source, 'C', 100-1); /* fill
    ↪ with 'C's */
  source[100-1] = '\0'; /* null
    ↪ terminate */
  /* POTENTIAL FLAW: Possibly copying
    ↪ data to memory before the
    ↪ destination buffer */
  strcpy(data, source);
  printLine(data);
}
```

(a) Primary bad function.

```c
static void goodG2B() {
  int h;
  char * data;
  char dataBuffer[100];
  memset(dataBuffer, 'A', 100-1);
  dataBuffer[100-1] = '\0';
  for(h = 0; h < 1; h++) {
    /* FIX: Set data pointer to
      ↪ the allocated memory buffer
      ↪ */
    data = dataBuffer;
  }
  char source[100];
  memset(source, 'C', 100-1); /* fill
    ↪ with 'C's */
  source[100-1] = '\0'; /* null
    ↪ terminate */
  /* POTENTIAL FLAW: Possibly copying
    ↪ data to memory before the
    ↪ destination buffer */
  strcpy(data, source);
  printLine(data);
}
void CWE124_Buffer_Underwrite__char_
  ↪ declare_cpy_17_good(){
  goodG2B();
}
```

(b) Secondary and primary good functions.

Listing 5.1: Primary and secondary functions of test case `CWE124_Buffer_Underwrite__char_declare_cpy_17`. New lines and no-op scopes were removed for space reasons.

The flow variant is further divided into *control flow variants* and *data flow variants* [Nat12]. The former add if-else, switch, or loop structures that do not prevent the vulnerable source code lines from being reachable. The latter split the vulnerable source code lines into multiple functions and files, and they add corresponding function calls to ensure the original behavior.

The functional variant further specifies whether a test case uses the Microsoft Windows API and therefore requires a Microsoft Windows environment for successful parsing. The presence of `w32` in the functional variant is the indicator for that requirement [Nat12]. Since we utilize a Linux operating system for this thesis, we cannot provide a real Microsoft Windows environment. To avoid losing a third of the dataset, we create an artificial Microsoft Windows environment. This is part of our implementation and thus described in section 6.1.2 in more detail.

For the example described above, the functional variant is specified by `char_declare_cpy`. This means that the SCS declares an array of type **`char`** and copies it using `strcopy`. Other functional variants use combinations of **`wchar_t`** array type, allocation with `ALLOCA` or on the heap, and copying with `memcpy`, with `memmove`, or with a `for` loop. More complex functional variants receive data via a socket, read data from console input, or create the data randomly. In all three cases, they convert the data to an integer and use it as an array index without checking for neither positiveness nor the array's bounds.

The flow variant number of the examples in listing 5.1 is 17, which corresponds to a control flow variant that adds a one-iteration for loop around one or multiple source code lines. Such a loop can be seen in lines 7 to 10 of the above-mentioned listings. A data flow variant of that example would have the lines 11 to 16 moved to a second, new function which would be located in a newly created file. The second function would be called at the end of the initial function and `data` would be passed as an argument to it.

For some CWEs the NSA's generation process yielded artifacts which allow an easy discrimination of good and bad function without considering the actual weakness. For example, CWE 615 (Info Exposure by Comment) SCSs log a user in some system and print a success message. The bad functions also contain a vulnerable comment which contains the username and password. However, not only the comments differ between the good and bad functions but also the message which is printed after a successful login. In the bad functions the string `"User logged in successfully"` is printed. The good functions print `"User logged in successfully with password"`.

In summary, the Juliet dataset consists of SCSs with many weaknesses and many variants of these. However, the generation of the SCS variants follows fixed rules resulting in lower variance between the SCS than with natural SCS. This may lead to the RNN generalizing the generation process' rules and artifacts instead of the program's logic.

For splitting the dataset, we distribute the pairs of SCSs and labels randomly into disjoint training, validation, and test sets. 70 % of the pairs form the training set, 15 % the validation set, and 15 % the test set. We refer to this split as *random*.

5.1.2 MemNet

The MemNet dataset consists of 14 000 SCSs for the buffer overflow weakness written in C. Each SCS consists of exactly one function `void fun()` which declares and initializes one or more `char` arrays. The last line of each function accesses one of these arrays at a certain index which may be out of that array's bounds. Listing 5.2 contains an example for such a vulnerable SCS. A separate label file specifies for each SCS whether it has or does not have a vulnerability of a buffer overflow weakness.

Choi et al. generated that dataset for their paper that describes the usage of memory networks for detection of buffer overflows[4] [Cho+17]. These networks are specialized in detecting buffer overflow vulnerabilities in SCSs, and they inspired us in naming the dataset. The dataset is publicly available on GitHub[5]. It was generated and used by Choi et al. for benchmarking their so-called memory networks.

The dataset is already split into a training set of 10 000 and four test sets of 1000 SCSs each. A validation set is not given.

The SCSs in the four test sets have an increasing level of complexity. Increasing means that each test set contains SCSs of its level and of its preceding levels. Example SCSs for each level are given in figure 5.1, which we took from Choi et al.'s paper [Cho+17]. We notice the SCSs from the paper in the figures to slightly differ from the SCS in the dataset. For example, no return statements are present in the dataset.

The level 1 test set's SCSs only contain array declarations where the arrays' lengths are specified by integer literals. Each SCS's possibly vulnerable array access is done with the subscript operator and an index which is specified by an integer literal too. The level 2 test set additionally consists of SCSs containing array accesses through copying from other arrays with `strcopy` or `memcpy`. Level 3 adds SCSs which declare and initialize arrays of lengths specified by integer variables. A variant of this is the declaration of a pointer to `char` followed by an allocation with `malloc` of length given by the integer variable. The level 4 test set also contains SCSs in which the integer variables are re-assigned before using them for array declarations.

The structure of the test sets' SCSs differs from the structure of the training ones. For example, assignments with `NULL` such as `entity_1 = NULL;` only exist in the training set. Since Choi et al. do not address such differences, we assume them to be artifacts of the generation process.

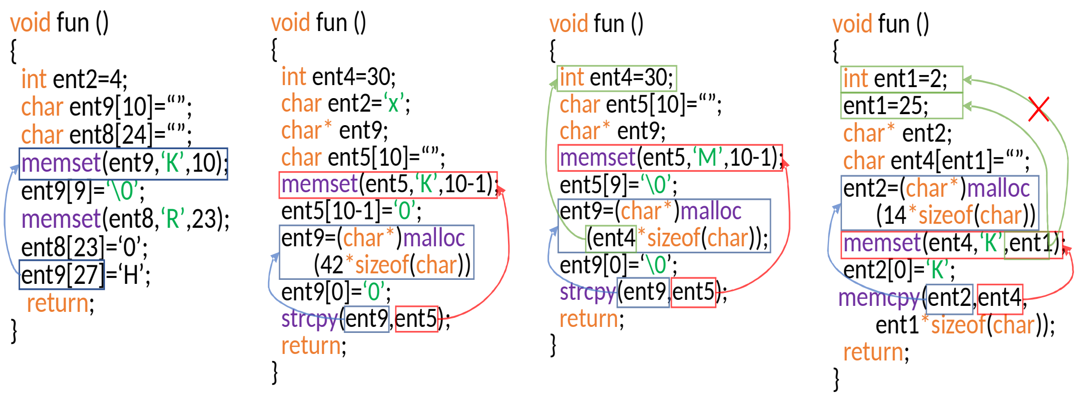
Problems with the just-mentioned assignments with `NULL` arise from their semantic invalidity in C and C++. This is due to the fact that variables of array type are not

[4]The synonymous term "buffer overrun" is used in the paper. We use the term "buffer overflow" throughout this thesis.

[5]https://github.com/mjc92/buffer_overrun_memory_networks (retrieved on May 20, 2019)

```c
void fun ()
{
  char entity_1[1] = "";
  entity_1 = NULL;
  char entity_3[61] = "";
  entity_3 = NULL;
  char entity_5 = 'r';
  char entity_2 = 'w';
  memset(entity_1, 'p', 1-1);
  entity_1[1-1]='\0';
  entity_1[46] = 'a';
}
```

Listing 5.2: Vulnerable SCS from the MemNet training set.



```c
void fun ()
{
 int ent2=4;
 char ent9[10]="";
 char ent8[24]="";
 memset(ent9,'K',10);
 ent9[9]='\0';
 memset(ent8,'R',23);
 ent8[23]='0';
 ent9[27]='H';
  return;
}
```

(a) Level 1.

```c
void fun ()
{
 int ent4=30;
 char ent2='x';
 char* ent9;
 char ent5[10]="";
 memset(ent5,'K',10-1);
 ent5[10-1]='0';
 ent9=(char*)malloc
      (42*sizeof(char))
 ent9[0]='0';
 strcpy(ent9,ent5);
 return;
}
```

(b) Level 2.

```c
void fun ()
{
 int ent4=30;
 char ent5[10]="";
 char* ent9;
 memset(ent5,'M',10-1);
 ent5[9]='\0';
 ent9=(char*)malloc
   (ent4*sizeof(char));
 ent9[0]='\0';
 strcpy(ent9,ent5);
 return;
}
```

(c) Level 3.

```c
void fun ()
{
 int ent1=2;
 ent1=25;
 char* ent2;
 char ent4[ent1]="";
 ent2=(char*)malloc
     (14*sizeof(char))
 memset(ent4,'K',ent1);
 ent2[0]='K';
 memcpy(ent2,ent4,
     ent1*sizeof(char));
 return;
}
```

(d) Level 4.

Figure 5.1: Examples of SCSs from the MemNet dataset with increasing level of complexity. Copied from Choi et al.'s paper [Cho+17].

assignable. We observe another problem with semantically ill-formed initializations of variable-length arrays as in **char** `entity_2[entity_0] = "";`. At least one of these problems occurs in each SCS from the training set. We suspect that Choi et al. did not notice these problems because they used a token-based approach. Splitting an SCS into tokens does not require the SCS to be semantically valid. The SCS becomes valid by adding appropriate include directives and by removing the problematic initializations and assignments.

In summary, MemNet contains buffer overflow SCS of different complexity. We base the dataset split on the given splits described above. To obtain a validation set, we randomly select 15 % of the training set. We refer to this split as MNPaper since it was provided by Choi et al.' paper.

5.1.3 ArrDeclAccess

We created a simple dataset in which each SCS declares an array of certain length followed by accessing the array up to a certain element. An SCS will be vulnerable to a buffer overflow weakness if the array is accessed outside its declared range.

The motivation for this dataset originates from Trask et al.'s paper [Tra+18]. They show that ANNs are usually not able to extrapolate basic arithmetic operations to the range outside the values encountered during training well. As solution, they propose NALUs, which we describe in section 4.3. For the evaluation of the NALUs' extrapolation ability in section 7.2.2, we use the ArrDeclAccess dataset with different ranges of lengths for the training/validation and the test set.

Each SCS follows the schema from listing 5.3. Each of the SCSs declares an array `arr` whose length is specified by l_{decl} in line 2. Lines 3 to 5 form a for loop, which iterates over `arr` from 0 to l_{access} and accesses each array element excluding the l_{access}-th element. In the final SCS, both l_{decl} and l_{access} are replaced by integer literals corresponding to their values. An SCS will be vulnerable if $l_{access} > l_{decl}$.

For the training and validation set, we set the maximum length as l_{max}^{train}such that both l_{decl} and l_{access} are $\in \left[0, l_{max}^{train}\right]$. We generate SCSs in two variants and merge them for a combined training and validation set. We shuffle this set and use 15 % of it for the validation set. The remaining SCSs form the training set.

With the *around-decision-border* variant, we iterate from 0 to l_{max}^{train}, i.e., $\in \left[0, l_{max}^{train}\right[$, and generate one non-vulnerable and one vulnerable SCS in each iteration. Let i be the current index, then the first SCS has l_{decl} and l_{access} set to i and i, respectively. The second's values for l_{decl} and l_{access} are i and $i+1$, respectively.

The second variant, *random*, does not ensure a maximum distance of one between l_{decl} and l_{access}. Based on a given number $r_{max} \in \mathbb{N}$, we generate r_{max} non-vulnerable and r_{max} vulnerable SCSs. For the first, non-vulnerable half of SCSs, we randomly draw a value $j \in \left[0, l_{max}^{train}\right]$ for l_{decl}. Then, for l_{access}, we draw its value from $[0, j]$. For the second, vulnerable half, the first interval's upper bound is exclusive, and we draw l_{access}'s value from $\left]j, l_{max}^{train}\right]$.

For the test set, we use $l_{max}^{test} > l_{max}^{train}$ as maximum length. In contrast to before, we only add around-decision-border SCSs.

As default values for the training/validation set, we use $l_{max}^{train} := 100$ and $r_{max} := 500$. We also add SCSs generated with the around-decision-border variant three times

```
int main() {
  int arr[$l_{decl}$];
  for(int i = 0; i < $l_{access}$; i += 1) {
    arr[i] = i;
  }
  return 0;
}
```

Listing 5.3: Generation schema of SCSs in the ArrDeclAccess dataset. l_{decl} is the length of memory allocated for the array. l_{access} is the length, up to which the array elements are accessed. An SCS will be vulnerable, if $l_{access} > l_{decl}$.

to maintain a better SCS-count-wise balance between both variants. This results in 1600 training/validation SCSs. For the test set, we use $l_{max}^{train} := 5000$ as default, which results in 10 000 SCSs. In situations that require a larger dataset, we multiple each default value by ten.

5.2 Feature Representations

This section describes two different approaches for representing each SCS as a sequence of real-valued vectors. This representation step is necessary since the RNNs used in this thesis only accept variable-length sequences of fixed-length, real-valued vectors as input. As mentioned in chapter 4, feature vector is another name for such a real-valued vector since they represent the features of the SCS. Analogously, *feature sequences* is the name for a sequence of feature vectors.

The choice of feature representation is an important part of training an RNN and is often considered more important than the training algorithm itself [Gra08]. Requirements on the feature representation are completeness and compactness. A complete representation contains all information necessary to correctly predict outputs based on inputs. A compact representation should be reasonable small to avoid the curse of high dimensionality [Bis+95; Gra08].

Approaches from the literature often lack completeness because they do no incorporate information about the name and type of a variable or function, about values of float, boolean, or string literals, and about an operator's kind [Rus+18; Mou+16]. Since these information are important for distinguishing vulnerable and non-vulnerable SCSs, both our feature representations employ them to establish features. They only differ in how and in which features are built from those information. We elaborate on this in more detail later.

Our two feature representations are a basic, manually crafted and a more sophisticated, embedding-based one. We therefore call the former feature representation *BasFR* and the latter *EmbFR*. To easily gather semantic information from SCSs such as a variable's type, both representations are based on Abstract Syntax Trees (ASTs). We introduce these in section 5.2.1. Afterwards, section 5.2.2 identifies and describes four choices we have to make during the creation of our representations. We also provide illustrations for both feature representations in that section.

5.2.1 Abstract Syntax Trees

As mentioned above, we base our feature representations on *Abstract Syntax Trees* (ASTs). ANN-based static analyzers presented in the literature almost always base their feature representation either on ASTs or on *tokens* [Mou+16; Rus+18; Cho+17]. The following paragraphs introduce both terms, discuss their advantages and disadvantages, and explain why we chose an AST-based approach.

We begin by looking at the first phases of compilation that convert an SCS to its AST. Figure 5.2 depicts them. Succeeding phases, which for example generate an executable file, are out of this thesis' scope.

During the *scan* phase, the SCS is interpreted as a stream of characters and fed to the scanner, which is also called lexer. Based on the grammar specification of the source language, which is C or C++ here, the scanner reads the SCS and aggregates multiple characters to tokens. The tokens are the SCS's elemental syntactic entities. Each of these tokens is an instance of a certain token class.

For example, the SCS `bar(i)` consists of the four tokens `bar`, `(`, `i`, and `)`, which are instances of the token classes `identifier`, `opening parenthesis`, `identifier`, and `closing parenthesis`, respectively.

Parse is the second phase. The parser reads the token stream produces by the scanning phase and builds a tree structure. This structure is called *parse tree* and represents the hierarchical relationships between the SCS's syntactic entities.

The parse tree often contains chains of nodes, which arise from nested production rules in the grammar specification. The *weed* phase squashes these chains. The following two phases *process symbols* and *check types*. By doing so, they analyse the static semantics of the weeded tree. This includes connecting declarations and uses of symbols such as variables or functions, and determining types of all expressions. Both take scopes like namespaces, classes, and functions into account. As a consequence,

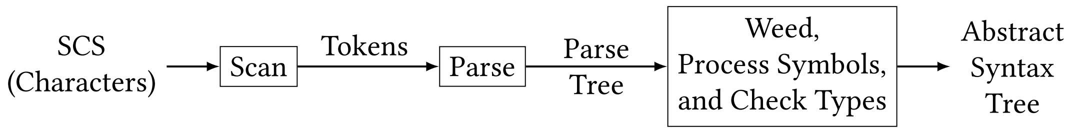
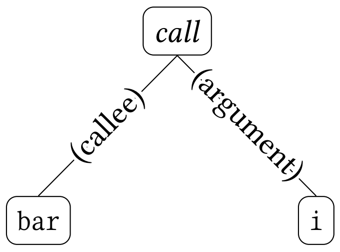


Figure 5.2: First phases of compilation.

ambiguities such as overloaded functions can often be resolved. Since these phases create an abstraction level, the resulting tree is called abstract syntax tree.

We use Clang for parsing as further describes in section 6.1.1. The AST build by Clang's parser corresponds to the result of the above-described phases. For brevity, we refer to the construction of such an AST from an SCS as *parsing* hereafter.



To continue the example from above, we assume that the four tokens denote a call of function `bar` with one argument `i`. The resulting AST is depicted on the right. The root node describes that `bar(i)` is a call expression, i.e., an expression in which a function is called and whose value is the function's return value. The child nodes give in-detail information about the call. The left child node was created from the token `bar` and represents the callee expression[6]. If `bar` identifies a function that child would be a node referencing `bar`'s declaration. The right child is based on the token `i` and denotes the argument expression. Analogously, it is a node referencing `i`'s declaration. If `bar` was overloaded, `i`'s type would select the appropriate overloaded version. Apart from the tree structure, the opening and closing parentheses are no further represented in the AST.

After introducing tokens and ASTs, we discuss their advantages and disadvantages in the context of vulnerability detection as well as why we chose an AST-based approach.

Token-based and AST-based approaches have different requirements on the SCSs they work on. While the former only require the SCSs to be syntactically valid, the latter also require them to be semantically valid. In general, token-based approaches can be applied to a wider range of SCSs[7].

To demonstrate this, we expand the above-mentioned example to the SCS in listing 5.4. It uses `bar`, which is declared outside the SCS. Since that declaration is not visible

[6]In C and C++, an expression names the callee function. That expression may be an arbitrary expression returning a function pointer.

[7]An exception are programming languages that require interlocking of the scan and parse phases. For example in C# starting with version 5.0, the token `await` will be a keyword if it is located in a function marked with `async`. Otherwise, it will be an identifier.

```
1 void foo(int len) {
2   char arr[len];
3   for (int i = 0; i <= len; i += 1) {
4     arr[i] = bar(i);
5   }
6 }
```

Listing 5.4: Example SCS with undeclared `bar`, which is syntactically valid but semantically ill-formed.

to the parser, the SCS is semantically ill-formed. However, it is still syntactically correct and can be further processed by a token-based approach. As a result, this approach has the chance to detect the buffer overflow vulnerability[8] in line 4. An AST-based approach, by contrast, cannot further process the SCS and thus does not detect the vulnerability.

The advantage of the AST is its inclusion of semantic information and its abstraction. As a result, it is more compact when comparing token counts and AST node counts, it has most ambiguities resolved resulting in more local information, and it directly exposes relationships between an SCS's entities.

For example, in `foo(42)` `foo` could in theory identify a function, a class, or a variable resulting in either a function call, an instance creation, or the invocation of the custom call operator on the object identified by the variable. If we only consider local tokens it will be ambiguous what the token's program behavior is. However, since the parser resolves such ambiguities, `foo(42)`'s AST nodes expose that a function call is happening.

5.2.2 BasFR and EmbFR

As mentioned above, both our approaches are based on ASTs. As a consequence, we parse each SCS and use the resulting AST as starting point for feature sequence generation. There is exactly one feature sequence for each SCS.

We identify the following four choices that we have to make during the creation process of our feature representation approaches.

1. Which order should be used for sequencing of the AST nodes?
2. Which kinds of AST nodes should contribute to the feature sequences?

[8]We assume that `bar` does not always break the for loop and does not always terminate the program.

3. Which features should we build from each AST node's information, i.e., which schema should we follow to receive concrete feature values from an AST node's information?

4. How should we construct the RNN's input layer(s) such that they can appropriately receive and forward the feature vectors?

The first two choices are independent of the feature representation approach.

We sequence the AST nodes with a depth-first, pre-order traverse through the AST. We chose this order since it corresponds to each AST node being in front of its children in the sequence. This way the coarse information comes first followed by more in-detail information. For example, the operator itself precedes its operands as well as the if statement precedes its condition, if part, else-if parts, and else part.

For the second choice, we ignore all AST nodes that represent either superfluous parentheses or casts. We also extend the AST by adding an auxiliary sibling node for each non-leaf AST node. We add these nodes as direct, right siblings and call them *end-of* nodes. To avoid two root nodes – the previous root and its end-of node – we create a new root as parent of both.

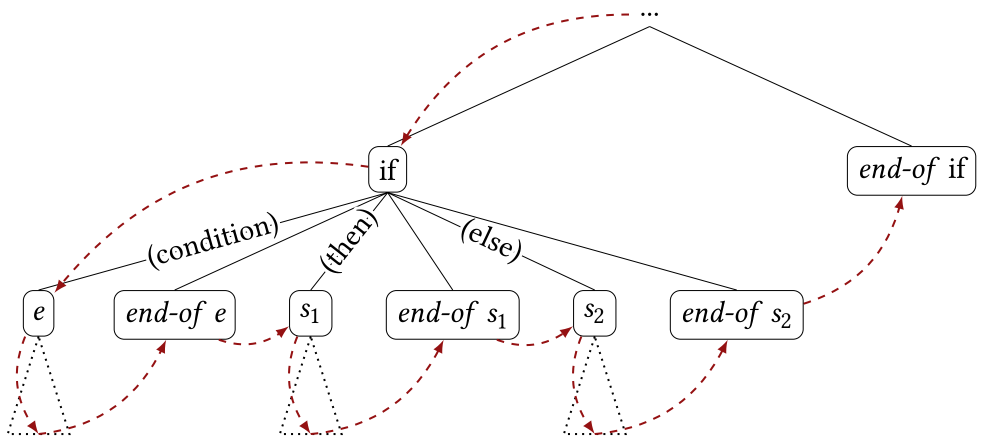
Each end-of node designates the end of the semantic entity that its corresponding sibling start node begun. Adding the end-of nodes prevents the loss of structural information since it allows the reconstruction of the AST's structure from a sequence of AST nodes. Without the end-of nodes, multiple differently behaving SCSs could be represented by the same feature sequence.

Figure 5.3 exemplifies this by showing two ASTs with end-of nodes. The two ASTs' corresponding SCSs are **`if`**$(e)\{\ s_1\ \}$ **`else`** $\{\ s_2\ \}$ and **`if`**$(e)\{\ s_1\ \}\ s_2$. The former executes either statement s_1 or s_2 depending on expression e. The latter executed s_1 depending on e and always executes s_2. The dashed, red arrows denote the depth-first, pre-order traverses, whose resulting sequences differ between the two ASTs. When ignoring the end-of nodes, the resulting sequences are identical.

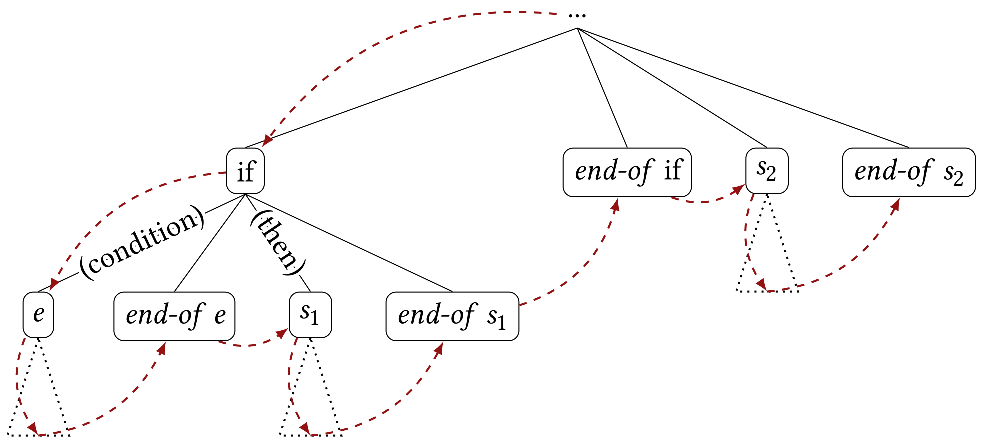
The third and fourth choices are strongly related for multiple reasons.

One reason is that each feature vector element must be fed to some input neuron and that each input neuron must receive values from a feature vector element. To ensure this and to maintain simplicity, we decide to use a bijective mapping between feature vector elements and input neurons. This also applies in case of multiple, parallel input layers that we further describe below.

Another reason is relevant with embedding layers. Since they only accept integer values as input we must ensure such values for feature vector elements flowing to embedding layers.



(a) **if**(e){ s_1 } **else** { s_2 }.



(b) **if**(e){ s_1 } s_2.

Figure 5.3: ASTs with end-of nodes for two differently behaving SCSs. Dashed, red arrows mark the depth-first, pre-order traverses. Without end-of nodes, both traverses would be identical.

This paragraph describes which features are built in approaches from the literature. They use information about the token's class or the AST node's kind to build a feature [Rus+18; Mou+16]. Russell et al. also add a feature for an integer literal token [Rus+18]. Harer et al.'s token-based approach builds further features from identifier-tokens and string-tokens. They map the token values to integer values, which allows them to recognize variables and strings. They reset the identifier mapping after each function to reflect that variables with the same name in different functions are completely unrelated in general.

These features are important for distinguishing vulnerable and non-vulnerable SCSs. For example, to decide whether an array access may be a buffer overflow, a static analyzer must compare the access index to the accessed array's previously seen declared length. Selecting the matching declaration can only be done based on the array's name.

Using an AST-based approach allows to not only use identifier tokens for variable, function, type identification but to also use fully-qualified names. These incorporate the contexts in which the entity is declared. In addition, function calls can be matched with their respective callee function.

Building a feature representation, which employs this high amount of an AST node's information, is the requirement of our first RQ. We therefore build features for the AST node's

- kind,
- operator, if the node represents a unary, binary, or ternary operator,
- variable or function, if the node represents a declaration of or a reference to a variable or function,
- value, if the node represents a literal, and
- the node's type, if the node represents a declaration or expression.

Which exact AST node information we use for these features and how we arrange the features in the resulting feature vector differs between our two feature representations. The following paragraphs describe our two feature representations by explaining their respective feature vector elements and their RNN input layers structure.

BasFR For the BasFR feature representation, we use a static, hard-coded schema to map an AST node's information to concrete feature values. Our prior knowledge about this thesis' context and programming in general is incorporated in this process. The RNN's input layer is a single LSTM layer, which precedes a NALU and two FC layers. The fact that the static schema and the simple RNN form a very basic feature representation gave it its name.

Figure 5.4 gives a visual overview of BasFR. It shows a feature sequence, the feature sequence's last feature vector's elements, the RNN's layers, and their data flow. U denotes the length of the sequence. The ellipsis symbolizes the sequence's other $U - 1$ feature vectors, which consists of the same elements. The flows' annotations illustrate the lengths in each dimension of their corresponding data. The neuron counts N^{LSTM}, N^{NALU}, and N^{FC} are HPs and depend on our tool configuration.

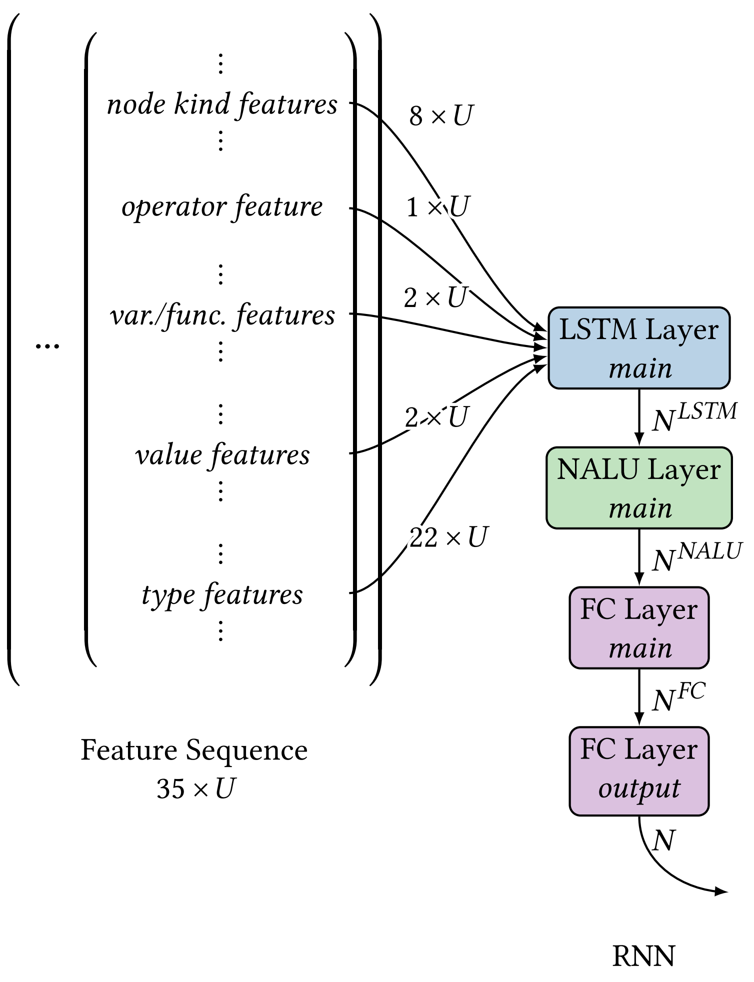


Figure 5.4: BasFR's feature vector elements, RNN structure, and how both are connected. The data flows' labels denote the data's length in each dimension

N specifies the RNN's output neuron count which corresponds to the number of weaknesses plus non-weakness of the tool configuration's dataset. Each feature vector has 35 elements, i.e., is $\in \mathbb{R}^{35}$. There are eight features for the AST node's kind, one operator kind feature, two features for the variable's or function's kind and ID, two features for the value and its kind, and 22 features for the type. Thus, $\hat{F_{\underline{W}}} : \mathbb{R}^{35 \times U} \to \mathbb{R}^{N}$.

The following paragraphs describe the feature vector elements.

- The first eight elements contain features determined by the AST **node's kind**. Examples for such kinds are `if`, `for`, `while`, and `return` statement as well as `call`, `declaration-reference`, and `operator` expression. The element's feature values are between zero and 30. A higher feature value corresponds to the AST node being more specialized with respect to that feature.

The first three elements denote whether the AST node is a declaration, expression, or statement, respectively. Other elements indicate that the AST node represents an entity interfering with the program's control flow, an entity referencing another entity, or the end of a range.

- The ninth element contains a non-zero feature value if the AST node represents an **operator** expression. The concrete feature values are based on a mapping, which maps the operator's strings to an integer in the interval $[1, 100]$. This mapping also clusters the operators since it maps similar behaving operators to similarly large integers. Examples for resulting clusters are binary logical, bitwise, arithmetic, and assignment operators, respectively. A limitation arises from the fact that the operator strings are used as input to the mapping. Thus, increment and decrement operators only differing in being pre-fix or post-fix are not distinguished. Since all of this thesis' datasets only use the increment or decrement operator as separate statements, this drawback yields no problem. For other datasets, unique operator strings or another identification method should be used instead.

- The **variable/function** features are contained in two feature vector elements and reflect the kind and the ID of a variable or function. When talking about variables and functions, we also mean member variables and member functions. Those features are available for both AST nodes representing a variable or function declaration and AST nodes representing a reference to a variable or function.

  For the kind, we distinguish between whether the variable or function is declared inside or outside of the dataset's source code files. For example, a variable from the standard library has another kind feature than a variable in a primary bad function.

  The variable/function ID feature is based on the variable's or function's fully qualified name. We further extend the fully qualified name by the relative path of the source code file in which the variable or function is declared. An example for such an extended fully qualfied name is `testcases/CWE843_Type_Confusion/CWE843_Type_Confusion__char_45.c:goodG2B()::data`, which identifies the variable `data` located in the file `CWE843_Type_Confusion__char_45.c`'s function `goodG2B`. This allows distinguishing identically named variables from different translation units.

  For both variable/function kinds, which denote whether a variable or function is declared inside our outside the dataset, we create a separate mapping. Each maps its corresponding extended fully qualified names to increasing integers such that equal variables or functions are represented by the same integer [Har+18].

We reset the mapping for the variables/functions that are declared inside the dataset after each SCS to work around the fact that in each test case's source code files the primary bad function is located in front of the good functions. Otherwise, the magnitude of the integers would be an easy-to-learn feature to distinguish bad and good SCS. The second mapping for variables/functions declared outside the dataset is persistent over all SCS. This is based on the fact, that for example `malloc` from the standard library denotes the same function in each SCS.

- The two **value** features represent an AST node's literal, if any. The first feature reflects the literal kind. We distinguish literals convertible to float from string literals. In case of the former, the second feature is the literal itself converted to float. In case of the latter, we map the string to an integer using a global mapping. We do not reset this mapping since strings are constant and only identified by their character sequence. As a drawback, the length of a string is not part of the feature representation unless the string is the initializer of an array of constant, matching length.

- To represent the **type** associated with an AST node, we choose 22 features. Before representing the type, we remove all syntactic sugar such as typedefs from the type. Without this invariance – according to Russell et al. – specific syntactic sugar could be learned for distinguishing weaknesses [Rus+18].

  Two features represent the underlying type's kind and ID. Four features each represent one of up to five overlying types. These four features are whether the respective overlying type is a reference type, a pointer type, or an array type, respectively, as well as the type's array length. As a limitation, multiple types, which only differ in the sixth or higher overlying type, are represented by the same features.

  For example, for the type `int (*&)[42][7]` the first two features represent the underlying type `int`. `int`'s kind is `built-in`, which is represented by `1`. Since `int` is the first type, its mapped ID is `1`. We represent the four overlying types `&`, `*`, `[42]`, and `[7]` by the 16 features' values $1, 0, 0, -1, \quad 0, 1, 0, -1, \quad 0, 0, 1, 42, 0, 0, 1, 7$. Since there is no fifth overlying type, its four features have their default values $0, 0, 0, -1$ indicating no reference, pointer, or array type.

EmbFR The EmbFR feature representation combines more sophisticated aspects like the utilization of self-learning, embedding-based feature mappings and the use of different input layers for different feature vector elements.

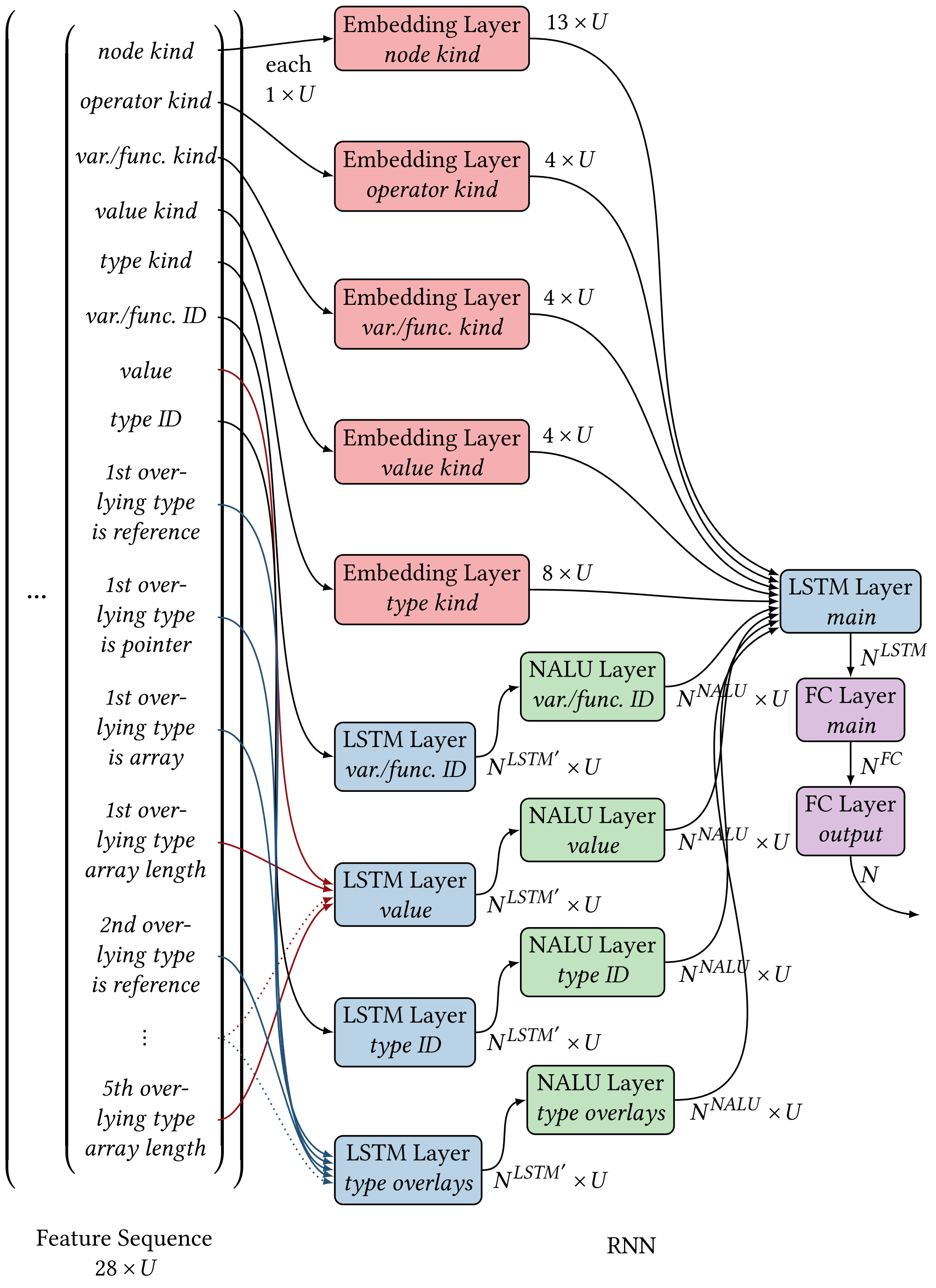


Figure 5.5: EmbFR's feature vector elements, RNN structure, and how both are connected. The data flows' labels denote the data's length in each dimension

Figure 5.5 visualizes EmbFR analogously to BasFR above. Each feature vector contains 28 elements and the RNN has 28 input neurons. In contrast to BasFR, we use nine input layers, which are parallel to each other. The outputs of all nine layers directly or indirectly flow to the RNN's main LSTM layer. From this point the data flow is the same as before except for the excluded main NALU layer. We exclude it since all value- or length-related features are processed by sub-NALU layer as described below. The NALU count HP's value is used for each of these sub-NALUs instead. With EmbFR, $\hat{F_{\underline{W}}} : \mathbb{R}^{28 \times U} \rightarrow \mathbb{R}^N$.

We use similar features as before with BasFR, but we do not feed all of them to the same single input layer. Instead, we classify the features based on whether they

- represent a kind (i.e., node kind, operator kind, variable/function kind, value kind, and type kind),
- represent an ID (i.e., variable/function ID and type ID), or
- represent a real or integer value (i.e., a literal's value and the type's array sizes), or
- represent a boolean-valued presence or absence (i.e., the type's pointer, reference, or array overlay).

Depending on this we create multiple input layers and connect the feature vector elements to them as described in the following.

- Instead of selecting feature vectors for the five different **kinds** manually as before, we represent each kind by an integer-valued feature whose value unambiguously represents the kind's value. For example, the node kind's feature takes the value 1 for an `if` statement and the value 2 for a call expression.

  For each of the five kinds, we create an embedding layer and feed the kind's feature to it. Each of these layers has a dynamic mapping that maps the integer kind features to feature vectors of a given length. Based on the training data, the training algorithm adjusts and improves the mapping with respect to the RNN's prediction power. For example, the mappings can be trained to map node kind feature values whose corresponding AST nodes have a similar behavior with respect to vulnerabilities to similar feature vectors and vice versa.

  We use these embeddings because they are promising for this thesis' goal and because they were successfully employed by Choi et al. and Russell et al. as well as in the somewhat similar field of Natural Language Processing (NLP) [Gho+16; SNS15].

We choose the node kind embedding layer to output feature vectors of length 13 since Russell et al. used the same value for their token kind embedding. They found this value to balance the expressiveness of the embedding against overfitting. We further initialized the embedding's mapping with our node kind representation that we created for BasFR. This way, the training algorithm does not need to start with a completely random mapping.

Similarly, the operator kind's and type kind's embedding layers' output lengths are four and eight, respectively. We use a random initialization for them for two reasons. Firstly, we do not have manually crafted representations that we could use. Secondly, the alternative of initializing them with one-hot encoded feature vectors for each possible feature value would result in feature vectors of length 43 and 47, respectively. Based on the above discussion we would expect these large vectors to lead to overfitting and therefore refrain from using them.

Since the variable or function kind and the value kind both have four possible integer values, we initialize their embedding layer's mappings with one-hot encoded feature vectors for each of these possible feature values.

- If a feature represents an **ID**, it is important to have a layer that is able to recognize a feature's specific value. Based on this, the layer can recognize a variable, function, or type, and it can output associated stored information to succeeding layers then. For example, recognizing a variable ID during the processing of a feature sequence is possible by subtracting the variable ID in the current feature vector from the memorized variable IDs from the previously seen feature vectors and look for a difference of zero.

  To support the RNN in storing the previous IDs as well as subtracting them, we create a chain of an LSTM and a NALU layer for each ID feature. Trask et al. also use such a combination in their experiments that require the RNN to store and calculate values [Tra+18]. Within this combination the LSTM layer does not only output an activation feature vector and the end of the feature sequence but outputs an activation after each processed incoming feature vector. The NALU layer performs its calculations on each of the LSTM's activations separately such that it also outputs a feature sequence. This is necessary since the main LSTM layer – and LSTM layers in general – expects a sequential input.

  While each NALU layer's neuron count N^{NALU} is a HP, we set its preceding LSTM layer's neuron count $N^{LSTM'}$ to be the half of the former. We assume the larger NALU count to be beneficial since the amount of different possible calculations on the stored numbers is higher than the amount of stored numbers.

- We do the same for features representing **values** or **presences/absences** since we assume the calculation on literals of different AST nodes to be advantageous. For example this allows to set an array declaration's type's array length off against an integer literal used in a for loop's condition.

In summary, BasFR and EmbFR incorporate more of each AST node's information than the feature representations from the literature. This allows our tool to detect more complex vulnerabilities in theory. The evaluation in chapter 7 investigates the influence of these representations in practice. BasFR and EmbFR differ in how much we directed the data flow by, e.g., connecting different types of features to different input layers. For the evaluation, we expect a difference between the two in terms of performance and resource consumption.

This chapter also answers RQ1's first sub-questions by describing the construction of various features from an SCS' AST, and by comparing the structure and data flow of two AST-based feature representations. We will answer RQ1's remaining sub-questions during our evaluation in section 7.2.1.

Chapter 6

Implementation

To use the theoretical foundations from the previous chapters they must be implemented in practice. For this, we developed a framework written in Python and C++. Sections 6.1 to 6.3 briefly describe the data preparation, training, and attribution components, respectively.

6.1 Data Preparation

We only describe the data preparation in case of the Juliet dataset since we consider the preparation of the MemNet dataset as a special case of it. In our implementation, data discovery and data representation are interlocked. During data discovery, we utilize ASTs to extract single SCSs from the test cases source code files. Afterwards, we directly forward the corresponding AST nodes to the data representation. Thus, each SCS only exists in form of AST nodes.

In summary, for all separate Juliet CWEs, our implementation generated 175 279 sequences of AST nodes. 61 627 of them represent a bad, vulnerable SCS and 113 652 a good, non-vulnerable one. Table 6.1 lists further aggregated statistics. For example, sequences consist of 89.40 AST nodes on average and are up to 1100 AST nodes long. The number of parsable test cases equals the count of bad sequences since each test case has exactly one primary bad function.

The sequences generated based on the MemNet dataset have an average length of 67.25 and thus are shorter than the ones generated from the Juliet dataset. Table 6.2 also shows that sequences generated from more complex test sets are on average longer than ones generated from less complex test sets.

The following sections describe the parsing of the test cases' source code files which was a challenging task for multiple reasons. Firstly, parsing the files written in C and C++ was not possible from within our Python framework. Therefore, we

Table 6.1: Aggregated data preparation statistics for separate CWEs of the Juliet dataset. Table 8.2 located in the appendix contains statistics for each separate CWE.

	Test Cases	**Prepared Sequences**						**Time**
	Count	**Count**			**Length**			**[s]**
	Parsable	**Total**	**Good**	**Bad**	**Min**	**Max**	**Mean**	
Sum	61 627	**175 279**	113 652	61 627				17 354
Min	1	**2**	1	1	3	4	4.00	0
Max	5922	**14 574**	10 440	5922	318	1100	338.61	1658
Mean	522	**1485**	963	522	45	242	89.40	147
SD	1039	**2992**	1993	1039	69	204	78.27	307

Table 6.2: Data preparation statistics of the MemNet dataset.

Part	**Prepared Sequences**						**Time**
	Count			**Length**			**[s]**
	Total	**Good**	**Bad**	**Min**	**Max**	**Mean**	
train	**10 000**	5014	4986	31	137	68.15	182
test level 1	**1000**	506	494	31	103	46.82	12
test level 2	**1000**	526	474	49	121	66.85	18
test level 3	**1000**	511	489	52	124	71.23	20
test level 4	**1000**	497	503	56	132	75.19	20
all	**14 000**	7054	6946	31	137	67.25	296

created a custom parser tool written in C++ as described in section 6.1.1. Secondly, many Juliet test cases expect to be parsed in a Microsoft Windows Environment. Since we worked on Ubuntu, this required some workarounds which we present in section 6.1.2. Finally, section 6.1.3 explain how we extract the SCSs' AST nodes from the test cases' ASTs.

6.1.1 Parsing C++

Each `.c` or `.cpp` file provided by data discovery corresponds to one translation unit and includes common test case support files. Some also include specific test case header files. An example for such a header file is `CWE124_Buffer_Underwrite__char_declare_cpy_81.h`. The result of parsing is an AST for each C and C++ source code file. An AST is a tree structure containing all information of the source code files it was parsed from. The succeeding steps operate on these AST and extract necessary information from the AST nodes which form the bad and good SCSs.

To integrate the parsing ability into our Python framework, a C and C++ parser for Python is required. To our best knowledge, there exists no parser written in Python which is capable of parsing C and C++ source code files. However, Python bindings[1] for LibClang[2], which is a stable, limited, high level C interface[3] to Clang, exist.

Clang is an open-source compiler frontend for C, C++ and other C-like languages. It is written in C++ and often used together with the LLVM compiler infrastructure as a backend. During parsing, the Clang compiler frontend builds an AST as described in section 5.2.1. The AST contains all information of the parsed source code. Clang also provides mechanisms to explore the AST and the connections between its nodes. For example, an AST node representing a function call is connected to the AST node representing the declaration of the called function.

Due to its Python bindings, LibClang would allow easy access to the Clang AST from inside our Python framework. However, it is limited in accessing all information associated with AST nodes. For example, AST nodes for integer literals arising from a macro expansion lack information about the literal's value. Such expansions exist in a large number in the Juliet dataset. Besides, LibClang does not expose the concrete operator kind of a binary operator AST node. This information is necessary for this thesis' feature representations as described in section 5.2. Other AST nodes, such as those for implicit casts, are not accessible at all. For the above reasons, we do not use LibClang for parsing.

An alternative is LibTooling, which is a C++ interface to the full Clang AST. As a drawback, it requires the creation of a standalone C++ application. Such applications built on top of LibTooling are called *Clang tools* according to the Clang documentation[4]. We use Clang 7.0.0 and C++14 for our clang tool.

To bridge the gap between our own Clang tool written in C++ and the rest of our framework written in Python, we utilize the JavaScript Object Notation (JSON). JSON is a standarized[5], language-independent data interchange format. JSON encoding and decoding is natively possible in Python using the Python Standard Library's json module[6]. In C++ we use the single file JSON implementation[7] by Niels Lohmann.

[1]https://github.com/llvm-mirror/clang/tree/master/bindings/python (retrieved on July 21, 2019)
[2]https://clang.llvm.org/doxygen/group__CINDEX.html (retrieved on July 21, 2019)
[3]https://clang.llvm.org/docs/Tooling.html#libclang (retrieved on July 21, 2019)
[4]https://clang.llvm.org/docs/Tooling.html#libtooling (retrieved on July 21, 2019)
[5]https://tools.ietf.org/html/rfc7159.html (retrieved on July 21, 2019)
[6]https://docs.python.org/3/library/json.html#module-json (retrieved on July 21, 2019)
[7]https://github.com/nlohmann/json (retrieved on July 21, 2019)

From the python framework's point of view, parsing with our Clang tool is done in four steps. Firstly, we write the collections of source code file paths to a temporary file using JSON. Secondly, we create another temporary file to which our clang tool will write its results. Thirdly, we run our clang tool in a sub-process passing both temporary files' paths as arguments. Fourthly, after our clang tool has finished, we read the resulting ASTs from the second temporary file into Python objects.

From our Clang tool's point of view, it starts with reading the source code file path collections from the JSON file specified by the first argument. Secondly, it parses each source code file using LibTooling. Thirdly, for each Clang AST we build a JSON tree structure with necessary information from the Clang AST. Lastly, the JSON tree structures are written to output file as specified by the second argument.

6.1.2 Treating SCSs' Environment Expectations

As shown in section 5.1, some Juliet and MemNet source code files need special treatment to successfully parse them. To exemplify, Juliet test cases with `w32` in their functional variant expect a Microsoft Windows operating system environment. The following paragraphs describe how we implement the special treatments.

According to the Juliet dataset's user guide, test cases having `w32` in their name will not be parsable on non-Windows operating systems [Nat12]. This is because these test cases utilize the Win32 API. The user guide also states that compilation of all test cases succeeds when triggered from the Microsoft Visual Studio 2010 command prompt on the Microsoft Windows 7 operating system. We further identified test cases having **`wchar_t`** in their name to be unparsable on Ubuntu in many cases. The parse errors are mainly about unviable candidate function because of missing conversions between **`wchar_t`**-typed arguments and the C++ standard library function's non-**`wchar_t`** parameters.

In summary, 22 287 of the 64 099 test cases – which corresponds to approximately 35 % – fail to parse without special treatment. Since omitting approximately 35 % of the Juliet dataset is not an option, we construct an artifical Windows environent using *Wine* and certain Clang flags. Wine is a compatibility layer that allows the execution of Windows application on Linux operating systems. We use the Wine C `windows` and `msvcrt` header files from the "libwine-development-dev" Ubuntu package[8]. These header files map Win32 API functions to appropriate POSIX function calls available on Linux.

[8]Version 1.9.6-1 from https://packages.ubuntu.com/xenial/libwine-development-dev (retrieved on July 21, 2019)

To parse a source code file which requires a Windows environment, we incrementally add Wine header directories and Windows related parse flags to Clang's include list and Clang's argument list, respectively. This must be done incrementally because some test cases require only a certain level of Windows environment. For example, some source code files require the Wine headers from the `windows` and `msvcrt` directories. Others will fail with these and only need the `windows` headers.

For time reasons, we did not further investigate the patterns which would perhaps explain each test cases requirements. Instead, we use a trial-and-error approach by trying to parse each source code file multiple times with increasing level of Wine headers and Windows related Clang flags. The following list contains the incremental levels of Windows environment. A given layer always entails its preceding levels.

Level 0 Baseline with no Windows specific includes or Clang flags

Level 1 Add Wine's `windows` directory to Clang's include list. Create a symbolic file system link from `windows/`**`W`**`inldap.h`, which is non-existent but included in testcases for CWE ID 90, to the existing `windows/`**`w`**`inldap.h`[9]. Set Clang's Microsoft compatibility mode version to Visual Studio 2010[10]. Define some `LDAP` and `LOGON32` macros[11] which are missing in the Wine headers but are necessary for parsing test cases for CWEs with IDs 90 and 256.

Level 2 Add Wine's `msvcrt` directory to Clang's include list.

Level 3 Deactivate Clang's builtin functions[12], which interfere with Wine-defined variables and functions in some test cases. Other test cases require the builtin functions.

Level 4 Define the `_WIN32` macro that is also defined by Microsoft compilers. This macro is used in the Juliet dataset's common header to switch between Linux and Windows includes. Due to this, Wine is used in more situations.

Level 5 Enable Clang's Microsoft extensions[13], which make Clang define **`wchar_t`** itself and which allows non-standard constructs used in Microsoft header files.

If a source code file cannot be parsed with any of the levels above, we give up and skip the file.

[9]This is a workaround for the Windows API being case-insensitive in regards to file system access.

[10]`-fms-compatibility-version=16.00`

[11]We set the macros `LDAP_PORT`, `LDAP_NO_LIMIT`, `LOGON32_PROVIDER_DEFAULT`, and `LOGON32_LOGON_NETWORK` with values according to https://github.com/delphij/openldap/blob/master/include/ldap.h and https://github.com/tpn/winsdk-10/blob/master/Include/10.0.10240.0/um/WinBase.h (retrieved on April 10, 2019).

[12]`-fno-builtin`

[13]`-fms-extensions`

In summary, our approach allows additional 19 815 test cases to be parsed and to be prepared further. Only 2472 test cases – which corresponds to approximately 4 % – remain unparsable.

6.1.3 Extraction of SCS from Test Case ASTs

As above mentioned, we extract the AST nodes belonging to a bad or good SCS from each test case's ASTs. This is done on a per-function level by identifying the good or bad functions and their callee functions in the ASTs.

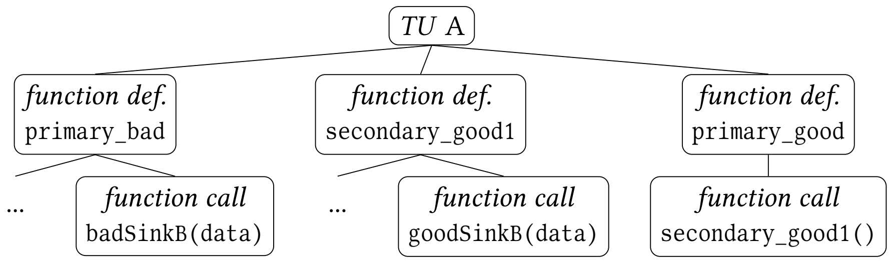
Firstly, for each test case we traverse its corresponding ASTs and build a list of all found function definitions. For example, consider a test case's ASTs as in figure 6.1 which is based on the Juliet test case `CWE124_Buffer_Underwrite__char_declare_cpy_52`. The following seven function definitions are present in the ASTs and are added to the list: `primary_bad`, `secondary_good1`, `primary_good`, `badSinkB`, `goodSinkB`, `badSinkC`, and `goodSinkC`.

Afterwards, we search bad and good functions in that list using the regular expressions given in the Juliet dataset's user guide [Nat12]. The Juliet dataset is designed in such a way that these regular expressions allow the unambiguous identification and distinction of primary bad, secondary bad, primary good, and secondary good functions. Section 5.1.1 describes and exemplifies primary and secondary functions in more detail.

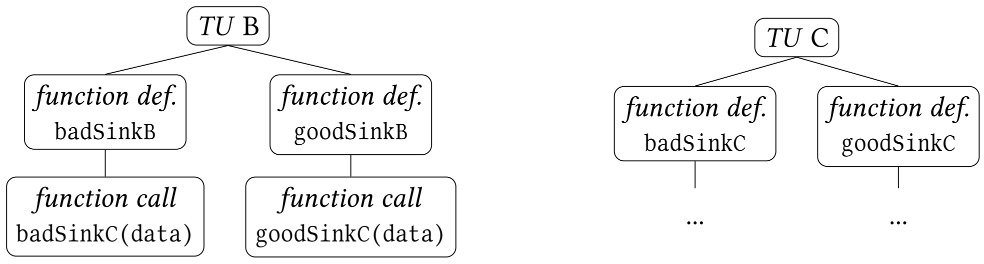
For extraction of AST nodes, we prefer secondary good functions over primary good functions because if secondary ones are present, the primary ones will only call the secondary ones. Otherwise, most good SCSs would start with a function call, which in turn would be an easy-to-learn criterion for distinguishing bad and good SCSs. We refer to the found secondary or primary functions as *entry function* since they are the entry point to a bad or good SCSs. As a result, there is one bad entry function and zero or more good entry functions for each test case. On average, a test case contains 1.54 good entry functions.

For the example above, we identify `primary_bad` as bad entry function and `secondary_good1` as good entry function.

The entry functions usually contain calls to other functions which results in multiple functions belonging to a bad or good SCS. To gather these, we build a simple call graph for each test case by recursively following call expressions present in the entry functions. Based on the call graph, we create a list for each entry function. Each such list contains the AST node representing the entry function's definition and the AST nodes representing the definitions of the entry function's direct and indirect callee functions.


TU A
function def. primary_bad
function def. secondary_good1
function def. primary_good
...
function call badSinkB(data)
...
function call goodSinkB(data)
function call secondary_good1()

(a) AST for source code file A.



(b) AST for source code file B. (c) AST for source code file C.

Figure 6.1: ASTs for an example test case with three associated source code files A, B, and C. AST nodes other than the ones representing function definitions or function calls are omitted.

For the call graph, we only support function calls for which Clang is able to determine the callee function definition. If we encounter a callee for which only a declaration is available, we ignore it. Calls on aliases, such as pointers or references, which would require a static alias analysis, are out of this thesis' scope. This is not a problem with our datasets in practice since Clang is almost always able to determine at least a callee's declaration.

Continuing the example from above, the AST nodes in the bad entry function's list represent the functions `primary_bad`, `badSinkB`, and `badSinkC`. The good entry function's list consists of AST nodes representing the functions `secondary_good1`, `goodSinkB`, `goodSinkC`.

For each list of function definition AST nodes, we construct a sequence of AST nodes that we will input to the feature representation later. We begin with an empty sequence and add AST nodes utilizing two nested for loops. The outer loop reversely iterates over the list of function definition AST nodes. For each function definition AST node, the inner loop traverses the function's sub-AST in depth-first pre-order and add the traversed AST nodes to the sequence. Due to iterating in reverse, in the resulting sequence a function definition's child AST nodes are in front of the AST node representing the call of that function. We expect this to be beneficial for the ANN since it has the chance to know what a call does when approaching it.

As a result, we have a set of sequences of AST nodes, where each sequence originates from either a bad or good SCS. Since its hard to tell from a sequence of AST nodes whether it represents bad, vulnerable SCS or a good, non-vulnerable one, we assign a label to each sequence. A set of pairs where each pair consists of a sequence of AST nodes and its label is the overall result.

To complete the example, the resulting set contains the two pairs

$$\big((\ldots, \texttt{badSinkC}\text{'s SAN}, \ldots, \texttt{badSinkB}\text{'s SAN}, \ldots, \texttt{primary\_bad}\text{'s SAN}, \ldots), \texttt{CWE124}\big) \text{ and}$$
$$\big((\ldots, \texttt{goodSinkC}\text{'s SAN}, \ldots, \texttt{goodSinkB}\text{'s SAN}, \ldots, \texttt{secondary\_good1}\text{'s SAN}, \ldots), \texttt{NV}\big)$$

where *SAN* denotes the function's sub-AST nodes in depth-first pre-order, and *NV* symbolizes that the sequence represents a non-vulnerable SCS.

6.2 Training

For conducting our experiments, we used TensorFlow in version 1.13.1. Achieving reproducibility of TensorFlow's outputs was impossible. We set all related seeds but still observed slight differences between consecutive runs. We explain this with concurrent calculations on the GPU not being deterministic.

To allow our tool to process SCS of arbitrary length, we utilized the TensorFlow Eager Execution mode. In this mode, the TensorFlow graph representing the RNN is not static but dynamically adjusts its shape to the inputs. That mode was rather new at the time of this thesis. As results, fundamental tasks such as saving and loading a model to and from disk were not easily available and required workarounds. Since this is out of this thesis' scientific scope, we refrain from further elaborations.

6.3 Attribution

We implemented the attribution techniques according to section 4.5. For the generation of PDF files containing attribution overlays, our implementation outputs each SCS into a minted[14] environment in a LaTeX file. Afterwards, it generates appropriate TikZ[15] commands for the rectangles and uses the tikzmark package[16] to align both. Finally, LuaLaTeX[17] generates the PDF file with the attribution overlays.

[14]https://ctan.org/pkg/minted?lang=en (retrieved on August 20, 2019)
[15]https://ctan.org/pkg/pgf?lang=en (retrieved on August 20, 2019)
[16]https://ctan.org/pkg/tikzmark?lang=en (retrieved on August 20, 2019)
[17]http://www.luatex.org/ (retrieved on August 20, 2019)

Chapter 7

Evaluation

The evaluation process' aim is to measure the quality of our tool. The first piece of our tool's quality is its power to correctly predict an unknown SCS's weakness or non-weakness. We also refer to this as *prediction power*. The second piece is our tool's power to explain the predicted weakness for a concrete SCS. We also call this *explanation power*.

We performed various experiments whose design and results allowed us to draw conclusions about our tool's quality. We mainly based the design process of these experiments on our RQs from section 1.1 and paid attention to ensure that the experiments' results were conclusive. Section 7.1 describes this process starting with the RQs and ending with a list of experiments which we conducted. Afterwards, section 7.2 contains the analysis of our conducted experiments' results. In addition to the usual figures, such as tables and plots, we also use attribution techniques with the goal to explain the results and their connections.

The whole evaluation process was partly iterative because some unexpected experiment results led to the demand for follow-up experiments. For example, the results of the extrapolation experiment on the ArrDeclAccess dataset were unexpectedly poor. Therefore, we performed another experiment with a simpler variant of that dataset to narrow down the cause.

We used three computing systems S1, S2, and S3 to conduct the experiments and kept the total available amount of computing resources in mind during the evaluation process. Table 7.1 lists the hardware information of the systems. On the software side, Ubuntu 16.04 LTS and CUDA 9.0 were installed on S1 and S2. S3 had Ubuntu 18.04 LTS and CUDA 10.0 installed. All systems used TensorFlow in version 1.13.1. We shared S1 with other researchers and S3 was only available during daytime. We even split related experiments over all systems depending on their availability.

At the end of the evaluation, section 7.3 discusses issues which threaten the validity of the design and analysis of our experiments.

Table 7.1: Hardware information for systems S1 to S3 which we used for performing the experiments.

System	CPU		RAM	GPU	
	Name	Clock		(Count) Name	Memory
S1	i7-7800X	3.5 GHz	32 GiB	(2) GTX 1080 Ti	12 GiB
S2	i5-7500	3.4 GHz	16 GiB	(1) GTX 1050 Ti	4 GiB
S3	i5-6600K	3.5 GHz	32 GiB	(1) GTX 1060	6 GiB

7.1 Experiment Design

The goal of the design process was that the resulting experiments allow us to conclude about our tool's quality. We divided the process of designing the experiments into three steps.

In the first step we deduced requirements for the experiments from our RQs. This mainly boiled down to the identification of HPs whose values needed to be varied over multiple experiments to draw conclusions. Since not all the RNN's and the training process' HPs were covered by the RQs, we identified other HPs based on values used in the literature and on this thesis' resource constraints. Section 7.1.1 describes the whole identification.

We obtained many HPs and many possible HP values from the first step. Since our resources were limited and we wanted to focus on the variation of essential HPs, we decided to fix some of the non-essential HP's values. Section 7.1.2 explains which HPs we considered essential and how we fixed the non-essential ones' values.

Based on the remaining essential HPs and our resource constraints, section 7.1.3 establishes the experiments that we actually conducted. This section also describes the common protocols for conducting each experiment.

7.1.1 Identification of Requirements and HPs

Table 7.2's three segments reflect where we derived the containing HPs from and how relevant they are for this thesis.

We identified the first two segments' HPs based on our RQs. We separated them based on their relevance for this thesis. The second segment's HPs are only based on a sub-RQ within RQ3 and are neither directly related to feature representations, LSTMs, NALUs, explainability, nor datasets used in other literature. We therefore identified

Table 7.2: HPs which we identified during experiment design. We identified the ones in the first two segments based on our RQs, while we only consider the italicized ones in the first segment to be essential for this thesis. We treat all others as non-essential and preliminarily fixed them to the bold values.

RQ(s)	HP	Values to Be Evaluated
RQ1	*feature representation*	BasFR, EmbFR
RQ2	*LSTM neuron count*	4, 8, 24, 32
RQ3	*NALU count*	none, 4, 8, 24, 32
RQ4	*attribution technique*	GradInput, IntGrad20, IntGrad100, IntGrad300
RQ4	*prediction correctness*	TP, TN, FP, FN
RQ3, RQ5	*dataset & dataset split*	Juliet separate CWEs & random, Juliet merged CWEs & random, MemNet & MNPaper, ArrDeclAccess & extrapolation
RQ1, RQ4, RQ5	metrics	**F_1 & epoch count**, **time consumption**
RQ2	FC neuron count	**4**, 32
RQ2	activation function	**Sigmoid**, ReLU
RQ2	training algorithm	**Adam**, RMSprop
RQ2	learning rate	**0.001**, 0.1
	min. weakness size	**7**
	repetition count	**3**
	batch size	**128**
	loss function	**cross entropy**
	weight initialization	**glorot uniform**
	max. epoch count	**1000** (restricted by early stopping)

them as of minor priority and refer to them as *non-essential* HPs. Correspondingly, we call the italicized ones in the first segment *essential*.

The table's third segment contains other HPs of RNNs or their training process. These were not covered by the RQs but we still had to fix their values for conduction our training.

RQ-based Identification of HPs The following describes how we went through the RQs from section 1.1 with the goal to derive desired conclusions and to derive HPs which needed to be varied to allow these conclusions.

- **RQ1** was about the crafting and comparison of different feature representations. Section 5.2 worked out the two feature representations BasFR and EmbFR. To allow comparison, we identified the need of multiple trainings where only the HP *feature representation* differed. This corresponds to table 7.2's first row.

  To compare the prediction power performance and resource consumption, we identified the HP *metrics*. We use F_1 to compare both feature presentations' prediction power performances since that metric is also used in the literature as further described in the context of RQ5 below. To compare BasFR's and EmbFR's resource consumption, we use the epoch count after which the training algorithm yielded the best tool configuration. We do not use the elapsed time since it depends on the respective system's hardware and load. As a result, F_1 & epoch count is a unified value for the HP metrics. We unify both metrics as we do not vary them over different experiments.

- **RQ2**'s and **RQ3**'s topics were the LSTM technique, NALUs, and a general variation of an ANN's HPs. The following paragraphs look at all three topics.

  Since an ANN with at least one recurrent layer is mandatory for processing sequential data, an LSTM layer was present in each experiment's ANN. Recurrent layers other than LSTM layers were out of this thesis' scope. To get a feeling about the influence of the LSTM layer's neuron count, we identified the HP *LSTM neuron count*.

  Neuron counts in general control an ANN's shape and thus the complexity of the function realized by it. On the one hand, too small values yield an ANN too simple to adapt to complex SCSs. On the other hand, larger values result in a bigger, more complex ANN containing more weights and performing more calculations. As all these weights need to be adjusted during training, a large neuron count delays the training algorithm's convergence. Furthermore, the more complex an ANN becomes, the more it tends to overfit on the given dataset. Section 4.2 describes these phenomena in more detail.

  In the literature, the neuron count highly varies between 4 and 400 [Rus+18; ZS14] depending on the layer kind, the layer position, and the application context. Having in mind that the neuron count should be as small as possible [Hag+14], we defined the set of neuron counts as 4, 8, 24, 32.

  Analogously, we identified the HP *NALU count*.

  RQ3 is also about the NALU layers' influence on extrapolation to the range outside the numbers encountered during training. For evaluation of this influence, we identified the presence of NALU layers as another HP to be varied. In contrast to LSTM layers, NALU layers can be omitted without losing the ability to operate on sequential inputs. To avoid senseless combinations of

HP like an absent NALU layer with 24 neurons, we merged the HP for NALU layer presence and NALU count. As a result, none, 4, 8, 24, 32 were the values for the NALU neuron counts with none indicating the absence of NALU layers.

For evaluation of the NALUs extrapolation ability, we also employ our own ArrDeclAccess dataset with the extrapolation split. As a result, ArrDeclAccess & extrapolation is one of the HP *dataset & dataset split*'s values.

For RQ3's general variation of HPs, we selected four HPs and varied their values in accordance with a grid search to get a feeling about their influence on prediction power.

These selected four HPs were the neuron count and the activation function of the RNN's main FC layer, the training algorithm, and the learning rate used by the training algorithm. We selected these specific HP based on our previous experience with ANNs.

Since we already motivated the evaluation of the neuron counts, we briefly describe the other HPs and also mention why we expected them to have a high potential to affect prediction power. The activation function impacts each neuron's behavior as described in section 4.1. For example, the ReLU function mitigates the possible problem of vanishing gradients in deep ANNs. The training algorithm and its associated learning rate govern the training of an ANN. In more detail, they control the direction and the size of each gradient descent step. Taking good steps is crucial for adequately exploring the loss function's minima by not only descending appropriately into a minimum, but by also escaping from local minima.

In summary, we identified seven HPs based on RQ2 and RQ3. Table 7.2 lists them in rows two, three, and six as well as in the second segment.

- **RQ4** is related to the evaluation of explanation power using attribution techniques. Section 4.5 introduced two such techniques: GradInput and IntGrad. For comparison of both, we identified the HP *attribution technique* and its values GradInput and IntGrad.

  The instructions for calculating GradInput attributions are self-contained. However, with IntGrad a HP must be specified that controls how many steps are done between the baseline and the given input. Empirically good values are approximately in the range of 20 to 300 [STY17]. Based on this we chose 20, 100, and 300 as possible numbers of steps. Since the number of steps does not make sense for GradInput, we merged the number of steps into the attribution technique HP. Its resulting values are GradInput, IntGrad20, IntGrad100, and IntGrad300 with the numbers specifying the steps.

Attribution techniques should support developers in understanding predictions of our tool. We expected that how our tool explains a prediction depends on whether it predicted a vulnerability or not and whether the prediction is correct. This resulted in the HP *prediction correctness* and its values TP, TN, FP, and FN. We motivate these values in the following paragraphs.

If the tool predicts a vulnerability in a given SCS, we will expect it to highlight the vulnerable AST nodes. For example, we expect it to highlight a declaration of an array and an out-of-bounds access to the same array. If that prediction is wrong, i.e., a false positive, we will expect the attribution technique to indicate that the ANN assumed wrong connections between the AST nodes. For example, we expect it to highlight an array's declaration and an allegedly out-of-bounds access to a different, bigger, and unrelated array.

For a false negative, we expected that the attribution technique does not highlight all AST nodes that are related to the vulnerability. We did not have expectations for the attribution's results in case of a true negative.

Another requirement is to access an attribution technique's resource consumption. Thus, we measure the time necessary to calculate the attributions. This is reflected in the HP metrics and its value time consumption.

- Lastly, **RQ5** is about comparing this thesis' prediction power with the prediction powers presented in the literature. To properly compare, the used dataset, its split, and the used metrics must all match between our experiments and the experiments from the literature. For this we identified the already existing HPs dataset & dataset split and metrics.

  We compared our results with the ones by Choi et al., Goseva-Popstojanova and Perhinschi, and Russell et al. in their respective papers [Cho+17; GP15; Rus+18]. Since we wanted to compare our best configuration, we set the other HPs' values to the combination of values which performed best in the preceding experiments.

  Choi et al. and Russell et al. provided F_1 and AuROC scores among others. Choi et al. and Goseva-Popstojanova and Perhinschi give accuracy scores among others. We therefore selected the F_1 as our primary metric, but also calculated accuracy scores for comparison with Goseva-Popstojanova and Perhinschi. We favored F_1 over AuROC and accuracy since the latter often overestimate the actual prediction power, as described in section 4.2.6.

  The three papers perform benchmarks on either the Juliet or the MemNet datasets. In case of MemNet, the whole dataset and the MNPaper split based on the one in Choi et al.'s paper is used. In case of Juliet, the situation is more complex since the different papers used different CWEs. The following paragraphs give more details and describe which CWEs and which split we identified for our experiments.

Goseva-Popstojanova and Perhinschi considered 22 Juliet CWEs for their benchmark. Unfortunately, we excluded three of these CWEs since their SCS count is too small. We further describe this in the context of other HPs below. As a consequence, we used the group of the remaining 19 CWEs as a merged dataset. In addition, for each of the 19 CWEs, we considered their SCSs as a separate dataset in our experiments. Since the three static analyzers examined in Goseva-Popstojanova and Perhinschi's benchmark are conventional ones, no dataset split is necessary nor given there. We therefore use our random split which we described in section 5.1.1.

Russell et al. did not only utilize the Juliet dataset but also a dataset containing CWEs not present in Juliet [Rus+18]. They do not specify which CWEs they considered but since their merged dataset contains 149 different weaknesses, we assumed them to use all Juliet 118 CWEs, among others. After their data curation, they only used 11 896 SCSs from the Juliet dataset. This is less than the 175 279 SCSs we extracted from all 112 CWEs that have more than six SCS per weakness. Again, the paper does not contain information which exact SCSs were used. For comparison with them, we decided that using all 112 CWEs both as separate datasets and as one big merged dataset is the closest and most conservative approach. Russell et al. randomly split their dataset and did not provide information about their specific split. As a result, we use our random split.

In summary, for the HP dataset & dataset split, we identified the additional values MemNet & MNPaper, Juliet paper CWEs merged & random, Juliet all CWEs merged & random, Juliet paper CWE <ID> separate & random, and Juliet CWE <ID> separate & random. The last two contain <ID> as placeholder for the IDs of the 19 CWEs utilized in Goseva-Popstojanova and Perhinschi's paper and for the IDs of all 112 CWEs, respectively.

In summary, we identified eleven HPs from our RQs which are listed in the first two segments of table 7.2. The italicized ones are essential.

Other Non-essential HPs Besides the HPs from the RQs we had six other non-essential HPs which had to have their values fixed for the implementation to work. These are listed in table 7.2's third segment. The following introduces them and shows how we fixed their values based on values used in the literature and on this thesis' resource constraints.

- During the data preparation, we ensure a **minimum amount of feature sequences per weakness**. We chose seven for this since it is the smallest amount for which the splitting of feature sequences with the random split does not result in either an empty validation or test set. As a consequence for the Juliet dataset, we completely ignore the six weaknesses with the CWE IDs 835, 562, 674, 561, 500, and 440.

- We set the **repetitions** of each experiment's training process to three. As described in section 6.2, we found this necessary since the training of an RNN using TensorFlow — even with fixed seeds — was a stochastic process. Repeating each experiment's training allows us to calculate mean and standard deviation (SD) over the repetitions' scores. Three is our trade-off between maintaining this thesis' resource constraints and the statistics' expressiveness since it is the smallest number for which the calculation of the standard deviation makes reasonable sense.

  As a consequence for the experiment results, we also provide three separate validation F_1s as well as their mean and SD. We planned to do so too for the test F_1s, but due to a bug in our implementation, a repetition's best tool configuration's test F_1 was only computed if the tool configuration's validation F_1 was higher than the previous repetitions' best tool configurations' validation F_1s. Unfortunately, we did not have enough time to rerun all experiments. As a result, the second and third repetitions' best tool configurations' test F_1s are often missing. We still provide test statistics if available.

- Choosing the **batch size** was a trade-off between generalization, training speed, and memory consumption [GBC16, pp. 274-277]. One the one hand, a smaller batch size often results in better generalization due to noisier gradients as discussed in section 4.2. On the other hand, a larger batch size yielded shorter epochs as it allowed for better parallelism on the used multicore systems. This is especially beneficial for batch sizes that are a power of two [GBC16, p. 276]. Additionally, fewer gradient updates are done per epoch with larger batch sizes.

  With our implementation, large batch sizes of 256 and upwards led to out-of-memory errors for some Juliet CWE datasets with long feature sequences. These errors occurred in particular on system S2 since it had the smallest GPU memory. Eventually, we fixed the batch size to the value 128. We chose this value since it still resulted in noisy gradients for most datasets, since it is a power of two, since it worked well on all systems, and since Russell et al. successfully used that value in their paper [Rus+18].

- We employed cross entropy as **loss function** since it is the common choice for multi-class classification as explained in section 4.2.

- For the RNN's **weight initialization**, we used the glorot uniform method because it is the TensorFlow default value. That initialization is described in section 4.2.
- We allow each training to last 1000 **epochs in maximum**. We expected about 200 epochs to be sufficient based on our previous experience but chose the bigger value to leave the training algorithm time to thoroughly explore the loss function. To avoid overfitting on the training data and to prevent the training from continuing without doing regular progress, we employ early stopping, as presented in section 4.2.5.
- TensorFlow and RNNs in general have various other HPs. We left all HPs not mentioned in this thesis at their respective default values.

The bold values in the third segment of table 7.2 summarize the resulting fixed values.

7.1.2 Reduction of Non-fixed HPs

From the first step, we obtained ten non-fixed HPs and many possible values for each of them. Since our resources were limited and since we wanted to focus on the variation of the six essential HPs, we decided to fix the values of the four non-essential HP that are present in the second segment of table 7.2. This way, we obtained a partial baseline for the upcoming experiments and reduced the number of HP configurations to evaluate.

Instead of arbitrary fixing the HPs, we conducted pre-experiments. For this, we chose two possible values for each of the four HPs. Afterwards, based on grid search, we conducted 16 pre-experiments for all 16 resulting configurations on the MemNet dataset. We utilized a random split, since we performed the pre-experiments at an early stage when we had not implemented the MNPaper split yet. For the other HPs, we used EmbFR, 32 LSTM neurons, and 32 NALUs. We then compared the resulting validation F_1s to select the best combination of HP values for the upcoming main experiments.

We did not consider the test set here since it should not be used for selecting a HP configuration for further experiments. Instead, we employ the test set for our final tool configurations to assess their actual performance on the unseen data. Conducting experiments for all combinations had the goal of not only identify relationships between the HPs and the F_1s on the validation set, but to also identify relationships between the HPs themselves.

The restriction to two values was based on the knowledge that one epoch took approximately 120 seconds to complete. As a result, all pre-experiments completed after approximately 478 hours which corresponds to approximately 20 days. This exemplifies the grid search's already mentioned exponential time consumption, which is one of its drawbacks.

The following briefly explains how we chose two concrete values for each HP based on the literature and on prior knowledge.

- The **training algorithm**'s selected values are Adam and RMSprop since they are among the most commonly chosen training algorithms for RNNs in the literature [Rus+18; Cho+17; Dam+17; PS18].
- We selected the sigmoid **activation function** since it is similar to the behavior in the human brain as shown in section 4.1. We evaluate ReLU since a much better performance with it would indicate that our ANN is deep and suffers from a vanishing gradient problem. Section 4.4 briefly describes this problem.
- For the **learning rate** we found many values between 0.0001 and 0.5 to be successful in the literature [Rus+18; Cho+17; ZS14]. We chose 0.001 and 0.1 since the former is the TensorFlow default value and the latter since we expected a bigger value allowing better exploration of the error function. If three values had been allowed, a smaller value would also have been of interest.
- The **FC layer's neuron counts** 4 and 32 are arbitrarily values from the set of neuron counts established in the previous section.

For each combination of HP values, we visualize the resulting F_1 in figure 7.1. Table 8.3 in the appendix contains numeric and repetition-wise results. For the visualization, we projected the four-dimensional space spanned by the four HPs row-wise onto the *Y* dimension of a two-dimensional space. The *X* axis represents the F_1 dimension. For each combination of HP values, a vertical black line at the corresponding F_1 connects its HP values.

We reached the best F_1s, which are around 90 %, with all tool configurations except for a learning rate of 0.1. We assume this learning rate to be too large to allow a successful descent into local minima as visualized in chapter 4's figure 4.5. As a consequence, we fixed the learning rate to 0.001.

Clear relationships between the HPs could neither be deduced from the figure nor from the table. However, for the activation function and the training algorithm, the configurations with sigmoid and Adam slightly outperformed the other configurations. The FC layer's neuron counts 4 and 32 were balanced, so we used the smaller one to keep the RNN's complexity low. The markings in bold in the second segment of table 7.2 summarize the resulting fixed values.

Based on the pre-experiments, we created a partial HP baseline as follows. For the upcoming experiments, we utilize FC layers consisting of 4 neurons each and using

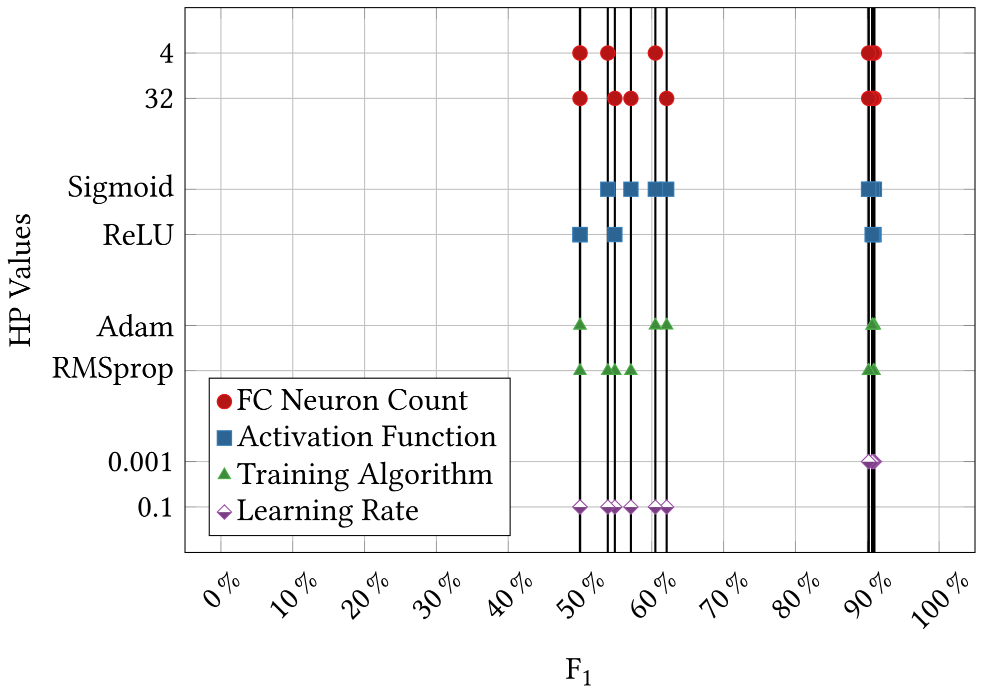


Figure 7.1: Visualization of the results from table 8.3. We projected the four-dimensional space spanned by the four HPs row-wise onto the Y dimension of a two-dimensional space. The X axis represents the F_1 dimension. For each combination of HP values, a vertical black line at the corresponding F_1 connects its HP values. The fact that the combinations around 90 % $F_1$are hard to tell apart reflects that there is no clear best combination.

the sigmoid activation function. Furthermore, we used the Adam training algorithm with a learning rate of 0.001.

The assumption that the results of the pre-experiments could be transferred to the other experiments is not ideal. For example, BasFR and EmbFR are different in their shape, behavior, and in the resulting RNN's number of weights. Thus, the influence of the training algorithm may by different when using BasFR instead. In both cases, we expect the big learning rate of 0.1 to be disadvantageous. We expected the FC layer's neuron count of 4 to be too small for training with multiple weaknesses. We therefore repeated one of the merged Juliet dataset experiments with the FC layer consisting of 32 neurons but did no not observe a significant change in F_1 during the analysis of the results.

Besides to the example-based reasons just given, we kept the assumption since doing not so would have resulted in additional pre-experiments for each upcoming experiment. This in turn would have exceeded this thesis' primary scope and resource constraints. Hence, the only alternative would have been to fix the values without experiments based solely on the literature and on prior knowledge. We prefer our approach, which is also based on the literature and on prior knowledge, but which also adds information from pre-experiments.

In summary, we fixed the values of four HPs, which are of minor importance for this thesis' central goals. As a result, during our experiments, we only needed to examine the essential HPs. This allowed a more compact and clearer analysis and presentation of the results. Another advantage arises from the fixed values establishing a baseline and thus simplifying the comparability of the experiments.

7.1.3 Conducted Experiments

This section describes the design of the experiments that we conducted within this thesis constraints. We paid attention to ensure that the experiments' results were conclusive. For example, we ensured that only the values of one HP differed when evaluating that HP's effect in terms of prediction power. In addition, this section describes the two protocols for conducting prediction power and explanation power experiments, respectively. As a result, each experiment is fully described by its protocol and its associated HP values. Since the non-essential HPs were fixed, we do not further mention them.

Prediction Power Table 7.3 lists the 239 experiments in the context of prediction power based on the identified HPs. The following paragraphs describe the origin of that table.

In section 7.1.1, we selected the **Juliet dataset** among others for the evaluation of feature representations and for comparison with the literature. For the former we wanted to evaluate both feature representations on multiple datasets. For the latter, we wanted to perform experiments with all CWEs and with the subset of CWEs specified in Goseva-Popstojanova and Perhinschi's paper. In both cases, we wanted separate experiments for each CWE and experiments on the merged datasets containing SCS of multiple CWEs.

We combine the experiments on separate CWEs into experiments E1 to E112 and E113 to E224. Each of these $2 \cdot 112 = 224$ experiments is one combination of a separate CWE and a feature representation. These also cover the experiments for the separate CWEs given in the paper (except for the excluded too small CWEs).

For the merged dataset containing SCS of all 112 Juliet CWEs, we were unable to fruitfully conduct an experiment. Reasons for this are on the one hand the time consumption of over two hours per epoch, which – assuming 100 epochs on average – corresponds to approximately four weeks per training process. On the

Table 7.3: Conducted experiments for prediction power. The first rows denote multiple experiments which differ in the utilized CWE.

Experiment	Dataset				Feature	NALU
ID(s)	Base	CWEs	Comb.	Split	Repr.	
E1 – E112	Juliet	*each*	separate	random	BasFR	yes
E113 – E224	Juliet	*each*	separate	random	EmbFR	yes
E225 – E230	Juliet	*six*	separate	random	EmbFR	no
E231	Juliet	paper [GP15]	merged	random	EmbFR	yes
E232[1]	Juliet	paper [GP15]	merged	random	EmbFR	yes
E233	Juliet	taxonomy	merged	random	EmbFR	yes
E234	MemNet			MNPaper	BasFR	yes
E235	MemNet			MNPaper	EmbFR	yes
E236	MemNet			MNPaper	BasFR	no
E237	MemNet			MNPaper	EmbFR	no
E238	ArrDeclAccess			extrapol.	BasFR	yes
E239	ArrDeclAccess			extrapol.	BasFR	no

other hand, during our first attempts to do so, too large literal values in few of the source code snippets yielded invalid gradients which prevented any training progress even after many epochs. After fixing the problems, we decided to focus on a smaller, merged dataset first. For this, we chose the merged dataset containing SCS of the 19 CWEs given in Goseva-Popstojanova and Perhinschi's paper. E231 is the resulting experiment, which utilized EmbFR and completed without further problems. Unfortunately, we did not have time to repeat the same with BasFR.

To avoid conducting only one experiment using a merged dataset, we looked for another subset of CWEs. Instead of arbitrary choosing such a subset, we employ a CWE taxonomy that puts all CWEs into a hierarchy. There are three taxonomies given on the CWE website[2], which also contain CWEs that are not covered by the Juliet dataset. They group the CWEs by research, development, and architectural concepts, respectively. We used the former since its grouping correspond to how vulnerabilities can be detected. For example, one group contains CWEs that are related to accessing a memory buffer outside its bounds. Such vulnerabilities can be detected by verifying the declared bounds during accesses.

From the eleven top-level groups of that taxonomy we found *Incorrect Access of Indexable Resource (Range Error)* most interesting. This group contains ten Juliet

[1]Same as E231 but with more weight for low FP count. See section 7.2.4 for more information.
[2]https://cwe.mitre.org/data/definitions/1000.html (retrieved on June 12, 2019)

CWEs which contain SCS of overflow weaknesses. We merged them and conducted experiment E233 with the resulting dataset. Analogously to above, we only used EmbFR for this.

As with the Juliet dataset, we selected the use of the **MemNet dataset** for the evaluation of feature representations and for comparison with the literature. In contrast, the MemNet dataset only contains SCSs of one weakness such that there is no distinction between separate and merged datasets. As a result, we conducted one experiment for each feature representation resulting in experiments E234 and E235.

For the evaluation of **NALUs**, we wanted to evaluate both the influence of NALUs being present and the influence of the NALU count. However, due to the unexpectedly tedious extrapolation-related experiments that we describe in section 7.2.2, we decided to only evaluate the presence and absence to meet this thesis time constraints.

For the Juliet dataset, we repeated six of the separate CWE experiments without NALUs. We arbitrarily selected the subset of CWEs 242, 367, 457, 476, 482, and 563 from the CWEs specified by Goseva-Popstojanova and Perhinschi's paper. This resulted in experiments E225 to E230.

For the MemNet dataset, we repeated both feature representation experiments without NALUs resulting in the experiments E236 and E237.

To specifically evaluate the extrapolation abilities of NALUs, we use our ArrDeclAccess dataset and BasFR. We chose the latter to keep the ANN's complexity low since the ArrDeclAccess dataset has almost no variation. We vary the presence of NALUs over experiments E238 and E239.

For the **LSTM** count, we were again too time constrained to examine it.

As a result, we extended our HP baseline by 24 LSTM neurons and eight NALUs (if present). We chose these specific values since they were our initial and intuitive values when we started our first experiments with BasFR. The reason for this is that these values result in the ANN's layers' neuron counts decreasing with layers being further back: 24 LSTM, 8 NALU, 4 FC, and 2 FC softmax neurons. This funnel shape forces the ANN to represent the input with gradually lesser weights and neurons similar to an autoencoder's encoding part [GBC16, p. 499ff.].

For a discussion of **false positives**, we also repeated E231 and put more weight on the non-weakness class. E232 is the resulting experiment.

Each prediction power experiment was conducted following the same protocol. The protocol's three steps correspond to the three components of our tool's training and testing pipeline introduced in chapter 2.

1. Prepare the associated dataset by using the associated feature representation and dataset split.

2. Perform a training of the RNN that was constructed by the associated feature representation and the associated NALU presence. Note that the feature representation has influence on both the data preparation and the construction of the RNN.

3. Use the associated metrics to compute the test scores after the training process. For this, we always utilize the same dataset as during training. Testing on another dataset than the one used during training is subject of future work.

The scores are each experiment's result, and we use them for discussion and comparison in section 7.2 later.

Explanation Power We present ten experiments for the evaluation of our tool's explanation power in table 7.4 and describe them below.

We vary the attribution techniques during the four experiments E240 to E243 in the table's first segment. For these, we used our best tool configuration on the Juliet taxonmy CWE group, which employs EmbFR. We utilize the SCS that we used in section 5.1.1 to exemplify the Juliet dataset as input. Our tool configuration correctly predicted that SCS as vulnerable.

For the other experiment described in the following, we employed IntGrad100 since we did not observe much difference between it and the three times slower IntGrad300.

For the evaluation of the feature reprensentations' influence on the attribution overlays, we conducted the four experiments E244 to E247. For the first two, we input a TP and a TN SCS to the best BasFR tool configuration. For the last two, we input the same SCS to our best MemNet tool configuration, which utilizes EmbFR.

Since we also identified FP and FN SCSs as evaluation subject, we conducted the experiments E248 and E249 for them, respectively.

Next to assessing the attribution overlays, we also measured the time consumption of their generation process.

The two steps in the protocol of the explanation power experiments correspond to our tool's classification & attribution pipeline as in chapter 2.

1. Predict the weakness of the given SCS by feeding its feature sequence to the given tool configuration's RNN model. Again, we always utilize SCS from the dataset the model was trained on. Predicting SCS of another dataset than the one used during training is subject of future work.

2. Construct an attribution overlay as specified by the attribution technique HP. Measure the time necessary to construct these.

Table 7.4: Conducted experiments for explanation power.

Experiment ID(s)	SCS Dataset	SCS Prediction	Feature Repr.	Attribution Technique
E240	Juliet	TP	EmbFR	GradInput
E241	Juliet	TP	EmbFR	IntGrad20
E242	Juliet	TP	EmbFR	IntGrad100
E243	Juliet	TP	EmbFR	IntGrad300
E244	MemNet	TP	BasFR	IntGrad100
E245	MemNet	TN	BasFR	IntGrad100
E246	MemNet	TP	EmbFR	IntGrad100
E247	MemNet	TN	EmbFR	IntGrad100
E248	MemNet	FP	EmbFR	IntGrad100
E249	MemNet	FN	EmbFR	IntGrad100

The prediction, the attribution overlay, and the time consumption are each experiment's result, and we use them for discussion and comparison in the following section 7.2.

As an overall result, the bold values from table 7.2, together with 24 LSTM neurons and 8 NALUs, formed the HP baseline that we used for all conducted experiments unless otherwise noted.

7.2 Analysis of Experiment Results

This section describes the analysis of our experiments' results. As one part of that analysis, we draw conclusions about the prediction and explainbility power and scientifically assess them. Based on this, we answer our RQs in this section.

The first section 7.2.1 covers experiments related to the feature representation. The subsequent sections 7.2.2 and 7.2.3 analyse the experiments with respect to NALUs and comparison with the literature, respectively. Since we only assess F_1 (and accuracy) in these sections, section 7.2.4 also discusses the false positive rate and introduces two approaches for reducing it. The just-mentioned sections already use attribution overlays to explain their results. Section 7.2.5 further evaluate different attribution techniques and their time consumption.

We usually provide validation and test *best*, *mean*, and *standard deviation* (SD) F_1s as follows. In case of validation F_1s, all three F_1s are the highest, average, and SD F_1 of the three training repetitions, respectively. The best validation F_1 is the highest

F_1 achieved on the validation set in any of the three repetitions. For mean and SD, we consider each repetition's highest validation F_1 and apply mean and SD to the resulting three F_1s.

For test F_1s, the test best F_1 is *not* the highest test F_1 achieved during the three repetitions. Instead, it is the test F_1 of the tool configuration with the best validation F_1. The test mean and test SD are defined analogously to their validation counterparts above. As a consequence, a test mean F_1 may be greater than the corresponding test best F_1.

In figures and tables, a stand-alone *Validation* F_1 or *Test* F_1 refers to the best validation or to the best test F_1, respectively.

7.2.1 Feature Representations

We conducted multiple experiments with respect to the feature representation.

First, we considered the $2 \cdot 112 = 224$ experiments E1 to E112 and E113 to E224, whose instructions were to train an RNN with BasFR and EmbFR, respectively, on each separate Juliet CWE. As mentioned above, we excluded six weaknesses from the overall of 118 CWEs since their amount of SCSs is too small.

Second, we merged multiple separate Juliet CWEs and trained RNNs with these multi-CWE datasets. The corresponding experiments are E231 and E233.

Third, we compared the two feature representations based on the MemNet dataset by examining the experiments E234 and E235. For the analysis on the MemNet dataset, we also use the attribution overlays from some explanation power experiments.

In the following first paragraph, we mainly examine the experiments' resulting F_1s. The second paragraph contains a discussion about the epoch counts after which the training algorithm yielded the tool configuration. These counts give information about the training processes' consumed resources.

F_1 Table 8.4 in the appendix lists validation and test F_1s for each separate Juliet CWE and for each feature representations. To visualize the table, figure 7.2 plots each CWE's best test F_1 against the validation best test F_1.

The average CWE-wise best **test** F_1s for BasFR and EmbFR, which are marked by a larger, unfilled red dot and blue square are 82.99 % and 46.45 %, respectively.

In other words, with BasFR the CWE-wise best tool configurations are capable of correctly predicting 82.99 % of unseen SCS on average. BasFR's lowest CWE-wise best test F_1 of 25.71 % is for CWE 780 (Use of RSA Algorithm Without OAEP). With

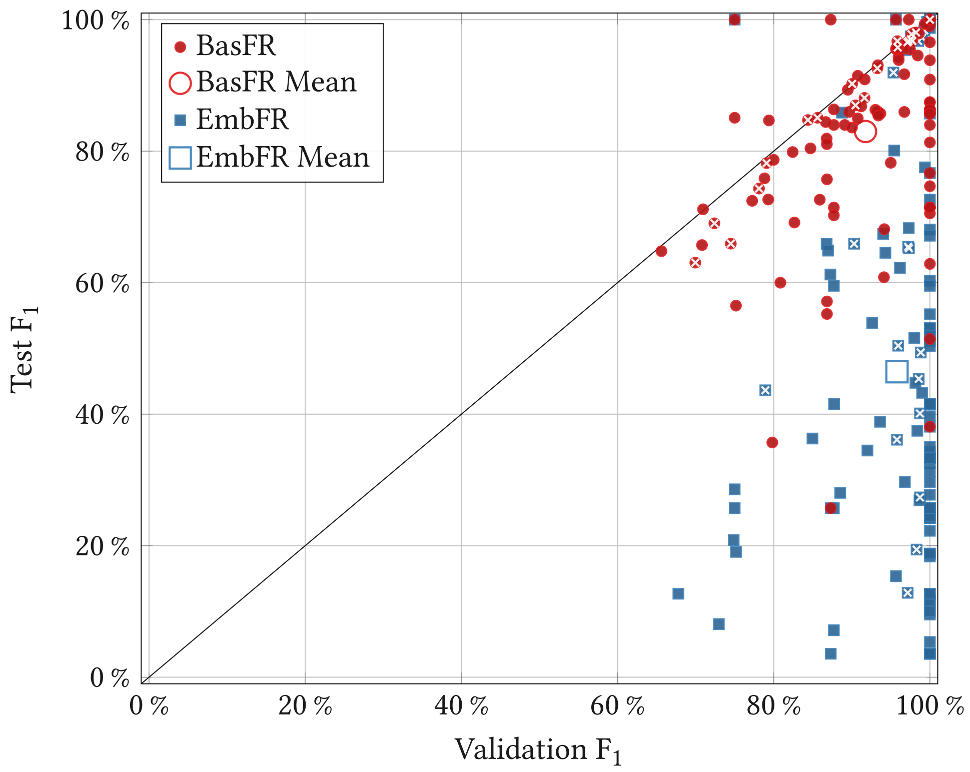


Figure 7.2: Visualization of the results of E1 to E112 and E113 to E224 for BasFR and EmbFR, respectively. There is one point for each separate CWE. Each point marks the repetition-wise best validation F_1 and the corresponding test F_1. The 20 feature-sequence-count-wise biggest CWEs are marked with a cross. Table 8.4 in the appendix is the basis of this plot.

49 SCSs, this CWE is one of the smallest CWEs. Together with the much higher corresponding CWE-wise best validation score of 87.30 %, it indicates that the tool configuration overfitted on the CWE's small validation set. A CWE-wise best test F_1 of 100 % was achieved for eleven CWEs.

With EmbFR the tool configurations are only capable of correctly predicting 46.45 % of unseen SCSs on average. This is almost the same as guessing whether an SCS is vulnerable. The lowest CWE-wise best test F_1 is 3.57 % for CWEs 242, 481, 484, and 605. Again, since each of these CWEs only consists of 49 SCSs, and since the corresponding validation score are 100 %, 100 %, 87.30 %, and 100 %, respectively, the tool configurations overfitted on each of these CWE's small validation set.

If we look at the average CWE-wise best **validation** F_1 over the 112 CWEs, which are 91.75 % and 95.79 %, a similar picture emerges.

For the former, BasFR, the average overfitting ratio of 1.11 between the validation score and the test score indicates some overfitting since its higher than the ideal value of 1.00. Section 4.2.6 contains more information about the overfitting ratio. In figure 7.2 the black, diagonal line denotes combinations of test and validation F_1s having an ideal overfitting ratio of 1.00. Tool configurations located below that line suffer from overfitting. In some cases, the best test F_1 is higher than the corresponding best validation F_1 resulting in tool configurations above that line. Since this only happens for small CWEs, we assume randomness to be the cause.

For the latter, EmbFR, the average overfittion ratio is 2.06, which corresponds to the CWE-wise best tool configurations with EmbFR highly overfitting on the respective validation data.

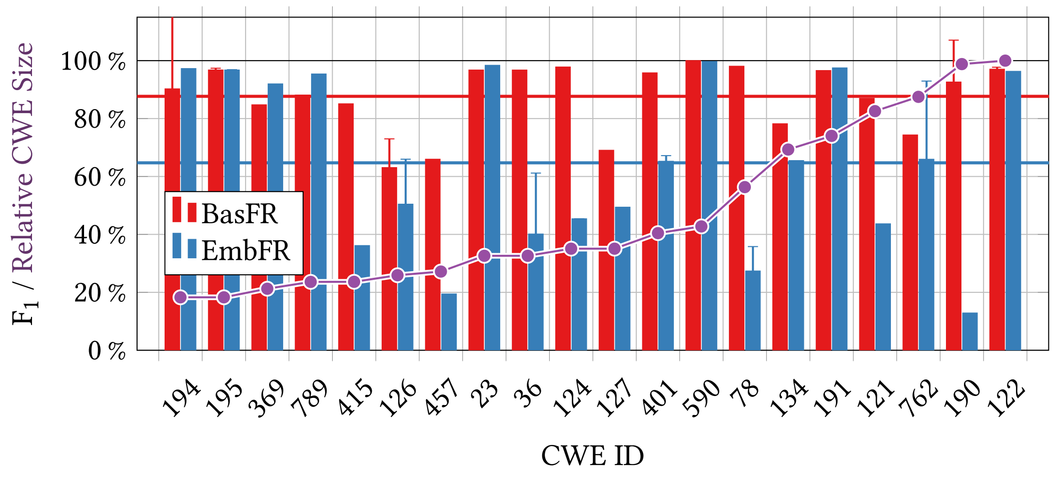
One possible explanation for the two feature representations behaving differently in terms of overfitting is that the number of weights in the RNNs differs between the two feature representation. BasFR's RNN contains 6286 weights while EmbFR's RNN has 17 730 weights. More weights allow the RNN to memorize more inputs while fewer weights force the RNN to form generalized rules to maintain a good F_1. Based on this, we assumed a better overfitting ratio when only considering large CWEs. These large CWEs have more variance due to their higher number of feature sequences. To investigate this assumption, we examined average CWE-wise best scores for only the 20 CWEs that have the highest count of feature sequences.

These 20 biggest CWEs all consist of more than 2663 feature sequences. In figure 7.2 they are marked with crosses. Their average feature sequence count is 6887.75, which is over four times higher than all 112 CWEs' average count of 1564.71. Figure 7.3 also visualizes the CWE-wise best F_1s as bar plots. BasFR' and EmbFR's average CWE-wise best test F_1s both increased by 4.69 % and 18.27 % to 87.68 % and 64.72 %, respectively.

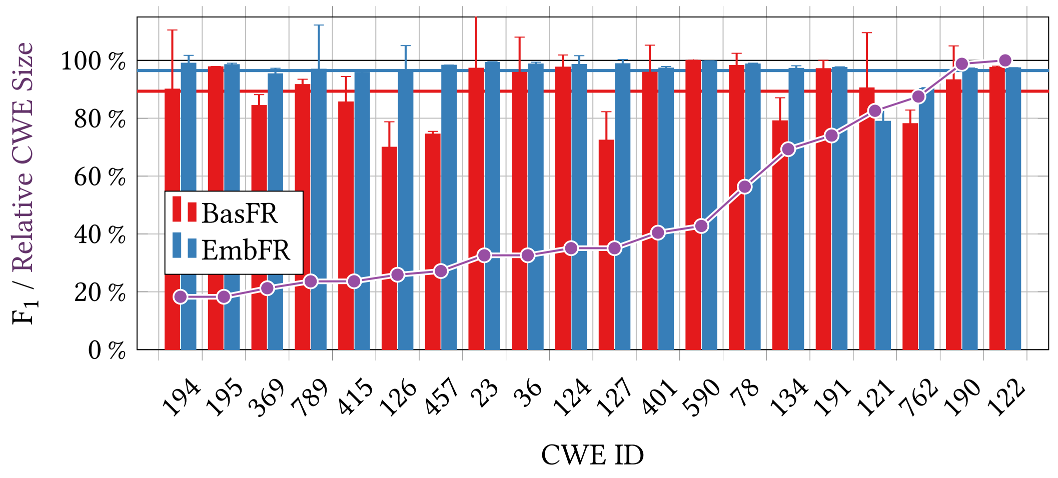
In the bar plot, the CWEs are sorted by their relative sizes, which are also plotted as purple dots. The relative size is the number of feature sequence for that CWE in relation to the plot's biggest CWE. This description generally applies to bar plots hereafter.

BasFR's average CWE-wise best validation F_1 decreased by 2.41 % yielding a decrease of the overfitting ratio from 1.11 to 1.02. A similar but more extreme effect is observed with EmbFR. The validation score slightly increased by 0.69 % but since that increase is much less than the test score increase of 18.27 %, the overfitting ratio decreased from 2.06 to 1.49. As a result, with both feature representations we observed less overfitting when only employing the 20 biggest CWEs.

For EmbFR, we expected the overfitting ratio to further converge to 1.00 when considering an even larger dataset consisting of multiple CWEs. For this, we utilized the results of experiment E231 which we conducted for comparing our best tool



(a) Test F_1s of the tool configurations' with the CWE-wise best validation F_1s.



(b) CWE-wise best validation F_1s.

Figure 7.3: Results of E1 to E112 (red bars) and E113 to E224 (blue bars) test (a) and validation (b) results for BasFR and EmbFR, respectively. We only show the 20 feature-sequence-count-wise biggest CWEs. Vertical lines denote the CWE-wise standard deviations, if available. Red and blue horizontal lines depict the feature-representation-wise means.

configuration with the literature. In this experiment, we merged the 19 CWEs used in Goseva-Popstojanova and Perhinschi's paper to one dataset. With this dataset the count of feature sequences further increased to 39 758. In addition, the tool configuration must not only predict the presence of one weakness but of 19 weaknesses. Due to the additional output neurons, the number of weights slightly increased to 17 820.

Figure 7.4 shows the results. With EmbFR, the best tool configuration's test and validation F_1s are 78.21 % and 86.37 %, respectively, which corresponds to a further decreased overfitting ratio of 1.10.

A training of the corresponding experiment using BasFR did not converge to a reasonably good validation F_1. We suspect BasFR being too simple to adapt to the more complex situation.

Another reason for EmbFR performing worse may be its more specialized structure. We put domain-specific knowledge into that structure with the result of treating features in a more specialized but less flexible way. This reduced flexibility may limit the training algorithm in generalizing the RNN on the data.

In summary, we were able to deduce a negative correlation between the number of feature sequences and overfitting on the Juliet dataset.

On the MemNet dataset we conducted the four experiments E234 to E237, which differ in their feature representation and NALU presence. Table 7.5 lists their validation and test F_1s as well as mean and SD values over these.

We achieved the best test F_1 of 87.22 % with the tool configuration using EmbFR and NALUs. The second row demonstrates the above-described situation where the test mean of 80.98 % is greater than the best test F_1 of 79.16 %.

To further evaluate the results, we utilized the IntGrad100 attribution technique to generate overlays on TP and TN SCSs from the MemNet dataset for both feature representations. Listing 7.1 shows the resulting attribution overlays of the corresponding experiments E244 to E247. For listings 7.1a and 7.1b, we overlay each source code range in red whose corresponding AST node's feature vector has a positive contribution to the buffer overflow weakness' output neuron's activation. Analogously, green rectangles visualize the positive contributions to the non-weakness' output neuron in listings 7.1c and 7.1d. The opacity of the overlays correlates with the amount of contribution.

As expected, our tool mainly focuses on literals, identifiers, and declarations. In listing 7.1b, which shows attributions for our best tool configuration, we observe high contribution of the `entity_4`'s declaration with length 24 and of the vulnerable access to its 74-th element. However, we also observe high contribution of declarations and literals which are unrelated to the vulnerability. A similar picture emerges for our best BasFR tool configuration in listing 7.1a. Interestingly, the declaration of `entity_4` in line 2 has a very low contribution such that our tool should not be able to correctly predict in theory. In practice, we assume our tool to focus on the penultimate line 4 which also contains `entity_4`'s length in that case. In general, we observed the penultimate line containing the length of the accessed array in many SCS. Based on this observation and the attribution overlays, we assume that our tool learned

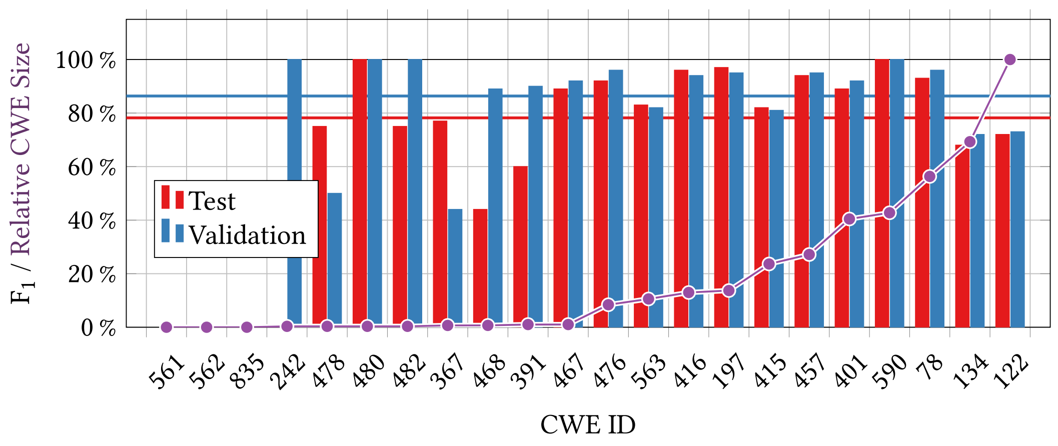


Figure 7.4: E231 test results for merged dataset with the 22 Juliet CWEs utilized by Goseva-Popstojanova and Perhinschi. No results for CWEs 561, 562, and 835 since their number of both good and bad test cases are less than seven, respectively. The test set does not contain vulnerable sequences for CWE 242.

Table 7.5: Results of E234 to E237 on the MemNet dataset.

Tool Configuration		**Validation F_1**			**Test F_1**		
Feat. Repr.	**NALU**	**Best**	**Mean**	**SD**	**Best**	**Mean**	**SD**
BasFR	yes	0.9093	0.9029	0.0048	**0.7449**	0.7757	0.0431
BasFR	no	0.9093	0.9035	0.0052	**0.7916**	0.8098	0.0324
EmbFR	yes	0.9047	0.8975	0.0074	**0.8722**	0.8170	0.0447
EmbFR	no	0.9073	0.9033	0.0036	**0.7677**	0.8225	0.0389

such generation process artifacts instead of truly generalized rules to detect buffer overflows. If that assumption holds, the attribution overlays gave us helpful insights on our tool's prediction.

The attribution overlays on the TN in listings 7.1c and 7.1d differ more between BasFR and EmbFR. The former's attribution are similar to the just-mentioned TP's ones, i.e., our tool focuses on the array declaration and the array access to predict the SCS as non-vulnerable. Our tool's attribution with EmbFR look diffuse and unrelated to the SCS' important entities. As a consequence, we can not extract useful information from that attribution.

To summarize the evaluation of the feature representations in terms of F_1, we cannot conclude one to be better than the other. Both have their advantages and disadvantages. EmbFR suffers from more overfitting on small datasets due to its more complex structure. Thus, BasFR achieved on average higher performance on the

```
int entity_2 = 44;
char entity_4[24] = "";
memset(entity_4, 'v', 24-1);
entity_4[24-1]='0';
entity_4[74] = 'p';
```

(a) TP of tool configuration with BasFR.

```
int entity_2 = 44;
char entity_4[24] = "";
memset(entity_4, 'v', 24-1);
entity_4[24-1]='0';
entity_4[74] = 'p';
```

(b) TP of tool configuration with EmbFR.

```
int entity_8 = 29;
char entity_2[12] = "";
memset(entity_2, 'L', 12-1);
entity_2[12-1]='0';
entity_2[6] = 'e';
```

(c) TN of tool configuration with BasFR.

```
int entity_8 = 29;
char entity_2[12] = "";
memset(entity_2, 'L', 12-1);
entity_2[12-1]='0';
entity_2[6] = 'e';
```

(d) TN of tool configuration with EmbFR.

Listing 7.1: Results of E244 to E247 visualizing TPs (a, b) and TNs (c, d) of the MemNet test set using BasFR (a, c) and EmbFR (b, d). The overlay opacity and color correspond to the positive contribution to the prediction being vulnerable and non-vulnerable, respectively.

separate Juliet CWEs. In contrast, on more complex dataset BasFR reached its limits by not converging properly anymore. EmbFR performs better in such situations with almost no overfitting.

Epoch Count This paragraph describes the feature representation results in terms of the epochs after which the training algorithm yielded the repetition-wise best tool configuration. Again, we look at the $2 \cdot 112 = 224$ experiments E1 to E112 and E113 to E224 whose instructions were to train an RNN with BasFR and EmbFR on each separate Juliet CWE, respectively.

Figure 7.5 marks each combination of repetition and epoch in which the training algorithm yielded a repetition-wise best tool configuration. The first 25 epochs cover approximately a third of these. The counts of repetition-epoch combinations go down further with increasing epochs and reach their minimum at about 350. On overall average, the training algorithm yields the best tool configuration after epoch 97. If we examine BasFR and EmbFR separately, the average epochs are 84 and 110, respectively. We did not observe much difference within each feature representation's repetitions.

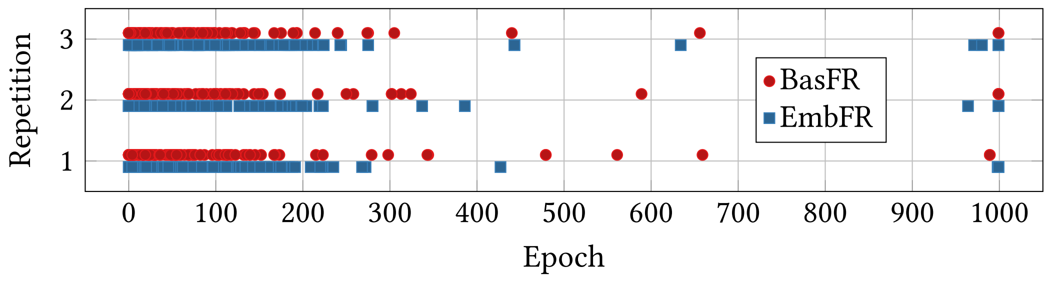


Figure 7.5: E1 to E112 (red dots) and E113 to E224 (blue squares) combinations of repetition and epoch in which the training algorithm yielded the repetition-wise best tool configurations.

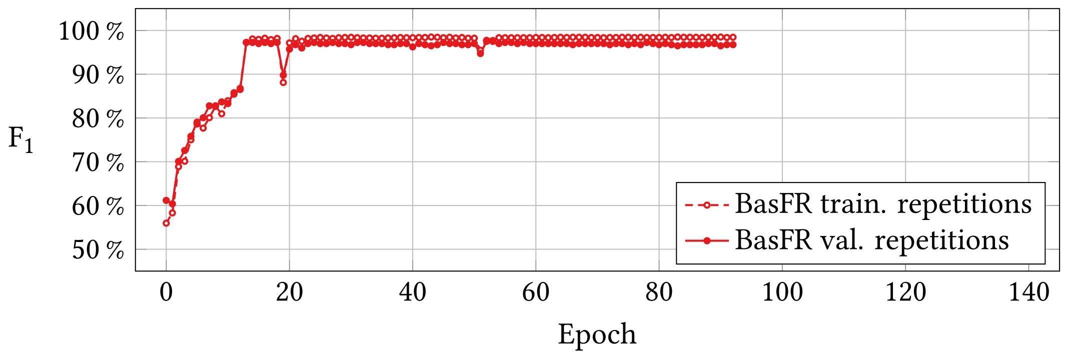
With EmbFR the training algorithm needs an average of 26 more epochs before it yields the repetition-wise best tool configurations. This results in a 30 % higher time consumption compared to utilizing BasFR. This is amplified by the fact that each epoch with EmbFR takes longer to complete than with BasFR. The factor varies between approximately 1.5 to 3 depending on the system's load and the CWE's feature sequence count and lengths. Since each epoch's duration greatly varied within each repetition, we refrain from giving concrete numbers.

We also exemplified the F_1 curves during two specific training process on the Juliet CWE 195 dataset. Figure 7.6 plots each repetition's training and validation score curve over the epochs.

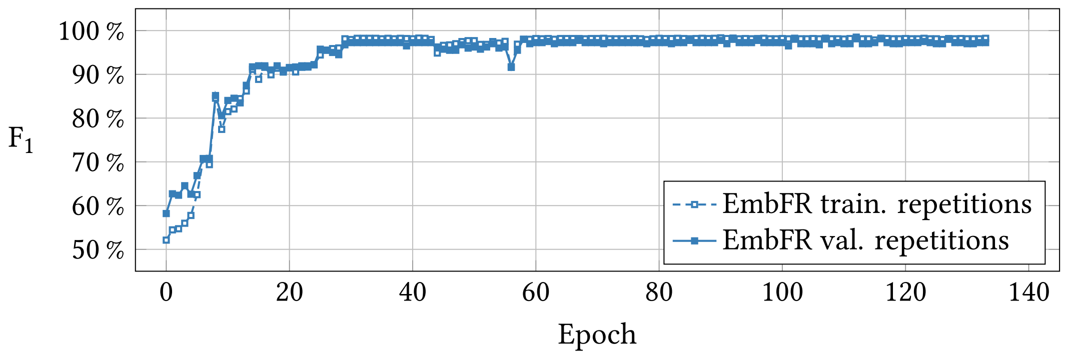
When looking coarsely, for the repetition with BasFR in figure 7.6a, which is denoted by red lines with circular marks, the F_1s reach their maximum range for the first time after 15 to 20 epochs. The maximum range is around 98 %. Early stopping terminates the training processes before epoch 100 since there was no progress in terms of achieving higher validation F_1s for 50 epochs. For the repetition with EmbFR in figure 7.6b, which is denoted by blue lines with square marks, the scores reach their maximum range for the first time after 25 to 40 epochs. This again exemplifies EmbFR's higher time consumption.

When looking closer, dashed lines with unfilled marks denote training F_1s while the solid lines with filled marks identify validation scores. In the beginning there is no clear order between corresponding pairs of training and validation scores. Starting with epoch 50, almost each training F_1 is slightly higher than its corresponding validation score. Thus, we did not observe high overfitting on the training data taking place with any feature representation.

We explain the decrease of BasFR's repetition's scores at epoch 19 and the subsequent increase by the fact that the training algorithm jumps from a local minimum to a higher area and then starts approaching another (or the same) local minimum again.



(a) BasFR.



(b) EmbFR.

Figure 7.6: Curves for training and validation F_1s of two CWE 195 training processes with BasFR (a) and EmbFR (b). For clarity, we only display the first repetition of each training.

In summary, we concluded the epoch count with EmbFR being about 30 % higher than with BasFR. This is amplified by the fact that each epoch with EmbFR takes approximately 1.5 to 3 times longer than with BasFR.

Section 5.2 already answered RQ1's first sub-questions which are about a feature representation's necessary information and the comparison of BasFR's and EmbFR's structure. To complete the answering of RQ1, we conclude that BasFR's and EmbFR's effects on the prediction power depend on the size of the dataset. Roughly, tool configurations with BasFR worked better on small datasets since EmbFR suffered from overfitting there. On bigger and more complex datasets, tool configurations with EmbFR achieved higher performance compared to configurations with BasFR. In terms of resource consumption, we conclude that our tool with BasFR requires about 30 % fewer epochs on average compared to EmbFR. The picture that BasFR results in a lower resource consumption is solidified by the fact that each epoch with EmbFR last approximately 1.5 to 3 times longer.

7.2.2 NALUs

The experiments' E225 to E230, E236, and E237 instructions were to train tool configurations without NALU layers for some Juliet CWEs and the MemNet dataset. We compared those experiments' resulting F_1s with the above-mentioned F_1s, which we achieved with NALU layers being present. Afterwards, we discuss the extrapolation capability of our tool based on the experiments E238 and E239, which utilize our ArrDeclAccess dataset and the special extrapolation split.

F_1 and Epoch Count For six separate Juliet CWEs, figure 7.7 plots the test and validation scores of our experiments without NALUs in red. The figure also plots the F_1 of the experiment with NALU layers, which we already discussed above. We arbitrarily chose these six CWEs as described in section 7.1.3.

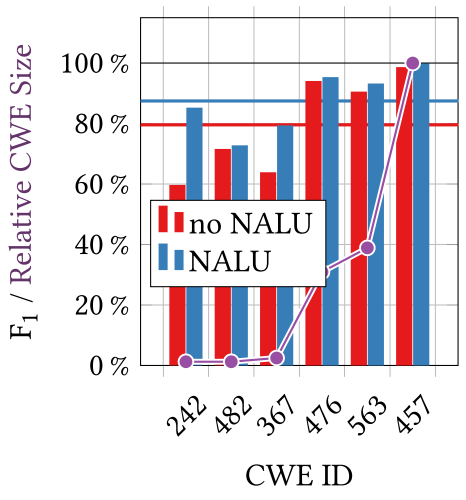
The test F_1s of the CWE-wise best tool configurations in figure 7.7a show the prediction power with NALU layer to be better than without. Without NALU layers, the average test F_1s of the CWE-wise best tool configuration is 79.57 %. With NALU layers it is 87.46 %. The corresponding on average CWE-wise best validation F_1s in figure 7.7b are 87.10 % and 94.72 %, respectively. These indicate both configurations overfitting similar with on average overfitting ratios of 1.09 and 1.08, respectively.

The presence or absence of NALUs did not have notable effect on the epoch counts after which the training algorithm yielded the repetition-wise best tool configuration.

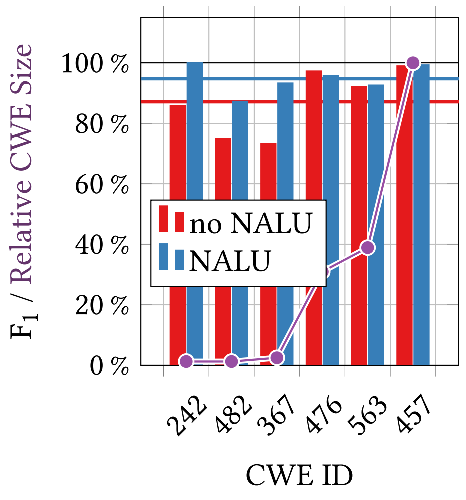
On the MemNet dataset, the already mentioned results in table 7.5 do not show any clear distinction between trainings with and without NALUs. While the best validation F_1s are around 90 %, the test F_1s vary between 74.49 %, which we achieved with BasFR and NALUs, and 87.22 %, which we achieved with EmbFR and NALUs. In case of BasFR the presence of NALUs had a negative influence. In contrast, with EmbFR, their presence was beneficial. Thus, the scores do not allow conclusions in terms of NALUs on the MemNet dataset.

In summary, the presence of NALU layers does not worsen the prediction power and generalization. In many cases, the presence of NALUs result in better F_1s. Furthermore, they do not affect the training processes' time consumption.

Extrapolation So far, we considered test sets from either a random split or a predefined split. In both cases, the SCSs in the training, validation, and test set have similar structure and also literals of similar intervals. Since NALU layers support the training algorithm in learning basic mathematical operations according to Trask et al. [Tra+18], we performed experiments to explore this ability.



(a) Test F$_1$s.



(b) Validation F$_1$s.

Figure 7.7: Results of E225 to E230 without NALU (red bars) and corresponding experiments with NALU (blue bars) for six separate Juliet CWEs.

We do so by utilizing our ArrDeclAccess dataset and our special extrapolation split. The training and validation set consist of SCSs in which we declare and iterate over arrays with upper bounds between zero and 100. The test set contains SCSs with upper bounds between 0 and 1000. Section 5.1.3 contains in-detail information about the exact integers we used for the respective upper bounds.

For both experiments E238 and E239 the best tool configurations achieved training and validation F$_1$s of 100 %. However, they both predicted all test SCSs with upper bounds over 100 as vulnerable. Hence, we were able to reproduce the extrapolation problems described by Trask et al. [Tra+18]. However, we did not expect this behavior for the NALU experiment.

We assumed overfitting to be the issue for the missing generalization of the tool configuration with NALUs since only if the training algorithm learns a simple subtraction of the respective upper bounds, the resulting tool configuration will be able to successfully extrapolate. Such a subtraction should in theory be learnable with one LSTM and one NALU layer. We therefore tried to reduce the complexity of our RNN with the goal of forcing the training algorithm to generalize.

We separately reduced the LSTM, NALU, and FC neuron counts down to four and conducted experiments for these reduced combinations of neuron counts. We also increased the size of our dataset by adding SCSs with upper bounds up to 1000. For each of these experiments, we either observed no change to the baseline in terms of extrapolation or the training F$_1$ did not converge to approximately 100 %. Allowing the training to last 1000 epochs without early stopping did not change the situation.

We further removed complexity from our task as described in the following. Since the only variation in the ArrDeclAccess dataset are the upper bounds of the declarations and iterations, we removed all other information from our feature representation until receiving a single feature vector for each SCS. Each of these feature vectors only contains array type lengths and integer literal values. For our ArrDeclAccess dataset these feature vectors are of length 5.

With the resulting removal of the need for an LSTM layer, we further reduced our ANN structure down to two layers. The first layer is either a NALU layer of four NALUs or a four neuron FC layer. The second layer is a final softmax FC layer of two neurons. This resulted in the ANN having 50 and 34 weights in total, respectively.

With these two configurations, our tool eventually achieved training and validation F_1s of 100 % together with notable extrapolation performance. Table 7.6 shows these two experiments' results. We used SCSs with upper bounds up to 1000.

Without the NALU layer, we achieved 100 % training and validation F_1s after epoch 52 but let the training algorithm continue minimizing the loss function. The training algorithm further improved in terms of loss function value. With the best tool configuration from epoch 722, our tool correctly predicted the presence of the buffer overflow weakness in SCSs with upper bounds up to 20 780. With higher upper bounds the tool always predicted the SCSs to be non-vulnerable.

With a NALU layer, we achieved 100 % training and validation F_1s after epoch 53. However, the training algorithm did not further improve in terms of loss function value. The configuration from epoch 53 extrapolated only up to 1520. We decided to retrain the configuration for 1000 more epochs and achieved a new best configuration after epoch 926 of the retraining. The extrapolation for this configuration slightly increased to upper bounds of 1580.

With more time, our next step would have been the continued reduction of the NALU layer's neuron count.

In summary, we showed that good extrapolation up to a factor of 22 is possible with ANNs. However, we were not able to reach this factor with NALU layers. Since the experiments with a NALU layer were not conclusive, we do not know in which way NALUs benefited the F_1 during the experiments on the other datasets. Further investigations in requirements on the environment in which a successful training of NALUs is feasible are the subject of future work.

To answer RQ3 we conclude that the presence of NALU layers does not worsen our tool's prediction power and ability to generalize. In many cases, the presence of NALUs results in better F_1s. Furthermore, the presence of NALUs do not affect the training processes' time consumption. In terms of extrapolation, we cannot conclude that NALUs increase our tool's extrapolation capability. Decisive conclusions require further experiments.

7.2.3 Datasets and Comparison with Literature

This section compares our tool with tools from the literature in terms of prediction power. To be more precise, we compare ourselves with Goseva-Popstojanova and Perhinschi's, Russell et al., and Choi et al.'s benchmarking results given in their respective papers [GP15; Rus+18; Cho+17].

For Goseva-Popstojanova and Perhinschi's and Russell et al.'s static analyzers, a single, specific analyzer configuration is capable of detecting and distinguishing 22 and 149 weaknesses, respectively. Thus, only our tool configuration which we trained on the corresponding merged dataset is valid for comparison with their results.

Results of tool configurations trained on each CWE separately are not directly comparable, since each tool configuration only tackled the simpler task of detecting a single weakness. Averaging the multiple two-class predictions from the respective best tool configurations to one multi-class prediction would require an appropriate selection mechanism, which in turn would probably result in a decreased F_1. However, in situations in which only vulnerabilities of a certain CWE are of interest, the results of the tool configuration trained on that separate CWE can be validly used for comparison. We therefore provide both results.

Goseva-Popstojanova and Perhinschi compared three conventional, i.e., non-ANN-based, static analyzers that are not further specified in their paper. They utilized only 22 specific CWEs for this since all other CWEs were not covered by all three static analyzers. Figure 7.8a plots the CWEs and their respective accuracies for the three static analyzers. The static-analyzer-wise mean accuracies are 59 %, 67 %, and 72 %, respectively.

We created a merged dataset from the 19 CWEs by merging all their SCSs. As mentioned above the CWEs 561, 562, and 835 are too small to train our tool. As a result, that dataset has SCSs of 19 different weaknesses and of the non-weakness. After training our tool with EmbFR on that dataset, it reached the accuracies plotted as red bars in figure 7.8b. We aligned the bars directly below the bars of figure 7.8a. Our on average test F_1 is 78.36 %, which is 6 % higher than the average accuracy of 72 % of the best conventional static analyzer.

For our best tool configurations trained on the separate CWEs, we plotted the respective accuracy scores as blue bars in the same figure. Our best tool configurations

Table 7.6: Results of extrapolation experiments with minimalist ANN and feature representation on the ArrDeclAccess dataset with upper bounds of 1000.

Tool Configuration			Training F_1 /	Extrapolation
NALU	**Weights**	**Epoch**	**Validation F_1**	**Upper Bounds**
no	34	722	100 % / 100 %	**20 780**
yes	50	53 + 926	100 % / 100 %	**1580**

utilize BasFR. Its mean accuracy for the 19 CWEs is 81.70 % as depicted by the horizontal blue line. Except for CWEs 367, 415, 457, and 478, the respective tool configuration achieved an higher accuracy on that CWE than all three conventional static analyzers.

Russell et al. did not provide specific information about which Juliet CWEs and which specific SCSs they used for their benchmark. They achieved 84.00 % F_1 on that dataset.

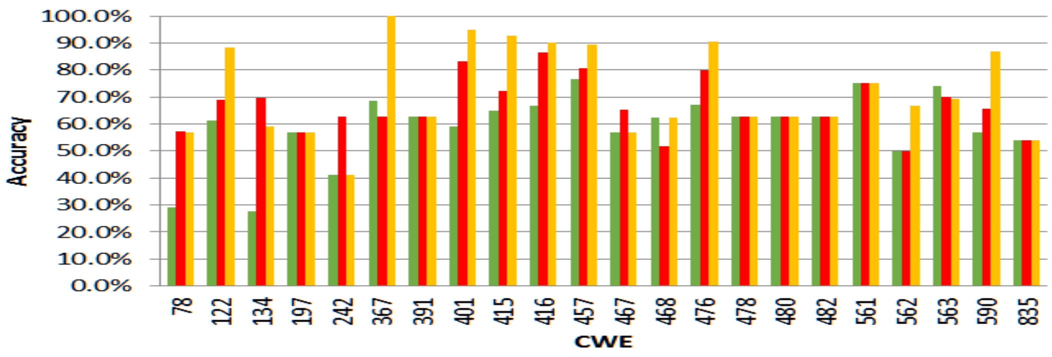
For comparison, we use the results of the experiments E231 and E233 as well as the results of the experiments discussed in section 7.2.1. The former experiments were conducted on merged CWE datasets while the latter were performed on separate CWEs.

The experiment design section 7.1.3 described that we were unable to train a tool configuration using the whole merged Juliet dataset. Instead, we trained two tool configurations on two datasets merged from two subsets of all CWEs.

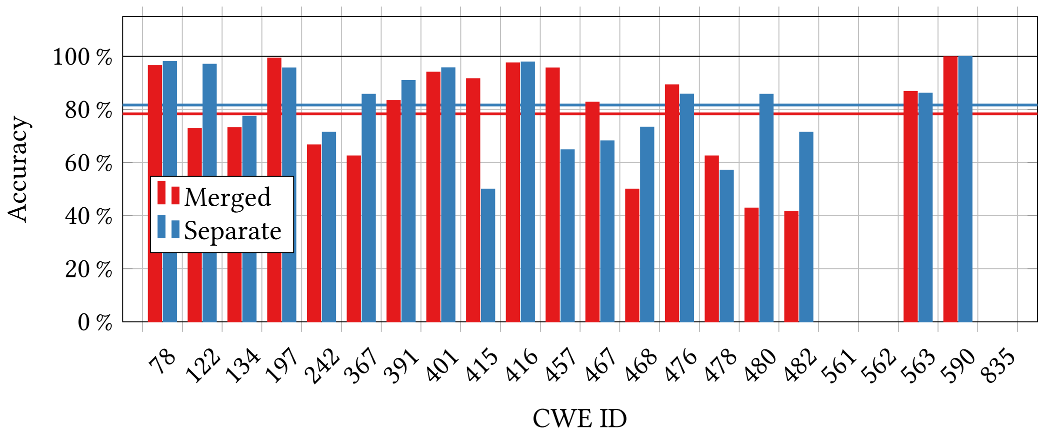
One of these subsets are the 19 CWEs from above that we used for comparison with Goseva-Popstojanova and Perhinschi's result. Figure 7.4 visualizes the resulting F_1s. With this tool configuration, we achieved an average F_1 of 78.21 %, which is noticeably worse than their F_1 of 84.00 %. An average overfitting ratio of 1.04 indicates that our tool configuration slightly overfits. Besides, explaining the difference between their and our F_1 is hard since we do not know much about their concrete dataset.

We derived our second merged dataset from the CWE taxonomy as described in section 7.1.3. In summary, 47 654 feature sequences are present in this merged dataset. For each CWE there are at least 300 features sequences except for CWE 785, which only has 49.

Figure 7.9 contains the results of the best tool configuration we trained on that dataset. The configuration uses EmbFR and we achieved an average F_1 of 87.80 % on the test data. In contrast to above, this F_1 is better than the one by Goseva-Popstojanova and Perhinschi. As visualized in figure 7.9a, almost no overfitting is taking place. This is



(a) Results for Goseva-Popstojanova and Perhinschi's experiments with three conventional static analyzers. Plot copied from their paper [GP15]. The static-analyzer-wise means are 59 %, 67 %, and 72 %, respectively.

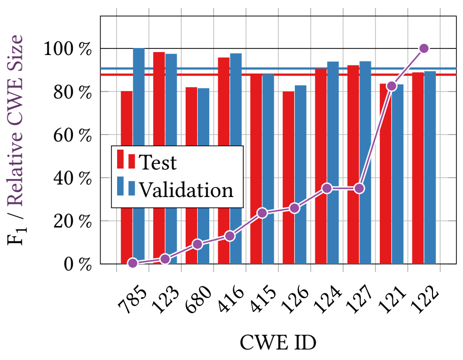


(b) Our test results of 19 experiments from E113 to E224 (red bars; left) and of experiment E231 (blue bars; right). No results for CWEs 561, 562, and 835 since their number of both good and bad test cases are less than seven, respectively. The overall means are depicted as red and blue horizontal lines, respectively.

Figure 7.8: Comparison of results of Goseva-Popstojanova and Perhinschi's (a) and of our (b) experiments for 22 Juliet CWEs utilized by Goseva-Popstojanova and Perhinschi in their paper [GP15].

reflected in the average overfitting ratio of 1.02. Again, CWE 785 is an exception with an overfitting ratio of 1.25. Both the merged, large dataset having a small overfitting ratio and the smallest CWE having the largest ratio, align with the above discussed negative correlation between the overall feature sequence count and overfitting.

The corresponding test confusion matrix in figure 7.9b reveals CWEs 121 (Stack-based Buffer Overflow) and 126 (Buffer Over-read) having the most FPs. The highest intra-CWE false classification count of 9 + 19 = 28 occur between CWE 124 (Buffer



(a) Test (red) and validation (blue) F_1s.

		Predicted										
		121	122	123	124	126	127	415	416	680	785	NV
Actual	121	338	0	0	0	0	0	0	0	0	0	3
	122	0	267	0	2	0	3	2	0	3	0	6
	123	0	0	26	0	0	0	0	0	0	0	0
	124	0	3	1	251	0	19	4	0	4	0	0
	126	0	1	0	1	194	1	1	0	5	0	10
	127	0	2	0	9	1	282	3	0	3	0	1
	415	0	1	0	0	0	1	151	0	0	0	1
	416	0	0	0	0	0	0	1	65	0	0	2
	680	0	0	0	0	1	0	0	0	99	0	0
	785	0	0	0	1	0	0	0	0	0	2	0
	NV	131	45	0	9	77	6	27	3	28	0	2526

(b) Test confusion matrix.

Figure 7.9: Results of E233 for merged dataset with CWEs from CWE taxonomy group Incorrect Access of Indexable Resource (Range Error).

Underwrite) and CWE 127 (Buffer Under-read). We manually compared many of their SCSs and observed that they are in fact very similar. They only differ in the reads' or writes' respective source and destination being swapped.

In summary, we can only draw limited conclusions because Russell et al. did not provide in-detail information and because we did not train a tool configuration on the merged dataset of all CWEs. Despite that, we can conclude that our F_1s are in the same order of magnitude as theirs.

Choi et al. used the MemNet dataset and their dataset split. We therefore did so too in our MemNet experiments E234 to E237, whose results are listed in table 7.5 and were discussed above.

Based on our results, we identified the tool configuration with EmbFR and NALU layers to perform best. We therefore compared that tool configuration's results with the results achieved by Choi et al. on their four test sets of different levels. As explained in section 5.1.2, the levels correspond to an increasing complexity in its corresponding SCSs. Table 7.7 shows Choi et al.'s F_1s for the four test set levels next to our F_1s.

We achieved better F_1s on all four test sets with our scores being 3.86 % higher on average. The results also show our tool not being affected by higher complexity in the SCSs. Instead, we observed the opposite since the difference between our and their score becomes roughly larger with increasing levels.

At this point, we are able to answer RQ5. We conclude our tool's prediction power scores to be at least 6 % and up to 19 % higher than Goseva-Popstojanova and Perhinschi's results [GP15]. In comparison with Russell et al., we cannot conclude since a

Table 7.7: Results of Choi et al.'s experiments and of our E235 for the four MemNet test sets. A higher level corresponds to the test set containing more complex SCSs. As scores were given with two decimal places in the paper, our scores were also formatted in this way.

Test Set	**Test F_1**	
Level	**Choi et al.**	**Thesis**
1	0.84	**0.87**
2	0.85	**0.86**
3	0.83	**0.88**
4	0.82	**0.89**

sound conclusion would require more information about their experiment design and would require further experiments [Rus+18]. In comparison with Choi et al., we conclude that our scores are at least 1 % and up to 7 % higher than theirs [Cho+17].

For our best tool configuration, we computed attributions for all four kinds of prediction correctnesses. TP and TN cases are located in listings 7.1b and 7.1d, and they were already discussed above. Listing 7.2 contains examples for the remaining kinds, FP and FN.

```
char entity_4[3] = "";
char* entity_8;
char entity_6[32] = "";
memset(entity_4, 'f', 3-1);
entity_4[3-1]='0';
memset(entity_6, 'A', 32-1);
entity_6[32-1]='0';
entity_8 = (char*)malloc(55*sizeof(char));
entity_8[0]='0';
entity_6[30] = 'n';
```

(a) Predicted vulnerable but actually non-vulnerable (FP).

```
char* entity_7;
char entity_6[35] = "";
char entity_2 = 'X';
entity_7 = (char*)malloc(69*sizeof(char));
entity_7[0]='0';
memset(entity_6, 'A', 35-1);
entity_6[35-1]='0';
entity_6[43] = 'y';
```

(b) Predicted non-vulnerable but actually vulnerable (FN).

Listing 7.2: Results of E248 and E249 visualizing an FP (a) and an FN (b) of the MemNet test set. The overlay color correlates with the positive contribution to the prediction being vulnerable and non-vulnerable, respectively.

Listing 7.2a exemplifies our tool falsely reporting a vulnerability which corresponds to an FP. The overlay shows the contributions of the AST nodes to the vulnerable output neuron. As mentioned above, we observed the penultimate line to contain the length of the last line's accessed array in many SCS. Based on this, one explanation would be that our tool falsely assumes that the literal zero in the penultimate line 9 is the length of the array accessed in line 9.

Listing 7.2b exemplifies our tool failing to predict a vulnerability which corresponds to an FN. The overlay shows the contributions of the AST nodes to the non-vulnerable output neuron. For instance, the number 69 highly contributes to our tool's prediction. However, that number is unrelated to the vulnerability. In addition, the integer literal in line 8, which is used for accessing the array and is therefore crucial for detecting the vulnerability, does not have much influence.

7.2.4 False Positives

During our above discussions, we only considered F_1 for measuring the prediction power of our tool configurations. As introduced in section 4.2.6, precision incorporates the TP and FP counts of a tool configuration in a way that it negatively correlates with the FP count. Analogously, recall does the same for the FN count. Since the F_1 is the harmonic mean of precision and recall, it gives a good overall impression of the prediction power.

In many environments, a low FP count, i.e., a high precision, is favored over a low FN count. Static analyzing in an *Integrated Development Environment* (IDE) is an example where high precision is important. If the IDE reports a vulnerability, a vulnerability should actually be there. Otherwise, the developer is interrupted in his work for no real reason, looses his trust in the IDE's reports, and perhaps disables the whole analysis feature. An undetected vulnerability, in contrast, does not worsen the user experience compared to an IDE without such static analysis. Of course, a tool that never reports a vulnerability has no use.

One approach to increase the focus on a low FP count is the use of a higher weight for these inside the loss function. This should be combined with test metrics that weight low FPs higher than low FNs. The disadvantage of this approach is its time consumption since the training process must be repeated or continued. Table 7.8 shows the confusion matrix of the already discussed experiment E231, whose configuration uses the merged 19-CWE dataset, EmbFR and NALU layers. We condensed the actual 20×20 matrix by aggregating the 19 CWEs to a single weakness and keeping the non-weakness as it is. Analogously, the table also contains the condensed confusion matrix for the same tool configuration but retrained with higher weights for FPs inside the loss function. While the F_1s of approximately 78 % are the same for both, the distribution of FP and FN is different.

With E231 the counts of FP and FN are unexpectedly unbalanced. In other experiments, we usually obtained similar FP and FN counts. We assume randomness to be the cause. With E232 we obtained a low count of FP as expected. However, not only the FP count is lower but also the number of TP. As a consequence, our tool less often predicts a vulnerability.

This shows that the approach reduces the FP count. If the demand for low FP is known before the training process, there is no additional overhead of this approach. Otherwise, the training must be repeated or continued.

Another approach, which works without a subsequent training process, is to add a prediction threshold. Usually, the class whose output neuron's activation is the highest becomes the tool's predicted class. In binary classification and with a softmax layer, this corresponds to the threshold being 0.5. With a specific prediction threshold, a class must have the highest activation and its activation must exceed that threshold. Otherwise, the negative class is predicted. This way, we can tune how certain our tool must be in its prediction. The validation set should be incorporated for this tuning process to be able to still assess the prediction power with the chosen threshold on the unseen test data.

In case of binary classification, a single *Precision-Recall curve* (PR curve) gives a visual overview of the precision and recall depending on the threshold. Figure 7.10 contains such a curve for our best MemNet tool configuration. Since the training algorithm maximized the F_1 score, the precision and recall values at threshold 0.5

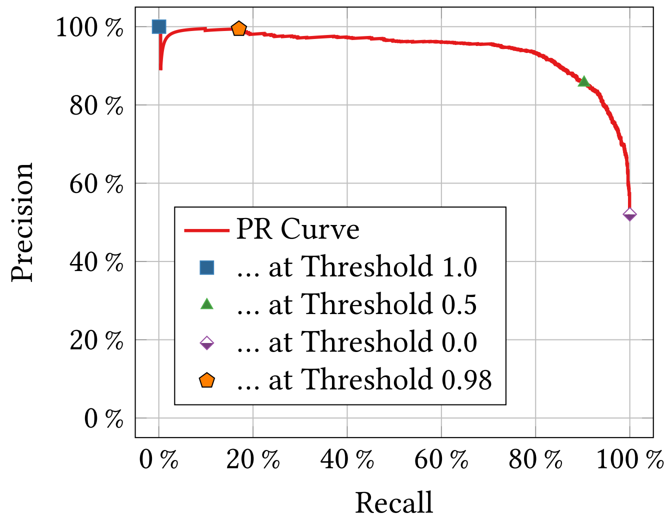


Figure 7.10: PR curve for our best MemNet tool configuration.

Table 7.8: Condensed test set confusion matrices for E231 and E232 to illustrate the effect of different loss function weights.

		E231 (Baseline)		**E232 (Focus on Low FP)**	
		Predicted		**Predicted**	
		Vulner.	**Non-vulner.**	**Vulner.**	**Non-vulner.**
Actual	**Vulner.**	1965	18	1378	585
	Non-vulner.	607	3373	32	3968

are both equally high. By varying the threshold, we can tune our tool configuration to arbitrary precision-recall-combinations on the curve. For example, by setting the tool configuration's threshold to 0.98, which is marked by the orange pentagon in the figure, its precision of 99.42 % would result in nearly no FP. At the same time, it is still able to detect some vulnerabilities.

In summary, our tool is able to operate in different FP and FN modes, which is an advantage compared to conventional static analyzers.

7.2.5 Attribution Techniques

The above sections already used the IntGrad100 attribution technique to create attribution overlays on SCS and they discussed their effect in terms of explainability.

To compare the four different attribution techniques GradInput, IntGrad20, IntGrad100, and IntGrad300, listing 8.1 in the appendix shows their attribution overlays on the Juliet bad SCS from testcase `CWE124_Buffer_Underwrite__char_declare_cpy_17`. This SCS is described in section 5.1.1. We used our tool configuration with EmbFR that we trained on the merged CWE taxonomy dataset, and that correctly predicted the SCS.

The overlays show the positive contribution of the AST nodes to the output neuron for CWE 124. They differ in the AST nodes they highlight and in their opacity. All overlays focus on variable references, array accesses, literals, loops, and function calls. None of them shows high contribution of declaration AST nodes. The last line has high contribution in all cases.

With GradInput in listing 8.1a the overlay looks haphazard due to the for loop's high contribution and the missing contribution of the actually vulnerable `strcpy` call. With IntGrad a more stable and clearer picture emerges which focuses on the actually vulnerable call and the accesses of both its argument variables `data` and `source`. While there is notable difference between IntGrad20 and IntGrad100, IntGrad100 and IntGrad300 do not differ much.

We assume the high focus on the for loop AST node to be related to the for loop only being present in some functional variants. The ANN may use its existence to determine the SCS's functional variant. Depending on this it may focus more on the succeeding AST nodes.

We also measured the time consumption of GradInput, IntGrad20, IntGrad100, and IntGrad300 and list the results in table 7.9. We did so for the Juliet SCS used for the attribution technique comparison above and for the first SCS from the level 1 MemNet test set. The latter is shown in listing 7.1c. We used our best MemNet tool configurations for each feature representation.

As expected, the number of IntGrad steps has linear influence on the time consumption in all three cases. Since GradInput needs one gradient calculation, it also fits into that schema.

We noticed differences between the SCS and the feature representations. The time consumption with BasFR is between 2200 % and 2800 % smaller than with EmbFR. This makes sense since the RNN structure with BasFR does not contain branches and merges and is generally smaller than EmbFR. With EmbFR we expect TensorFlow to calculate multiple gradients for different paths through the RNN. With BasFR we assume it to perform the gradient calculation in one pass.

When we only look at the tool configurations using EmbFR, the time consumption with the Juliet SCS is about 77 % higher than with the MemNet SCS. This is also plausible since the former's feature sequence length of 76 is 100 % larger than the latter's.

We conclude IntGrad100 to be a good trade-off between a clear attribution overlay and time consumption.

To support a developer during typing in an IDE, an overlay should appear on vulnerable SCS a few seconds after the developer finished typing it. Since the time consumption of tool configurations with EmbFR are already higher than that for small SCS as discussed above, such configurations are unsuitable. They can still be applied upon the developer's request or integrated into Continuous Integration / DevOps workflows. With BasFR, checking for vulnerabilities may be feasible during typing depending on the SCS's length and number of IntGrad steps.

To answer RQ4, we conclude attribution overlays helping us to understand the predictions in many cases while being unhelpful in other cases. In terms of time consumption, their use in an IDE is feasible in general but depends on the chosen attribution technique and the ANN's complexity.

Finally, we answer RQ2 which is about the general feasible of vulnerability detection with our novel combination of AST-based feature representations together with NALU- and LSTM-based RNNs. We conclude that our tool is capable of detecting

Table 7.9: Time consumption of different attribution techniques for two source code snippets. Numbers are per output neuron in seconds.

SCS's	Feature	Time Consumption [s]			
Dataset	Repr.	GradInput	IntGrad20	IntGrad100	IntGrad300
Juliet	EmbFR	0.2901	5.9735	30.2367	90.4941
MemNet	BasFR	0.0091	0.1747	0.7933	2.3841
MemNet	EmbFR	0.1629	3.4053	17.0079	48.9751

vulnerabilities in SCS with scores better than the ones from the literature. We identified overfitting to be an issue for small datasets. Based on our pre-experiments, we further conclude that the selection of HP values has high influence on the prediction power.

7.3 Threats to Validity

Some results and conclusions of this thesis are to be viewed with caution because issues threaten their validity. We differentiate between systemic issues, which were already inherent in our RQs, and issues that arose during the elaboration of this thesis. The following sections describe issues of both kinds.

7.3.1 Systemic Issues

All datasets utilized in this thesis are synthetic, i.e., their containing SCS are simpler and less versatile than natural SCSs written by humans [Nat12]. As a consequence, statements made in this thesis are limited to synthetic SCSs and cannot be simply transferred to natural SCSs found in the real world. Section 5.1 gives more information about this and other threats to validity.

7.3.2 Experiments

During our pre-experiments we varied four HPs and assumed their best combination to also being the best combination for our main experiments. However, since the other HPs differed between pre- and main experiments, this assumption is not necessarily valid. For example, while a learning rate of 0.001 performed best with EmbFR on the MemNet dataset, this does not necessarily hold for other datasets or feature representations. See section 7.1.2 for a more in-depth discussion.

For the comparison with the results of Russell et al. we were unable to conduct experiments that would have allowed a sound and conservative comparison. The reason for this is that their paper does not contain detailed information about the Juliet CWEs utilized in their experiments. Our conservative approach to use all CWEs in a merged dataset was unsuccessful because of implementation-related problems during the corresponding experiment.

Chapter 8

Conclusion and Outlook

The introduction showed the need for vulnerability-free software in our increasingly digitial world and static analysis as a way to achieve this. With this goal in mind, we created and evaluated an ANN-based static analyzing tool that utilizes a novel combination of different techniques. This combination consists of the LSTM technique for capturing long-distance relationships within the SCSs and of NALUs to support our tool in learning basic arithmetic operations. We thoroughly described not only these theoretical foundations but also attribution techniques such as GradInput and IntGrad. They helped to explain an SCS's entities' contributions to our tool's prediction. With these contributions, our tool automatically constructs attribution overlays on the SCS to get explainable insights.

In correspondence to our RQ1, we constructed various features from the SCSs' ASTs with the help of our custom C++ parser based on Clang. Our two feature representations BasFR and EmbFR both employ these features but differ in the amount of manually crafted features and in their corresponding RNNs' structure. Our evaluation concluded that BasFR's and EmbFR's effects on the prediction power depend on the size of the dataset. Roughly, tool configurations with BasFR worked better on small datasets while the one with EmbFR achieved higher performance on bigger and more complex datasets. In terms of resource consumption, we concluded that our tool with BasFR requires about 30 % fewer epochs on average compared to EmbFR. This conclusion is solidified by the fact that each epoch with EmbFR last approximately 1.5 to 3 times longer.

Based on the evaluation for RQ3, we concluded that the presence of NALU layers does not worsen our tool's prediction power and ability to generalize. In many cases, the presence of NALUs results in better F_1s. In terms of extrapolation, we could not conclude that NALUs increase our tool's extrapolation capability. Decisive conclusions require further experiments.

In comparison to the literature as required by RQ5, we concluded our tool's prediction power scores to be at least 6 % and up to 19 % higher than Goseva-Popstojanova and Perhinschi's results [GP15]. In comparison with Russell et al., we could not conclude since a sound conclusion would require more information about their experiment design and further experiments [Rus+18]. In comparison with Choi et al., we concluded that our scores are at least 1 % and up to 7 % higher than theirs [Cho+17].

For RQ4, we concluded that attribution overlays helped us to understand the predictions in many cases while being unhelpful in other cases. In terms of time consumption, their use in an IDE is feasible in general but depends on the chosen attribution technique and the ANN's complexity.

Finally, RQ2 was about the general feasible of vulnerability detection with our novel combination. We concluded that our tool is capable of detecting vulnerabilities in SCS with scores better than the ones from the literature. We identified overfitting to be an issue for small datasets. Based on our pre-experiments, we further concluded that the selection of HP values has high influence on the prediction power.

By evaluating the use of our tool in IDEs, we closed the loop to our introduction. We showed that our tool can function in IDEs due to its flexibility to operate in different FP- and FN-modes and due to its real-time explanatory overlays. In summary, we made a contribution to the reduction of vulnerabilities since our tool allows to interactively supports developers in creating vulnerability-free software.

During elaboration of this thesis, we had ideas for further improvements and future work. The following paragraphs give an overview over these.

One of the first steps we would have taken with more time is to test each tool configuration on this thesis' datasets on which it was *not* trained on. The results of these inter-dataset experiments would provide more insights into our tool's ability to generalize.

To mitigate this thesis' systemic issue of only considering synthetic datasets, our best tool configuration could be evaluated on naturally created SCSs. An example for this is the publicly available source code of LibreOffice[1].

By employing ASTs for feature representation, a feasible next step is static normalization on those. This has the aim to reduce the SCSs overall complexity and to relieve the ANN from resolving aliases, matching function calls, etc.

We assume that the overfitting issues on small, simple datasets can be reduced by not only using early stopping but other regularizaion methods such as dropout or weight decay [Sri+14; Bis06].

Our extrapolation-related experiments showed that achieving extrapolation with NALUs is difficult. Future work could elaborate more on various HPs and their affect on the NALUs' ability to generalize arithmetic operations.

[1]https://www.libreoffice.org/ (retrieved on August 21, 2019)

Lastly, during sequencing of an SCS's AST nodes, the different orders could be further examined. We used the depth-first pre-order, which corresponds to an order used by humans when reading SCS. However, the inverse or a breadth-first order may be beneficial for the ANN. To process a feature sequence in both directions at the same time, the bidirectional LSTM technique by Graves and Schmidhuber could be used [GS05].

In conclusion, we proposed and evaluated the novel combination of AST-based feature representations together with NALU- and LSTM-based RNNs for vulnerability detection. We thoroughly examined the different techniques both theoretically and practically. As a consequence, we reached our main objective and answered all our RQs even though some answers were that sound conclusions require further information or experiments. Based on these reservations, our outlook described experiments and improvements that are subject of future work.

Abbreviations

ANN Artificial Neural Network 2–6, 9, 11–13, 15–17, 20, 21, 23, 26, 29, 30, 34, 38–41, 43, 50, 52, 71, 76–78, 82, 86, 100–102, 109, 113–115, 147

AST Abstract Syntax Tree 3–6, 8, 12, 13, 34, 52–60, 62, 64–68, 70–72, 78, 93, 106, 108, 109, 113–115, 143

AuROC Area under Receiver Operating Characteristic curve 29, 30, 78

BasFR Basic Feature Representation 52, 54, 57, 58, 62–64, 75, 76, 83, 85–98, 102, 109, 110, 113, 136–141, 143–145, 147

CE Cross Entropy 19, 20, 23–26

CWE Common Weakness Enumeration 13, 43–45, 47, 65, 66, 69, 75, 78–80, 84–87, 89–92, 94–99, 101–104, 106, 108, 111, 125–134, 136–142, 144, 147

EmbFR Embedding Feature Representation 52, 54, 60–62, 64, 75, 76, 81, 83, 85–98, 101, 102, 104, 106, 108–110, 113, 136–141, 143–145, 147

FC Fully-Connected 16, 57, 58, 61, 75, 77, 82, 83, 86, 99, 100, 135

FN False Negative 1, 2, 29, 30, 75, 78, 87, 88, 105–108, 114, 145

FNN Feed-forward ANN 15, 16, 19–21, 23, 24, 33–35

FP False Positive 1, 2, 29, 30, 75, 78, 85, 87, 88, 103, 105–108, 114, 145

GD Gradient Descent 24, 25, 118

GradInput Gradient*Input 4, 39–41, 75, 77, 88, 108–110, 113, 142, 143

HP Hyper-Parameter 4, 7, 16, 23, 26–28, 57, 62, 63, 74–79, 81–84, 86–88, 110, 114, 143, 147

IntGrad Integrated Gradients 4, 39–41, 75, 77, 87, 88, 93, 108–110, 113, 142, 143

LSTM Long Short-Term Memory 3, 4, 6, 12, 15, 36–38, 57, 58, 61–63, 74–76, 81, 86, 88, 99, 100, 109, 113, 115, 143

Bibliography

[All+18] Miltiadis Allamanis et al. "A Survey of Machine Learning for Big Code and Naturalness". In: *ACM Computing Surveys* 51.4 (Sept. 2018), 81:1–81:37. DOI: 10.1145/3212695 (cit. on pp. 1, 11).

[Anc+18] Marco Ancona et al. "Towards better understanding of gradient-based attribution methods for Deep Neural Networks". In: *6th International Conference on Learning Representations.* 2018. DOI: 10.3929/ethz-b-000249929 (cit. on p. 4).

[Avi+04] Algirdas Avižienis et al. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2 (cit. on p. 1).

[Bax+98] Ira D. Baxter et al. "Clone Detection Using Abstract Syntax Trees". In: *Proceedings of the International Conference on Software Maintenance.* Nov. 1998, pp. 368–377. DOI: 10.1109/ICSM.1998.738528 (cit. on p. 12).

[Bis+95] Christopher M. Bishop et al. *Neural Networks for Pattern Recognition.* Clarendon Press, 1995. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.679.1104&rep=rep1&type=pdf (cit. on pp. 2, 15, 51).

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Information science and statistics. Springer, 2006. URL: https://www.springer.com/de/book/9780387310732 (cit. on pp. 28, 114).

[Bla18] Paul E. Black. *Juliet 1.3 Test Suite: Changes From 1.2.* US Department of Commerce, National Institute of Standards and Technology, June 2018. DOI: 10.6028/NIST.TN.1995 (cit. on p. 44).

[Blo62] H. D. Block. "The Perceptron: A Model for Brain Functioning. I". In: *Reviews of Modern Physics* 34 (1 Jan. 1962), pp. 123–135. DOI: 10.1103/RevModPhys.34.123 (cit. on p. 18).

[Cho+17] Min-je Choi et al. "End-to-End Prediction of Buffer Overruns from Raw Source Code via Neural Memory Networks". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence.* Aug. 2017, pp. 1546–1553. DOI: 10.24963/ijcai.2017/214 (cit. on pp. 5, 12, 13, 48–50, 52, 62, 78, 82, 101, 104, 105, 114, 118).

[CM04] Brian Chess and Gary McGraw. "Static Analysis for Security". In: *IEEE Security and Privacy* 2.6 (Nov. 2004), pp. 76–79. DOI: 10.1109/MSP.2004.111 (cit. on pp. 1, 2).

[CW08] Ronan Collobert and Jason Weston. "A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning". In: *Proceedings of the 25th International Conference on Machine Learning.* 2008, pp. 160–167. DOI: 10.1145/1390156.1390177 (cit. on p. 11).

[Dam+17] Hoa Khanh Dam et al. "Automatic feature learning for vulnerability prediction". In: *CoRR* abs/1708.02368 (2017). arXiv: 1708.02368 (cit. on pp. 3, 82).

[GB10] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics.* 2010, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html (cit. on p. 26).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* eBook version. MIT Press, 2016. URL: https://www.deeplearningbook.org (cit. on pp. 25, 35, 80, 86).

[GD98] M. W. Gardner and S. R. Dorling. "Artificial Neural Networks (the Multilayer Perceptron) – A Review of Applications in the Atmospheric Sciences". In: *Atmospheric environment* 32.14/15 (1998), pp. 2627–2636. DOI: 10.1016/S1352-2310(97)00447-0 (cit. on p. 15).

[Gho+16] Shalini Ghosh et al. "Contextual LSTM (CLSTM) models for Large scale NLP tasks". In: *CoRR* abs/1602.06291 (2016). arXiv: 1602.06291 (cit. on pp. 3, 62).

[GP15] Katerina Goseva-Popstojanova and Andrei Perhinschi. "On the capability of static code analysis to detect security vulnerabilities". In: *Information and Software Technology* 68 (2015), pp. 18–33. DOI: 10.1016/j.infsof.2015.08.002 (cit. on pp. 1, 2, 5, 44, 78, 79, 84–86, 92, 94, 101–104, 113, 114).

[Gra08] Alex Graves. "Supervised Sequence Labelling with Recurrent Neural Networks". PhD thesis. Universität München, 2008. DOI: 10.1007/978-3-642-24797-2 (cit. on pp. 16, 34, 51).

[GS05] Alex Graves and Jürgen Schmidhuber. "Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures". In: *Neural Networks* 18.5-6 (2005), pp. 602–610. DOI: 10.1016/j.neunet.2005.06.042 (cit. on p. 115).

[GSC99] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to Forget: Continual Prediction with LSTM". In: *Proceedings of the 9th International Conference on Artificial Neural Networks*. IET, Sept. 1999. DOI: 10.1049/cp:19991218 (cit. on p. 36).

[Gup+17] Rahul Gupta et al. "DeepFix: Fixing Common C Language Errors by Deep Learning". In: *Proceedings of the 31th AAAI Conference on Artificial Intelligence*. 2017, pp. 1345–1351. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/viewPaper/14603 (cit. on p. 12).

[Hag+14] Martin T. Hagan et al. *Neural Network Design*. 2nd. eBook version. Martin Hagan, 2014. URL: http://hagan.okstate.edu/nnd.html (cit. on p. 76).

[Har+18] Jacob A. Harer et al. "Automated software vulnerability detection with machine learning". In: *CoRR* abs/1803.04497 (Aug. 2018). arXiv: 1803.04497 (cit. on pp. 2, 56, 60).

[HLZ16] Xuan Huo, Ming Li, and Zhi-Hua Zhou. "Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code". In: *Proceedings of the 25th International Joint Conference on Artificial Intelligence*. 2016, pp. 1606–1612. URL: https://dl.acm.org/citation.cfm?id=3060845 (cit. on p. 11).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735 (cit. on pp. 3, 36).

[Kim+17] Seulbae Kim et al. "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery". In: *IEEE Symposium on Security and Privacy*. May 2017, pp. 595–614. DOI: 10.1109/SP.2017.62 (cit. on p. 12).

[LeC+12] Yann LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade*. Springer, 2012, pp. 9–48. DOI: 10.1007/978-3-642-35289-8_3 (cit. on pp. 25, 26).

[Li+18] Zhen Li et al. "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection". In: *CoRR* abs/1801.01681 (2018). arXiv: 1801.01681 (cit. on pp. 2, 5).

[LJR08] Jorge M. Lobo, Alberto Jiménez-Valverde, and Raimundo Real. "AUC: a misleading measure of the performance of predictive distribution models". In: *Global Ecology and Biogeography* 17.2 (2008), pp. 145–151. DOI: 10.1111/j.1466-8238.2007.00358.x (cit. on p. 30).

[Met78] Charles E. Metz. “Basic Principles of ROC Analysis”. In: *Seminars in Nuclear Medicine* 8.4 (Oct. 1978), pp. 283–298. DOI: https://doi.org/10.1016/S0001-2998(78)80014-2 (cit. on pp. 29, 30).

[Mou+16] Lili Mou et al. “Convolutional Neural Networks over Tree Structures for Programming Language Processing”. In: *Proceedings of the 30th AAAI Conference on Artificial Intelligence.* 2016, pp. 1287–1293. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/viewPaper/11775 (cit. on pp. 2, 5, 13, 51, 52, 56).

[Nat12] Center for Assured Software at the National Security Agency. *Juliet Test Suite v1.2 for C/C++ User Guide.* US Department of Commerce, National Institute of Standards and Technology, Dec. 2012. URL: https://samate.nist.gov/SARD/resources/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf (cit. on pp. 5, 44–47, 68, 70, 110).

[PA17] Ponemon Institute and Accenture. *2017 Cost of Cyber Crime Study.* Report. 2017. URL: https://www.accenture.com/t20170926T072837Z__w__/us-en/_acnmedia/PDF-61/Accenture-2017-CostCyberCrimeStudy.pdf (cit. on p. 1).

[PS18] Michael Pradel and Koushik Sen. “DeepBugs: A Learning Approach to Name-based Bug Detection”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018). DOI: 10.1145/3276517 (cit. on pp. 13, 82).

[Pyl99] Dorian Pyle. *Data Preparation for Data Mining.* Morgan Kaufmann, 1999. Google Books: hhdVr9F-JfAC (cit. on p. 8).

[RCK09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach”. In: *Science of Computer Programming* 74.7 (2009), pp. 470–495. DOI: 10.1016/j.scico.2009.02.007 (cit. on p. 12).

[Roj96] Raúl Rojas. *Neural Networks: A Systematic Introduction.* Springer, 1996. DOI: 10.1007/978-3-642-61068-4 (cit. on p. 25).

[Ros61] Frank Rosenblatt. *Principles of Neurodynamics. Perceptrons and the Theory of Brain Mechanisms.* Cornell Aeronautical Laboratory, Inc., Mar. 1961. URL: https://apps.dtic.mil/docs/citations/AD0256582 (cit. on p. 18).

[Rus+18] Rebecca L. Russell et al. “Automated Vulnerability Detection in Source Code Using Deep Representation Learning”. In: *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications.* Nov. 2018, pp. 757–762. DOI: 10.1109/ICMLA.2018.00120 (cit. on pp. 2–5, 12, 14, 44, 51, 52, 56, 60, 62, 63, 76, 78–80, 82, 101, 102, 104, 105, 111, 114).

[SK16] Abdullah Sheneamer and Jugal Kalita. “Semantic Clone Detection Using Machine Learning”. In: *Proceedings of the 15th IEEE International Conference on Machine Learning and Applications*. Dec. 2016, pp. 1024–1028. DOI: 10.1109/ICMLA.2016.0185 (cit. on p. 12).

[SNS15] Martin Sundermeyer, Hermann Ney, and Ralf Schlüter. “From Feedforward to Recurrent LSTM Neural Networks for Language Modeling”. In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 23.3 (Mar. 2015), pp. 517–529. DOI: 10.1109/TASLP.2015.2400218 (cit. on pp. 3, 62).

[Sri+14] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html (cit. on p. 114).

[SSV18] Carson D. Sestili, William S. Snavely, and Nathan M. VanHoudnos. “Towards security defect prediction with AI”. In: *CoRR* abs/1808.09897 (Sept. 2018). arXiv: 1808.09897 (cit. on pp. 2, 13, 44).

[STY17] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. “Axiomatic Attribution for Deep Networks”. In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70. 2017, pp. 3319–3328. URL: https://dl.acm.org/citation.cfm?id=3306024 (cit. on pp. 38–41, 77).

[SVZ13] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”. In: *arXiv e-prints* (Dec. 2013). arXiv: 1312.6034 (cit. on p. 14).

[Tra+18] Andrew Trask et al. “Neural Arithmetic Logic Units”. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 2018, pp. 8046–8055. URL: https://dl.acm.org/citation.cfm?id=3327899 (cit. on pp. 4, 30, 31, 33, 50, 63, 98, 99).

[WL17] Huihui Wei and Ming Li. “Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code”. In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. 2017, pp. 3034–3040. DOI: 10.24963/ijcai.2017/423 (cit. on pp. 2, 3, 12).

[WLT16] Song Wang, Taiyue Liu, and Lin Tan. “Automatically Learning Semantic Features for Defect Prediction”. In: *Proceeedings of the 38th International Conference on Software Engineering*. May 2016, pp. 297–308. DOI: 10.1145/2884781.2884804 (cit. on pp. 2, 3, 13).

[WZ95] Ronald J. Williams and David Zipser. "Gradient-Based Learning Algorithms for Recurrent Connectionist Networks". In: *Backpropagation: Theory, Architectures, and Applications*. Psychology Press, 1995, pp. 433–486. Google Books: B71nu3LDpREC (cit. on p. 35).

[Zho+16] Bolei Zhou et al. "Learning Deep Features for Discriminative Localization". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. June 2016, pp. 2921–2929. DOI: 10.1109/CVPR.2016.319 (cit. on pp. 13, 14).

[ZS14] Wojciech Zaremba and Ilya Sutskever. "Learning to Execute". In: *arXiv e-prints* (2014). arXiv: 1410.4615 (cit. on pp. 76, 82).

Appendix

Table 8.1: Overview of Juliet CWEs.

CWE ID	CWE Name
15	External Control of System or Configuration Setting
23	Relative Path Traversal
36	Absolute Path Traversal
78	OS Command Injection
90	LDAP Injection
114	Process Control
121	Stack-based Buffer Overflow
122	Heap-based Buffer Overflow
123	Write-what-where Condition
124	Buffer Underwrite
126	Buffer Over-read
127	Buffer Under-read
134	Use of Externally-Controlled Format String
176	Improper Handling of Unicode Encoding
188	Reliance on Data/Memory Layout
190	Integer Overflow or Wraparound
191	Integer Underflow (Wrap or Wraparound)
194	Unexpected Sign Extension
195	Signed to Unsigned Conversion Error
196	Unsigned to Signed Conversion Error
197	Numeric Truncation Error
222	Truncation of Security Relevant Information
223	Omission of Security-relevant Information
226	Sensitive Information Uncleared Before Release
242	Use of Inherently Dangerous Function
244	Heap Inspection

Continued on next page ...

Table 8.1 continued from previous page

CWE ID	CWE Name
247	Reliance on DNS Lookups in Security Decision
252	Unchecked Return Value
253	Incorrect Check of Function Return Value
256	Unprotected Storage of Credentials
259	Use of Hard-coded Password
272	Least Privilege Violation
273	Improper Check for Dropped Privileges
284	Improper Access Control
319	Cleartext Transmission of Sensitive Information
321	Use of Hard-coded Cryptographic Key
325	Missing Required Cryptographic Step
327	Use of a Broken or Risky Cryptographic Algorithm
328	Reversible One-Way Hash
338	Use of Cryptographically Weak PRNG
364	Signal Handler Race Condition
366	Race Condition Within Thread
367	Time-of-check Time-of-use Race Condition
369	Divide by Zero
377	Insecure Temporary File
390	Detection of Error Condition Without Action
391	Unchecked Error Condition
396	Declaration of Catch for Generic Exception
397	Declaration of Throws for Generic Exception
398	Poor Code Quality
400	Uncontrolled Resource Consumption
401	Memory Leak
404	Improper Resource Shutdown or Release
415	Double Free
416	Use After Free
426	Untrusted Search Path
427	Uncontrolled Search Path Element
440	Expected Behavior Violation
457	Use of Uninitialized Variable
459	Incomplete Cleanup
464	Addition of Data Structure Sentinel
467	Use of sizeof() on a Pointer Type
468	Incorrect Pointer Scaling

Continued on next page ...

Table 8.1 continued from previous page

CWE ID	CWE Name
469	Use of Pointer Subtraction to Determine Size
475	Undefined Behavior for Input to API
476	NULL Pointer Dereference
478	Missing Default Case in Switch Statement
479	Signal Handler Use of Non-reentrant Function
480	Use of Incorrect Operator
481	Assigning Instead of Comparing
482	Comparing Instead of Assigning
483	Incorrect Block Delimitation
484	Omitted Break Statement in Switch
500	Public Static Field Not Marked Final
506	Embedded Malicious Code
510	Trapdoor
511	Logic/Time Bomb
526	Information Exposure Through Environmental Variables
534	Info Exposure Debug Log
535	Information Exposure Through Shell Error Message
546	Suspicious Comment
561	Dead Code
562	Return of Stack Variable Address
563	Assignment to Variable without Use
570	Expression Always False
571	Expression is Always True
587	Assignment of a Fixed Address to a Pointer
588	Attempt to Access Child of Non Structure Pointer
590	Free Memory Not on Heap
591	Sensitive Data Storage in Improperly Locked Memory
605	Multiple Binds Same Port
606	Unchecked Input for Loop Condition
615	Information Exposure Through Comments
617	Reachable Assertion
620	Unverified Password Change
665	Improper Initialization
666	Operation on Resource in Wrong Phase of Lifetime
667	Improper Locking
672	Operation on Resource After Expiration or Release
674	Uncontrolled Recursion

Continued on next page ...

Table 8.1 continued from previous page

CWE ID	CWE Name
675	Duplicate Operations on Resource
676	Use of Potentially Dangerous Function
680	Integer Overflow to Buffer Overflow
681	Incorrect Conversion Between Numeric Types
685	Function Call With Incorrect Number of Arguments
688	Function Call With Incorrect Variable or Reference as Argument
690	Unchecked Return Value to NULL Pointer Dereference
758	Reliance on Undefined Behavior
761	Free of Pointer not at Start of Buffer
762	Mismatched Memory Management Routines
773	Missing Reference to Active File Descriptor or Handle
775	Missing Release of File Descriptor or Handle
780	Use of RSA Algorithm Without OAEP
785	Path Manipulation Function Without Max.-sized Buffer
789	Uncontrolled Memory Allocation
832	Unlock of Resource That is Not Locked
835	Infinite Loop
843	Type Confusion

Table 8.2: Statistics of test cases and data preparation for separate CWEs of the Juliet dataset. Parenthesis denote the number of the non-parenthesised test cases that expect a Microsoft Windows Environment. Tables 5.1 and 6.1 contain column-wise aggregated values.

	Test Cases				**Prepared Sequences**						**Prep.**
CWE	**Count (Need Windows)**				**Count**			**Length**			**Time**
ID	**Overall**		**Not Parsable**		**Total**	**Good**	**Bad**	**Min**	**Max**	**Mean**	**[s]**
122	5922	(2052)	0	(0)	**14 574**	8652	5922	14	617	85.05	1451
190	3960	(0)	0	(0)	**14 400**	10 440	3960	14	511	67.03	1407
762	3564	(576)	0	(0)	**12 752**	9188	3564	12	106	37.13	985
121	4944	(1844)	0	(0)	**12 033**	7089	4944	14	533	73.91	1034
191	2952	(0)	0	(0)	**10 788**	7836	2952	14	497	69.35	1061
134	2880	(1680)	73	(73)	**10 101**	7294	2807	19	555	124.39	1658
78	4800	(3840)	1220	(1220)	**8210**	4630	3580	19	392	97.07	789
590	2680	(400)	0	(0)	**6231**	3551	2680	29	157	59.87	471
401	1658	(294)	0	(0)	**5894**	4236	1658	12	223	55.92	539
127	2048	(880)	0	(0)	**5112**	3064	2048	14	487	95.93	531
124	2048	(880)	0	(0)	**5112**	3064	2048	14	533	99.31	533
23	2400	(1440)	346	(346)	**4753**	2699	2054	19	401	103.10	484
36	2400	(1440)	346	(346)	**4753**	2699	2054	19	401	103.10	490
457	948	(22)	0	(0)	**3960**	3012	948	8	247	74.54	421
126	1452	(582)	0	(0)	**3774**	2322	1452	14	487	103.41	405
415	962	(144)	0	(0)	**3444**	2482	962	14	105	36.66	245
789	960	(480)	0	(0)	**3440**	2480	960	14	551	116.39	520
369	864	(0)	0	(0)	**3096**	2232	864	14	467	83.51	347

Continued on next page ...

Table 8.2 continued from previous page

CWE	Test Cases				Prepared Sequences						Prep.
	Count (Need Windows)					Count			Length		Time
ID	Overall		Not Parsable		Total	Good	Bad	Min	Max	Mean	[s]
194	1152	(0)	0	(0)	**2664**	1512	1152	14	352	103.27	254
195	1152	(0)	0	(0)	**2664**	1512	1152	16	332	102.55	255
400	720	(0)	0	(0)	**2580**	1860	720	16	613	120.42	383
690	960	(192)	1	(1)	**2218**	1259	959	14	123	49.15	141
197	864	(0)	0	(0)	**1998**	1134	864	16	288	66.16	141
416	459	(66)	0	(0)	**1891**	1432	459	18	229	57.84	170
253	684	(414)	0	(0)	**1862**	1178	684	12	161	34.54	86
606	480	(240)	1	(1)	**1717**	1238	479	19	610	147.95	311
252	630	(360)	0	(0)	**1715**	1085	630	4	161	29.05	72
758	581	(58)	0	(0)	**1578**	997	581	11	179	48.22	92
563	512	(84)	0	(0)	**1540**	1028	512	4	111	25.84	74
680	576	(0)	0	(0)	**1332**	756	576	16	338	105.67	132
761	576	(288)	1	(1)	**1330**	755	575	39	381	178.36	196
476	348	(42)	0	(0)	**1225**	877	348	14	103	36.43	81
427	480	(240)	5	(5)	**1100**	625	475	19	375	83.23	89
114	576	(576)	134	(134)	**1036**	594	442	19	389	94.52	96
404	384	(336)	30	(30)	**828**	474	354	14	106	45.02	48
90	480	(480)	129	(129)	**824**	473	351	19	511	214.38	146
617	306	(0)	0	(0)	**715**	409	306	15	297	69.53	50
675	192	(48)	0	(0)	**688**	496	192	14	101	41.01	48
272	252	(252)	0	(0)	**686**	434	252	35	173	64.21	44
284	216	(216)	0	(0)	**588**	372	216	40	133	59.88	37

Continued on next page ...

Table 8.2 continued from previous page

	Test Cases				Prepared Sequences						Prep.
CWE	Count (Need Windows)				Count			Length			Time
ID	Overall		Not Parsable		Total	Good	Bad	Min	Max	Mean	[s]
398	181	(0)	0	(0)	**492**	311	181	3	65	25.40	18
665	193	(96)	0	(0)	**446**	253	193	17	127	58.87	30
426	192	(96)	2	(2)	**440**	250	190	19	106	46.37	26
377	144	(90)	18	(18)	**343**	217	126	33	149	66.20	22
775	144	(48)	0	(0)	**333**	189	144	14	105	41.80	18
123	144	(0)	0	(0)	**333**	189	144	65	333	139.37	37
773	144	(48)	0	(0)	**333**	189	144	14	133	65.70	23
506	158	(124)	0	(0)	**313**	155	158	22	591	141.96	27
319	192	(192)	116	(116)	**284**	208	76	26	1100	320.57	100
256	96	(96)	20	(20)	**284**	208	76	71	574	253.84	82
390	90	(18)	0	(0)	**245**	155	90	27	97	45.85	13
546	90	(0)	0	(0)	**245**	155	90	4	20	10.69	6
666	90	(0)	0	(0)	**245**	155	90	248	501	264.65	49
321	96	(96)	0	(0)	**222**	126	96	19	380	261.52	44
259	96	(96)	10	(10)	**202**	116	86	19	171	93.24	18
591	96	(96)	10	(10)	**202**	116	86	14	169	102.07	20
226	72	(72)	0	(0)	**196**	124	72	98	257	125.40	21
244	72	(72)	0	(0)	**196**	124	72	129	347	167.70	29
325	72	(72)	0	(0)	**196**	124	72	148	355	180.87	28
511	72	(18)	0	(0)	**196**	124	72	21	201	49.72	14
588	80	(0)	0	(0)	**186**	106	80	33	105	47.06	11
843	80	(0)	0	(0)	**186**	106	80	26	92	37.63	10

Continued on next page ...

Table 8.2 continued from previous page

CWE	Test Cases				Prepared Sequences						Prep.
	Count (Need Windows)				Count			Length			Time
ID	Overall		Not Parsable		Total	Good	Bad	Min	Max	Mean	[s]
672	47	(0)	0	(0)	**169**	122	47	14	167	86.33	21
327	54	(54)	0	(0)	**147**	93	54	316	637	335.43	39
396	54	(0)	0	(0)	**147**	93	54	48	115	62.06	16
681	54	(0)	0	(0)	**147**	93	54	38	123	57.67	9
391	54	(0)	0	(0)	**147**	93	54	18	125	48.67	7
328	54	(54)	0	(0)	**147**	93	54	318	644	338.61	59
467	54	(0)	0	(0)	**147**	93	54	46	101	55.76	8
176	48	(48)	10	(10)	**142**	104	38	24	124	67.61	14
510	70	(0)	0	(0)	**132**	62	70	97	421	187.11	17
464	48	(0)	0	(0)	**111**	63	48	14	137	65.08	7
15	48	(48)	0	(0)	**111**	63	48	19	327	145.09	12
468	37	(0)	0	(0)	**100**	63	37	19	78	42.51	4
535	36	(36)	0	(0)	**98**	62	36	103	212	114.10	10
188	36	(0)	0	(0)	**98**	62	36	20	81	38.03	4
366	36	(0)	0	(0)	**98**	62	36	88	217	108.44	12
367	36	(0)	0	(0)	**98**	62	36	110	246	127.82	10
475	36	(18)	0	(0)	**98**	62	36	34	73	41.92	5
459	36	(18)	0	(0)	**98**	62	36	49	115	61.27	6
534	36	(36)	0	(0)	**98**	62	36	119	244	130.76	11
273	36	(36)	0	(0)	**98**	62	36	4	217	65.34	6
469	36	(18)	0	(0)	**98**	62	36	42	94	52.08	5
397	20	(1)	0	(0)	**53**	33	20	5	23	12.62	1

Continued on next page ...

Table 8.2 continued from previous page

	Test Cases				Prepared Sequences						Prep.
CWE	**Count (Need Windows)**				**Count**			**Length**			**Time**
ID	**Overall**		**Not Parsable**		**Total**	**Good**	**Bad**	**Min**	**Max**	**Mean**	**[s]**
483	20	(0)	0	(0)	**53**	33	20	36	78	43.89	2
620	18	(18)	0	(0)	**49**	31	18	74	167	88.27	3
832	18	(0)	0	(0)	**49**	31	18	45	111	58.76	3
688	18	(0)	0	(0)	**49**	31	18	18	41	25.27	1
222	18	(18)	0	(0)	**49**	31	18	281	585	305.61	10
615	18	(18)	0	(0)	**49**	31	18	41	87	49.20	2
685	18	(0)	0	(0)	**49**	31	18	13	33	20.73	1
478	18	(0)	0	(0)	**49**	31	18	40	91	50.18	2
479	18	(0)	0	(0)	**49**	31	18	5	21	11.73	1
667	18	(0)	0	(0)	**49**	31	18	37	111	55.82	3
605	18	(0)	0	(0)	**49**	31	18	248	527	274.20	9
364	18	(0)	0	(0)	**49**	31	18	86	197	102.78	5
480	18	(0)	0	(0)	**49**	31	18	12	52	32.78	2
587	18	(0)	0	(0)	**49**	31	18	13	41	23.43	1
223	18	(18)	0	(0)	**49**	31	18	286	579	304.88	18
676	18	(0)	0	(0)	**49**	31	18	22	61	33.47	3
785	18	(18)	0	(0)	**49**	31	18	47	119	62.80	3
247	18	(18)	0	(0)	**49**	31	18	300	619	323.49	11
196	18	(0)	0	(0)	**49**	31	18	44	113	59.06	2
481	18	(0)	0	(0)	**49**	31	18	17	39	24.22	1
780	18	(18)	0	(0)	**49**	31	18	226	457	241.76	9
242	18	(0)	0	(0)	**49**	31	18	43	95	52.63	10

Continued on next page ...

Table 8.2 continued from previous page

CWE	Test Cases				Prepared Sequences						Prep.
	Count (Need Windows)				Count			Length			Time
ID	Overall		Not Parsable		Total	Good	Bad	Min	Max	Mean	[s]
526	18	(0)	0	(0)	**49**	31	18	4	23	11.80	1
484	18	(0)	0	(0)	**49**	31	18	43	93	51.96	2
338	18	(18)	0	(0)	**49**	31	18	11	123	50.31	2
482	18	(0)	0	(0)	**49**	31	18	17	39	24.22	1
571	16	(0)	0	(0)	**32**	16	16	7	69	14.84	0
570	16	(0)	0	(0)	**32**	16	16	7	69	14.84	0
835	6	(0)	0	(0)	**13**	7	6	14	36	24.38	0
562	3	(0)	0	(0)	**6**	3	3	14	26	20.00	0
674	2	(0)	0	(0)	**4**	2	2	6	35	20.75	0
561	2	(0)	0	(0)	**4**	2	2	4	11	6.25	0
500	1	(0)	0	(0)	**2**	1	1	4	4	4.00	0
440	1	(0)	0	(0)	**2**	1	1	7	7	7.00	0

Table 8.3: Pre-experiments results. For each of the 16 tool configurations, we performed one pre-experiment on the MemNet dataset. We repeated each pre-experiment's training three times. Each best F_1 denotes the repetition-wise best tool configuration's validation F_1. Figure 7.1 provides a visual representation.

FC Neuron	**Activation**	**Training**	**Learning**			**Validation F_1**			
Count	**Function**	**Algorithm**	**Rate**	**Best**	**Mean**	**SD**	**Rep. 1**	**Rep. 2**	**Rep. 3**
4	Sigmoid	Adam	0.001	**0.9099**	0.9033	0.0047	0.9005	0.9099	0.8996
32	ReLU	RMSprop	0.001	**0.9094**	0.9027	0.0074	0.9064	0.9094	0.8923
4	ReLU	RMSprop	0.001	**0.9086**	0.9013	0.0054	0.8996	0.8958	0.9086
4	ReLU	Adam	0.001	**0.9077**	0.7700	0.1910	0.9024	0.9077	0.5000
32	Sigmoid	Adam	0.001	**0.9071**	0.9014	0.0041	0.8979	0.8991	0.9071
32	ReLU	Adam	0.001	**0.9068**	0.9055	0.0009	0.9049	0.9048	0.9068
32	Sigmoid	RMSprop	0.001	**0.9019**	0.8961	0.0051	0.8895	0.8969	0.9019
4	Sigmoid	RMSprop	0.001	**0.9019**	0.8995	0.0031	0.9017	0.8951	0.9019
32	Sigmoid	Adam	0.1	**0.6207**	0.5665	0.0404	0.5235	0.6207	0.5552
4	Sigmoid	Adam	0.1	**0.6049**	0.5843	0.0210	0.5555	0.6049	0.5924
32	Sigmoid	RMSprop	0.1	**0.5709**	0.5627	0.0060	0.5565	0.5607	0.5709
32	ReLU	RMSprop	0.1	**0.5485**	0.5240	0.0198	0.5235	0.5485	0.5000
4	Sigmoid	RMSprop	0.1	**0.5387**	0.5197	0.0137	0.5136	0.5069	0.5387
32	ReLU	Adam	0.1	**0.5000**	0.5000	0.0000	0.5000	0.5000	0.5000
4	ReLU	RMSprop	0.1	**0.5000**	0.5000	0.0000	0.5000	0.5000	0.5000
4	ReLU	Adam	0.1	**0.5000**	0.5000	0.0000	0.5000	0.5000	0.5000

Table 8.4: Results of E1 to E112 and E113 to E224 for BasFR and EmbFR, respectively. See section 7.2 for an description of the best, mean, and SD columns. We column-wise applied min, max, mean, and SD aggregation functions to all 112 CWES and to the 20 feature-sequence-count-wise biggest CWEs. The aggregated numbers are shown in the two segments at the end of this table. We do not list and aggregate test mean and SD values for many CWEs since they are not available due to a bug in our implementation.

CWE ID / Aggregation	**BasFR F_1**						**EmbFR F_1**					
	Validation			**Test**			**Validation**			**Test**		
	Best	**Mean**	**SD**	**Best**	**Mean**	**SD**	**Best**	**Mean**	**SD**	**Best**	**Mean**	**SD**
15	1.0000	1.0000	0.0000	**0.9383**			1.0000	0.9605	0.0558	**0.6807**		
23	0.9727	0.8429	0.1836	**0.9674**			0.9927	0.9909	0.0026	**0.9837**		
36	0.9583	0.8503	0.1220	**0.9673**			0.9873	0.9818	0.0065	**0.4011**	0.4510	0.2107
78	0.9822	0.9519	0.0423	**0.9805**			0.9873	0.9830	0.0030	**0.2734**	0.3222	0.0846
90	0.9599	0.9493	0.0075	**0.9432**			0.9839	0.9812	0.0039	**0.3748**		
114	0.9745	0.9616	0.0105	**0.9549**			0.9936	0.9482	0.0322	**0.7753**		
121	0.9047	0.7617	0.1911	**0.8696**			0.7892	0.7327	0.0444	**0.4363**		
122	0.9757	0.9717	0.0041	**0.9702**	0.9642	0.0071	0.9739	0.9728	0.0008	**0.9628**		
123	0.9600	0.8934	0.0680	**0.9385**			0.9800	0.9667	0.0094	**0.5158**		
124	0.9765	0.9172	0.0421	**0.9778**			0.9857	0.9597	0.0305	**0.4537**	0.4538	0.0001
126	0.6996	0.6043	0.0881	**0.6305**	0.5579	0.0994	0.9594	0.8352	0.0916	**0.5042**	0.3246	0.1555
127	0.7240	0.5905	0.0985	**0.6903**			0.9883	0.9766	0.0147	**0.4938**		
134	0.7908	0.7288	0.0799	**0.7818**			0.9722	0.9592	0.0092	**0.6546**		
176	0.8678	0.7155	0.1322	**0.8190**			0.8678	0.8303	0.0314	**0.6589**		
188	0.8000	0.7548	0.0320	**0.7868**			1.0000	0.9327	0.0549	**0.2571**		
190	0.9327	0.7727	0.1173	**0.9259**	0.7629	0.1448	0.9715	0.9699	0.0020	**0.1284**		

Continued on next page ...

Table 8.4 continued from previous page

CWE ID / Aggregation	BasFR F$_1$						EmbFR F$_1$					
	Validation			Test			Validation			Test		
	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD
191	0.9712	0.9447	0.0290	**0.9656**			0.9755	0.9737	0.0021	**0.9747**		
194	0.9004	0.7212	0.2046	**0.9024**	0.7072	0.2551	0.9900	0.9707	0.0273	**0.9724**		
195	0.9775	0.9756	0.0012	**0.9675**	0.9666	0.0063	0.9850	0.9790	0.0052	**0.9675**	0.9658	0.0014
196	0.8770	0.8770	0.0000	**0.8398**	0.8059	0.0802	0.8770	0.8317	0.0579	**0.5952**		
197	0.9666	0.9413	0.0313	**0.9561**			0.9867	0.9833	0.0027	**0.2690**		
222	1.0000	1.0000	0.0000	**0.7143**			1.0000	0.9167	0.1179	**0.6714**		
223	0.8590	0.8226	0.0514	**0.7262**			1.0000	0.9212	0.0557	**0.1270**		
226	1.0000	1.0000	0.0000	**0.9656**			1.0000	1.0000	0.0000	**0.2775**		
242	0.8682	0.8288	0.0557	**0.7571**			1.0000	0.9590	0.0580	**0.0357**		
244	1.0000	0.8687	0.1074	**1.0000**	0.9428	0.0714	1.0000	1.0000	0.0000	**0.2422**		
247	1.0000	1.0000	0.0000	**0.7143**			1.0000	1.0000	0.0000	**0.2571**		
252	0.9845	0.9702	0.0175	**0.9454**			1.0000	0.9948	0.0073	**1.0000**		
253	0.8949	0.7682	0.1276	**0.8933**	0.7535	0.1536	0.9964	0.9820	0.0157	**0.9964**		
256	1.0000	0.9510	0.0539	**0.9088**			1.0000	0.9225	0.1096	**0.6030**		
259	1.0000	0.8970	0.1238	**1.0000**			1.0000	1.0000	0.0000	**0.2970**		
272	1.0000	0.9872	0.0181	**0.9902**			1.0000	1.0000	0.0000	**0.5113**		
273	0.9333	0.7967	0.0966	**0.9307**			1.0000	1.0000	0.0000	**0.0952**		
284	0.8666	0.7472	0.0845	**0.8443**			1.0000	0.9228	0.1091	**0.1835**		
319	0.8243	0.7741	0.0365	**0.7986**			1.0000	0.8667	0.1017	**0.3251**	0.4928	0.1537
321	0.9412	0.8732	0.0765	**0.6083**			1.0000	0.9120	0.0720	**0.2227**		
325	0.7982	0.7643	0.0453	**0.3569**			0.7487	0.7267	0.0200	**0.2086**		
327	0.9565	0.9124	0.0624	**0.9553**	0.6881	0.4628	0.9565	0.9565	0.0000	**0.1536**		

Continued on next page ...

Table 8.4 continued from previous page

CWE ID /	BasFR F_1						EmbFR F_1					
Aggregation	Validation			Test			Validation			Test		
	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD
328	1.0000	0.9689	0.0440	**0.7054**			1.0000	0.9851	0.0211	**0.3333**		
338	0.8770	0.8741	0.0041	**0.7143**			1.0000	1.0000	0.0000	**0.4156**		
364	0.7500	0.7361	0.0196	**1.0000**			0.7500	0.7103	0.0561	**1.0000**		
366	1.0000	0.9578	0.0299	**0.8601**			1.0000	0.9789	0.0299	**0.5031**		
367	0.9333	0.9099	0.0332	**0.8571**			1.0000	0.9556	0.0314	**0.3333**	0.3333	0.0000
369	0.8438	0.8094	0.0378	**0.8473**			0.9533	0.9364	0.0194	**0.9195**		
377	0.8471	0.7384	0.0773	**0.8042**			0.8851	0.8269	0.0420	**0.2803**		
390	0.9729	0.8644	0.1173	**1.0000**			1.0000	1.0000	0.0000	**1.0000**		
391	0.9165	0.8865	0.0358	**0.9091**			1.0000	0.9858	0.0200	**0.1169**		
396	0.9120	0.8837	0.0200	**0.8681**			0.8696	0.8253	0.0365	**0.6488**		
397	1.0000	0.9561	0.0621	**0.8571**			1.0000	1.0000	0.0000	**0.3429**		
398	0.8681	0.7818	0.0781	**0.8106**	0.7204	0.1891	1.0000	0.9688	0.0227	**0.5294**		
400	0.7882	0.6985	0.0638	**0.7586**			0.9616	0.9297	0.0398	**0.6224**	0.6442	0.0331
401	0.9588	0.8490	0.0937	**0.9576**			0.9732	0.9687	0.0055	**0.6523**	0.6300	0.0198
404	0.7930	0.7611	0.0345	**0.7264**			0.9361	0.9016	0.0247	**0.3884**	0.3526	0.0620
415	0.8561	0.7713	0.0882	**0.8508**			0.9578	0.9579	0.0032	**0.3612**		
416	0.9859	0.8939	0.1180	**0.9789**	0.8861	0.1397	1.0000	0.9814	0.0157	**0.5316**	0.2987	0.2148
426	0.9672	0.9563	0.0155	**0.9170**			1.0000	0.9891	0.0154	**0.5520**		
427	0.9934	0.8727	0.1570	**0.9934**			0.9934	0.9891	0.0062	**0.9867**		
457	0.7450	0.7329	0.0095	**0.6594**			0.9830	0.9818	0.0008	**0.1941**		
459	0.9333	0.9331	0.0003	**0.8601**			1.0000	0.9778	0.0314	**0.1880**	0.2930	0.1819
464	0.9416	0.9221	0.0275	**0.6814**			1.0000	1.0000	0.0000	**1.0000**		

Continued on next page ...

Table 8.4 continued from previous page

CWE ID /	BasFR F_1						EmbFR F_1					
Aggregation	Validation			Test			Validation			Test		
	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD
467	0.7938	0.7735	0.0245	**0.8468**	0.6364	0.1913	0.8726	0.8732	0.0667	**0.6124**		
468	1.0000	0.8000	0.2828	**0.7467**			1.0000	0.9782	0.0308	**0.1123**		
469	0.9366	0.9169	0.0280	**0.8571**			0.8773	0.7766	0.1040	**0.4156**		
475	1.0000	0.9564	0.0308	**1.0000**			1.0000	0.9782	0.0308	**0.0536**		
476	0.9302	0.8207	0.1317	**0.8629**			0.9729	0.9655	0.0069	**0.6830**		
478	0.8682	0.7894	0.0557	**0.5714**			0.8770	0.8347	0.0599	**0.0714**		
479	0.8682	0.7100	0.1118	**0.5524**			0.7500	0.6706	0.0561	**0.2857**		
480	1.0000	0.8697	0.1023	**0.8571**			1.0000	0.9530	0.0665	**0.8571**		
481	1.0000	1.0000	0.0000	**0.8745**			1.0000	0.8757	0.1021	**0.0357**		
482	1.0000	0.8333	0.1179	**0.7143**			0.7500	0.7500	0.0000	**0.2571**		
483	0.7500	0.7361	0.0196	**0.8508**			1.0000	0.9180	0.0580	**0.2571**		
484	1.0000	0.9153	0.0599	**0.8745**			0.8730	0.7910	0.0580	**0.0357**	0.0357	0.0000
506	0.9573	0.9146	0.0462	**0.9564**			1.0000	1.0000	0.0000	**0.3333**		
510	0.9499	0.7970	0.1098	**0.7823**			0.8496	0.8324	0.0243	**0.3630**		
511	0.9673	0.9458	0.0294	**0.8596**			1.0000	1.0000	0.0000	**0.2422**		
526	1.0000	0.9561	0.0621	**0.7143**			1.0000	1.0000	0.0000	**0.2571**		
534	0.9339	0.8874	0.0645	**0.8542**			1.0000	0.9089	0.0645	**0.3333**	0.3222	0.0192
535	1.0000	0.9329	0.0549	**0.8601**			1.0000	1.0000	0.0000	**0.1880**		
546	0.7082	0.6720	0.0256	**0.6572**	0.6768	0.0489	0.7297	0.6808	0.0407	**0.0808**		
563	0.8967	0.8854	0.0129	**0.8596**			0.9261	0.9157	0.0117	**0.5386**	0.5322	0.0273
570	1.0000	0.8413	0.1122	**1.0000**	0.8794	0.1045	1.0000	0.8148	0.2619	**0.7667**	0.8444	0.1347
571	1.0000	0.9333	0.0943	**0.7667**			1.0000	1.0000	0.0000	**0.1000**		

Continued on next page ...

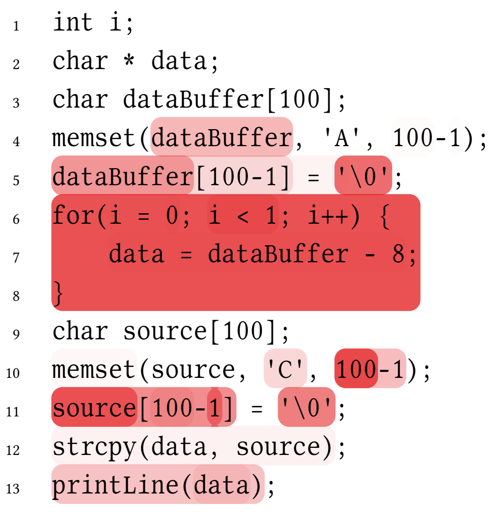
Table 8.4 continued from previous page

CWE ID /	BasFR F_1						EmbFR F_1					
Aggregation	Validation			Test			Validation			Test		
	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD
587	0.8910	0.8248	0.0468	**0.8398**			1.0000	0.8942	0.0851	**0.5952**		
588	0.8265	0.8028	0.0316	**0.6915**	0.5568	0.1490	1.0000	0.9067	0.1319	**0.3968**		
590	1.0000	1.0000	0.0000	**1.0000**			0.9979	0.9993	0.0010	**0.9968**		
591	0.8085	0.7489	0.0758	**0.6000**			0.9680	0.8724	0.0688	**0.2970**		
605	1.0000	1.0000	0.0000	**0.6286**			1.0000	1.0000	0.0000	**0.0357**		
606	0.9578	0.8819	0.0466	**0.9576**	0.8863	0.0618	0.8873	0.8608	0.0041	**0.8586**	0.8510	0.0074
615	1.0000	0.9590	0.0580	**0.3810**			1.0000	0.8269	0.2448	**0.1270**		
617	0.9077	0.8586	0.0632	**0.8498**			0.9815	0.9476	0.0264	**0.4477**		
620	1.0000	1.0000	0.0000	**0.5143**	0.6455	0.1801	1.0000	1.0000	0.0000	**0.4156**	0.4156	0.0000
665	1.0000	0.7433	0.2148	**0.8635**			1.0000	1.0000	0.0000	**0.3503**		
666	1.0000	0.9910	0.0127	**1.0000**			1.0000	1.0000	0.0000	**0.2451**		
667	0.8770	0.8347	0.0599	**0.8635**			1.0000	0.9151	0.0602	**0.3810**		
672	0.7516	0.7319	0.0280	**0.5651**			0.7516	0.7136	0.0504	**0.1906**		
675	0.9077	0.8936	0.0099	**0.9146**			0.9430	0.9330	0.0086	**0.6454**		
676	0.8770	0.8347	0.0599	**0.7024**			0.8770	0.8347	0.0599	**0.2571**		
680	0.9800	0.9568	0.0184	**0.9799**			0.9900	0.9834	0.0047	**0.4325**		
681	0.9565	0.9274	0.0206	**1.0000**			0.9565	0.9420	0.0205	**1.0000**		
685	1.0000	0.9561	0.0621	**0.8635**			1.0000	1.0000	0.0000	**0.5195**		
688	0.8730	0.7474	0.1037	**1.0000**			1.0000	0.9577	0.0599	**0.2571**		
690	0.7095	0.6827	0.0190	**0.7116**			0.9401	0.9381	0.0028	**0.6740**		
758	1.0000	0.9890	0.0155	**1.0000**	0.9915	0.0147	1.0000	0.9781	0.0155	**0.9873**	0.9760	0.0129
761	0.7724	0.6977	0.0531	**0.7245**			0.9543	0.9309	0.0297	**0.8010**		

Continued on next page ...

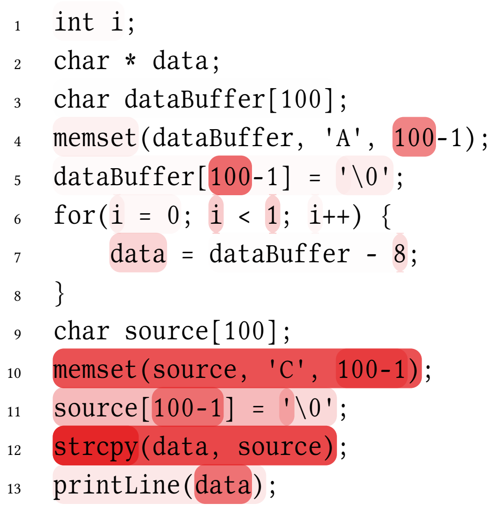
Table 8.4 continued from previous page

CWE ID / Aggregation		BasFR F_1						EmbFR F_1					
		Validation			Test			Validation			Test		
		Best	Mean	SD	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD
	762	0.7811	0.7271	0.0471	**0.7431**			0.9029	0.8998	0.0027	**0.6592**	0.4865	0.2702
	773	0.6562	0.5911	0.0460	**0.6478**	0.6404	0.1109	0.6779	0.6090	0.0499	**0.1270**	0.1270	0.0000
	775	0.9002	0.9002	0.0000	**0.8359**	0.5452	0.2578	0.9200	0.9268	0.0094	**0.3447**		
	780	0.8730	0.8265	0.0658	**0.2571**			0.8730	0.8265	0.0658	**0.2571**		
	785	1.0000	0.9136	0.0618	**0.8571**			1.0000	0.9606	0.0557	**0.4156**		
	789	0.9163	0.8914	0.0185	**0.8809**			0.9693	0.8585	0.1529	**0.9538**		
	832	1.0000	0.9561	0.0621	**0.8398**	0.7662	0.1485	1.0000	1.0000	0.0000	**0.7262**		
	843	1.0000	0.9880	0.0170	**0.8133**			1.0000	1.0000	0.0000	**0.3130**		
all 112 CWEs	Min	0.6562	0.5905	0.0000	**0.2571**	0.5452	0.0063	0.6779	0.6090	0.0000	**0.0357**	0.0357	0.0000
	Max	1.0000	1.0000	0.2828	**1.0000**	0.9915	0.4628	1.0000	1.0000	0.2619	**1.0000**	0.9760	0.2702
	Mean	0.9175	0.8588	0.0581	**0.8299**	0.7610	0.1370	0.9579	0.9257	0.0342	**0.4645**	0.4835	0.0757
	SD	0.0874	0.1014	0.0517	**0.1466**	0.1427	0.1039	0.0723	0.0857	0.0451	**0.2879**	0.2561	0.0884
20 biggest CWEs	Min	0.6996	0.5905	0.0000	**0.6305**	0.5579	0.0063	0.7892	0.7327	0.0008	**0.1284**	0.3222	0.0001
	Max	1.0000	1.0000	0.2046	**1.0000**	0.9666	0.2551	0.9979	0.9993	0.1529	**0.9968**	0.9658	0.2702
	Mean	0.8934	0.8207	0.0749	**0.8768**	0.7918	0.1025	0.9648	0.9444	0.0213	**0.6472**	0.5191	0.1060
	SD	0.0969	0.1194	0.0638	**0.1166**	0.1754	0.1042	0.0462	0.0657	0.0378	**0.2992**	0.2230	0.1084



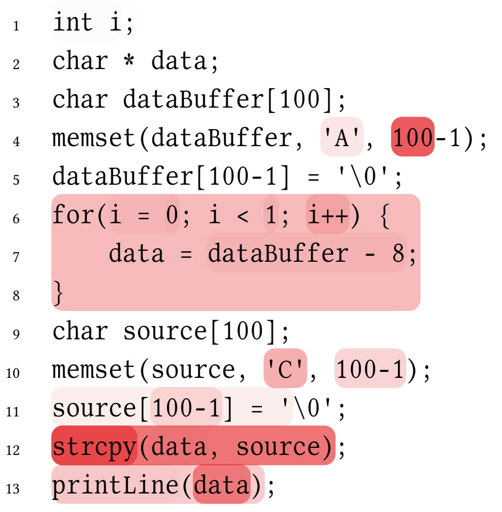
```
int i;
char * data;
char dataBuffer[100];
memset(dataBuffer, 'A', 100-1);
dataBuffer[100-1] = '\0';
for(i = 0; i < 1; i++) {
    data = dataBuffer - 8;
}
char source[100];
memset(source, 'C', 100-1);
source[100-1] = '\0';
strcpy(data, source);
printLine(data);
```

(a) GradInput.



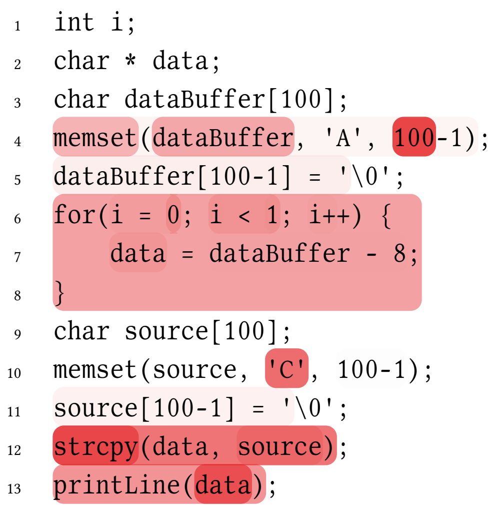
```
int i;
char * data;
char dataBuffer[100];
memset(dataBuffer, 'A', 100-1);
dataBuffer[100-1] = '\0';
for(i = 0; i < 1; i++) {
    data = dataBuffer - 8;
}
char source[100];
memset(source, 'C', 100-1);
source[100-1] = '\0';
strcpy(data, source);
printLine(data);
```

(b) IntGrad20.



```
int i;
char * data;
char dataBuffer[100];
memset(dataBuffer, 'A', 100-1);
dataBuffer[100-1] = '\0';
for(i = 0; i < 1; i++) {
    data = dataBuffer - 8;
}
char source[100];
memset(source, 'C', 100-1);
source[100-1] = '\0';
strcpy(data, source);
printLine(data);
```

(c) IntGrad100.



```
int i;
char * data;
char dataBuffer[100];
memset(dataBuffer, 'A', 100-1);
dataBuffer[100-1] = '\0';
for(i = 0; i < 1; i++) {
    data = dataBuffer - 8;
}
char source[100];
memset(source, 'C', 100-1);
source[100-1] = '\0';
strcpy(data, source);
printLine(data);
```

(d) IntGrad300.

Listing 8.1: Results of E240 to E243 visualizing different attribution techniques' overlays on a TP. The overlay opacity and color correspond to the positive contribution to the output neuron for CWE 124.

List of Figures

List of Listings

List of Tables