Master Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Vulnerability Assessment of Smart Contracts using Explainable AI Methods

Alexander Hefter
Department of Computer Science
Chair of Computer Science II (Secure Software Systems)

Prof. Dr. Alexandra Dmitrienko
First Reviewer

Prof. Dr. Samuel Kounev
Second Reviewer

Submission
02. January 2023

www.uni-wuerzburg.de

Abstract

Smart contracts are small computer programs stored on a blockchain and play an important role in many cryptocurrencies, such as Ethereum. The code of a smart contract represents a sort of digital agreement between its users. However, like all computer programs, smart contracts possibly contain security vulnerabilities an attacker can exploit to achieve goals beyond the original intent of the contract. Therefore, tools have been developed to identify smart contracts with vulnerabilities at source code or bytecode level. Some of these detection methods make use of deep neural networks, a machine learning technique. However, these methods only indicate whether a smart contract contains a certain vulnerability or not, but a programmer is also interested in its location in the code. The main problem is here that it is not obvious which parts of the source code are responsible for the prediction result of a neural network.

This work describes two methods that determine whether a smart contract is vulnerable and find the locations of the vulnerabilities in the source code. Both methods are built on graph neural networks and, therefore, convert source code files into contract graphs. The first method identifies vulnerable contract graphs by graph classification. The vulnerability positions are figured out with an explainer, a program that determines the parts of the input graph that are responsible for the graph classification result. The second method is based on node classification. This method requires a dataset of source codes where the exact vulnerability positions are given by line numbers. Therefore, a dataset regarding the reentrancy vulnerability has been labeled by hand for this thesis. Both methods are tested and evaluated on this dataset.

Although the first method reliably detects vulnerable contracts with a f1-score of up to 97.12%, the f1-score for locating the vulnerabilities does not exceed 9.0%. However, the second method determines vulnerable contracts with a f1-score of up to 93.02% and figures out the positions of the vulnerabilities with a f1-score of up to 93.69%. Moreover, the second method analyzes source codes in less than three seconds. Both methods can handle contract source codes of any lengths. The output of the analysis is in both cases given by the line numbers that contain the vulnerabilities.

Zusammenfassung

Smart Contracts sind kleine Computerprogramme, die auf der Blockchain gespeichert werden, und eine wichtige Rolle in vielen Kryptowährungen wie Ethereum spielen. Der Code eines Smart Contracts stellt eine Art digitale Vereinbarung zwischen seinen Nutzern dar. Jedoch können Smart Contracts, wie andere Computerprogramme auch, Sicherheitslücken aufweisen, die ein Angreifer nutzen kann um Ziele zu erreichen, die weit vom ursprünglichen Verwendungszweck abweichen. Deshalb sind Programme entwickelt worden um Smart Contracts, die Schwachstellen beinhalten, entweder auf Quellcode- oder auf Bytecodeebene zu identifizieren. Einige dieser Detektionsprogramme nutzen tiefe Neuronale Netze, eine Technik des Maschinellen Lernens. Jedoch können derartige Verfahren lediglich angeben, ob ein Smart Contract eine Schwachstelle besitzt oder nicht. Ein Programmierer ist aber auch an der Stelle im Quellcode interessiert. Das Hauptproblem besteht aber nicht zuletzt darin, dass im Allgemeinen nicht klar ist, welcher Teil des Quellcodes verantwortlich für das Klassifizierungsergebnis des Neuronales Netzwerkes ist.

In dieser Arbeit werden zwei Verfahren beschrieben, die feststellen ob ein Smart Contract Sicherheitslücken aufweist und darüber hinaus die Stellen der Sicherheitslücken im Quellcode finden. Beide Verfahren basieren auf neuronalen Graphen Netzwerken und konvertieren daher Quellcode Dateien in Contract Graphen. Die erste Methode wendet ein Graphklassifizierungsverfahren auf den Smart Contract Graphen an, um Smart Contracts mit Sicherheitslücken zu finden. Die Positionen der Schwachstellen im Quellcode werden dann mit Hilfe eines Explainers ermittelt, der die Teile des Graphen bestimmt, die für das Ergebnis der Graphklassifikation verantwortlich sind. Die zweite Methode basiert auf einem Knotenklassifizierungsverfahren und benötigt einen Datensatz von Quellcodes bei dem die exakten Positionen der Schwachstellen mit Hilfe von Zeilenangaben verfügbar ist. Deshalb besteht auch ein Teil dieser Arbeit darin, einen Datensatz hinsichtlich der Reentrancy Schwachstelle zeilengenau zu markieren. Beide Verfahren werden anhand dieses Datensatzes getestet und bewertet.

Obwohl das erste Verfahren Smart Contracts, die Sicherheitslücken aufweisen, mit einem F1-Score von bis zu 97.12% findet, werden diese lediglich mit einem F1-Score von höchstens 9.0% lokalisiert. Das zweite Verfahren identifiziert Smart Contracts mit Schwachstellen mit einem F1-Score von bis zu 93.02% und stellt die Positionen der Schwachstellen im Quellcode mit einem F1-Score von bis zu 93.69% fest. Darüber hinaus analysiert dieses Verfahren die Quellcodes in weniger als drei Sekunden. Beide Verfahren können Smart Contract Quellcodes von beliebiger Länge analysieren. Das Ergebnis dieser Analysen wird bei beiden Verfahren anhand von Zeilennummern an den Nutzer zurückgegeben.

Acknowledgement

I was supported by many people while writing my thesis. First of all, I would like to thank my supervisor Prof. Dr. Alexandra Dmitrienko for giving me the opportunity to work on this interesting and challenging topic of computer science. My special thanks goes to Christoph Sendner who supported me in our weekly meetings with his knowledge and experience. In addition, I would like to thank Lipin Christhudas Lalitha for helping me to label this large amount of smart contract source codes. Moreover, I would like to thank Huili Chen from University of California and Dr. Hossein Fereidooni from University of Darmstadt for their ideas and support in the early stages of this thesis.

Contents

1 Introduction **1**

2 Background **3**
2.1 Machine Learning . . . 3
2.1.1 Linear Classification and Multilayer Perceptron . . . 3
2.1.2 Optimization with Backpropagation . . . 6
2.1.3 Graph Neural Networks . . . 9
2.2 Cryptocurrencies . . . 11
2.2.1 Blockchain . . . 11
2.2.2 Smart Contracts . . . 12
2.2.3 Reentrancy and other Vulnerability Types . . . 13
2.3 Abstract Syntax Tree of Solc . . . 15

3 Related Work **17**
3.1 Smart Contract Security . . . 17
3.2 Source Code Graphs . . . 19
3.3 Methods for Explaining Machine Learning Models . . . 20

4 Approach **23**
4.1 Problem Statement and Goals . . . 23
4.2 Solidity Code Graph . . . 24
4.2.1 Edges of the Graph . . . 24
4.2.2 Node Feature Vectors . . . 29
4.3 Vulnerability Localization with Machine Learning . . . 33
4.3.1 Graph Classification with Explanation . . . 34
4.3.2 Node Classification . . . 35

5 Implementation **37**
5.1 Software Prerequisites . . . 37
5.2 Program Architecture . . . 38
5.3 Preprocessing Step 1 - Abstract Syntax Tree . . . 40
5.4 Preprocessing Step 2 - Graph Generation . . . 41
5.5 Graph Neural Network and Explainer . . . 42
5.6 Line and Node Label Conversion . . . 43

6 Evaluation **45**
6.1 Metrics for Evaluation . . . 45
6.2 Dataset and Hardware . . . 47
6.3 Evaluation of the Graph Classification Approach . . . 48
6.4 Evaluation of the Node Classification Approach . . . 52
6.5 Time Consumption Test . . . 57
6.6 Result Comparison with Methods from Literature . . . 58
6.7 Prediction Examples . . . 59

7 Conclusion **65**

List of Figures **67**

List of Tables **69**

Listings **71**

Acronyms **73**

Bibliography **75**

1. Introduction

In the coming years the digital transformation will play an increasingly important role in the modern society. While on the one hand cashless payments are already part of everyday life for many people, on the other hand cryptocurrencies, such as Bitcoin or Ethereum, are becoming more and more established as a form of financial investment and also as means of payment. In contrast to traditional currencies, cryptocurrencies avoid the necessity of banks, since transactions are performed by an open community on a publicly accessible blockchain. In addition, some cryptocurrencies such as Ethereum offer the possibility to create digital versions of contracts, so-called *smart contracts*. This allows complex transactions like credit agreements or auctions to be conducted. Nevertheless, smart contracts are just small computer programs and, therefore, can contain security holes, too. The exploitation of these bugs offer attackers opportunities to achieve goals beyond the original intent of the contract.

The detection of these vulnerabilities is a well studied topic in literature. Methods like *symbolic execution* [93, 101, 100] or *fuzzing* [68, 148, 56] have been adapted to find vulnerable smart contracts. Even *machine learning* and *neural networks* are employed to scan smart contracts for bugs [162, 156, 178, 87]. This is performed by treating the detection process as a *classification problem*: As neural networks learn from a large amount of examples, a high number of sample contracts are labeled by occurence or absence of a certain vulnerability, for instance. During the training process machine learning schemes learn patterns from these samples to distinguish both groups from each other. Afterwards, the patterns are used to classify previously unseen contracts and, therefore, to identify smart contracts containing this vulnerability. The clear advantage of machine learning over other static detection methods is that the detection patterns are not hardcoded and can easily be updated and improved by adding new training samples. However, although vulnerable contracts are detected by such methods, it is not evident on which parts of the code this prediction is based or, in other words, where the vulnerability is located inside the contract. The scenario provided below depicts this problem:

Alice is a big fan of the cryptocurrency Ethereum. After creating an account and buying some Ether she also wants to try out the capabilities of smart contracts. Therefore, she learns to program with *Solidity*, Ethereum's language for smart contracts, and writes several hundreds lines of code. During her studies she read about vulnerabilities and their possible consequences. So, she decided to scan her source code with a vulnerability detection tool. This tool identifies a certain vulnerability, but does not provide the position of it. After a short search without success Alice thinks of a false alarm and deploys the

contract on the blockchain. For further processing she also deposits several Ether with it. A few days later Alice recognizes that the deposited Ether is gone. In fact, a hacker named Eve has found the detected security hole, successfully has exploited the vulnerability, and has stolen the money.

Contributions of this work. The aim of this master thesis is to provide two methods based on machine learning that are able to find the line positions of vulnerabilities in smart contract source codes. The approaches are based on graph neural networks, more exactly on Graph Convolutional Networks (GCNs) [75], and work either by graph classification with explainer or by node classification. Thus, these methods include the construction of a new type of source code graph for the programming language Solidity. In addition, to test both methods a dataset of 13,773 smart contract source codes has been line labeled for the reentrancy vulnerability. This is necessary, since currently no representative line labeled dataset regarding Solidity source code vulnerabilities exists. However, there are two detection methods based on machine learning that also provide the locations of the vulnerabilities as prediction result. SCScan [60] uses a *support vector machine* (SVM) to detect vulnerable contracts. In a second step, a pattern matching algorithm determines the positions of the vulnerability in the vulnerable contracts. The second step is independent of the first one, which means that the found locations of the vulnerability are not necessarily the reason for the SVM to classify the contract as vulnerable. In contrast to SCScan, both methods presented in this work only provide vulnerability locations that either are the classification result or are directly derived from the classification result. Moreover, the input of SVMs must have a fixed size in contrast to the input of graph neural network which are used in this thesis. In a parallel work, Mando [103] has been developed. Mando is based on a graph neural network approach and consists of two phases, similar to SCScan. In the first phase, vulnerable contracts are detected by graph classification. In the second phase, the lines containing the vulnerability are determined by node classification. Again, both phases work mainly independent of each other and, thus, Mando is much different to the methods provided in this work. Moreover, the heterogeneous graphs that Mando uses as input are combinations of control flow graphs and call graphs. Therefore, unlike the methods presented in this thesis, Mando does not take into account the data dependencies between variables or the hierarchy between different code parts. Especially, graphs that consider data dependencies have been proved as suitable input form for vulnerability detection [162]. In addition to SCScan and Mando, there exist a few methods based on symbolic execution that can predict the locations of the vulnerability such as Mythril [101] and Oyente [93]. However, these methods use fixed vulnerability detection patterns that cannot be updated easily in contrast to the patterns of machine learning methods.

Outline. This work is structured as follows. The necessary background information is provided in Chapter 2. More precisely, these are details about blockchains, smart contracts, the structure and optimization of graph neural networks, and a description of an *Abstract Syntax Tree* (AST) of Solidity. Chapter 3 surveys the related work and gives an overview on existing tools for vulnerability detection in smart contracts, on existing graph types for source codes, and on available explainers for neural networks. Chapter 4 discusses the problem this work is devoted to and presents the two methods that are developed to solve this problem. This includes the construction of a Solidity code graph. Chapter 5 considers the implementation of the methods in Python and PyTorch Geometric. Chapter 6 presents the evaluation of both methods based on a hand-labeled dataset for the Solidity source code and reentrancy vulnerability. This Chapter also provides details about this dataset. Afterwards, the thesis concludes in Chapter 7.

2. Background

This master thesis is about the localization of software vulnerabilities in smart contract source code with graph neural networks. Therefore, this chapter describes and explains the necessary background information in machine learning, blockchains, and smart contracts. Moreover, an Abstract Syntax Tree (AST) of Solidity is considered.

2.1 Machine Learning

Machine learning is the field in which computer systems learn from data to perform certain tasks or actions. For the localization of vulnerabilities in this thesis, graph neural networks are applied. The following subsections give a short overview on machine learning and this special type of neural network. At first, the working of linear classification and *Multilayer Perceptrons* is presented, since both are parts of most graph neural networks, too. The subsequent Section 2.1.2 explains how neural networks are optimized to certain data or, in other words, how neural networks learn from data. The last part of this chapter describes graph neural networks by a general setting. Although there are many different approaches for this topic, there are some parts most of the existing graph neural network types have in common.

2.1.1 Linear Classification and Multilayer Perceptron

One of the tasks that are performed by machine learning algorithms is *classification* (see [11]). Given a set of data and a number of classes the algorithm shall assign a certain class or multiple classes to each element of the set. The classes are numbered and the correct class number of a data element is called *label*. Data are for example images, text, or even source code. In supervised learning, the machine learning algorithm is fed with a high amount of labeled data, the *training set*. From these data it learns patterns to make a generalized connection between data and labels. After training, the algorithm is able to assign the correct labels to before unseen data.

A simple model for the classification with only two classes is the *perceptron* [116, 95, 41]. Figure 2.1 shows the structure of the perceptron. The input data are in this case vectors. Perceptrons work as follows: In a first step, each dimension of the input vector is multiplied by a randomly initialized weight. In a second step, the output is summed up and a random scalar w_0, called *bias*, is added. The predicted label of the vector is then determined by a

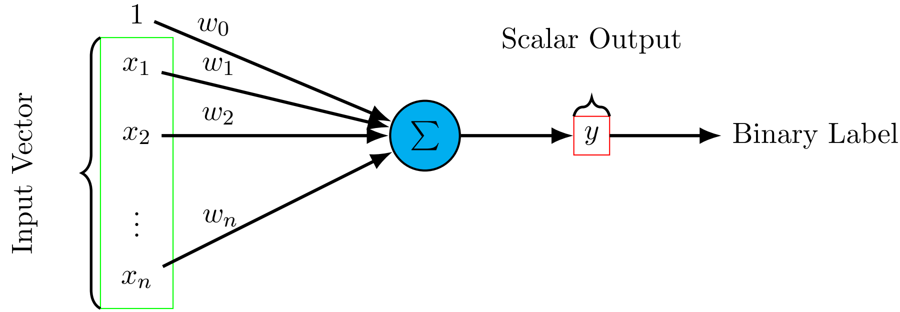


Figure 2.1: Functionality of a perceptron: An input vector is multiplied by weights and the output is generated by summing the results and a bias w_0. Then the label is determined by a threshold.

threshold: If the result is greater or equal than the threshold, the predicted label is one, otherwise it is zero. Therefore, the label of vector $x = (x_1, x_2, ..., x_n)$ is calculated as

$$\text{label}(x) = \begin{cases} 1 & \text{if } w_0 + x_1 * w_1 + x_2 * w_2 + ... + x_n * w_n \geq \text{threshold} \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

Clearly, in most cases the predicted class is not the correct class. So, the weights and the bias have to be adjusted in such a way that this is the case for as many vectors as possible. In fact, this weight optimization procedure is the learning process. It is described more precisely in Section 2.1.2.

The perceptron only offers the possibility to distinguish between two different classes. However, there are tasks with datasets having more than two classes, for example, when separating the vectors by their directions. In this case, more than one perceptron and additional weights are needed to predict the label. In *linear classification* [11, 170, 19], a number of perceptrons are arranged as a layer as Figure 2.2 shows. The concept is similar to before: Each of perceptrons generate the output by a weighted sum of the dimensions of input vector x and a bias. The equivalent linear equation

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} w_{10} \\ w_{20} \\ \vdots \\ w_{m0} \end{pmatrix} \tag{2.2}$$

allows to compute the output vector y efficiently with matrix operations. If the data can have more than one label, the labels are again determined line by line by a threshold. If only one class is assigned to each vector, the predicted label is the index of the largest output. In both cases, the number of perceptrons m equals the number of classes that are possible.

Linear classification is restricted to predict the labels of data that are linear separable (see *Chapter 6.1* of [64]). Thus, for most applications a more complex network design is necessary. A simple idea is to use more than one layer of perceptrons, where the input of each additional layer is output of the layer before. However, this does not change the situation: Setting W_1, W_2 as weight matrices and b_1,b_2 as bias the resulting output vector y for such a two layer network with input x is

$$y = W_2 * (W_1 * x + b_1) + b_2 = (W_2 * W_1) * x + (W_2 * b_1 + b_2) \tag{2.3}$$

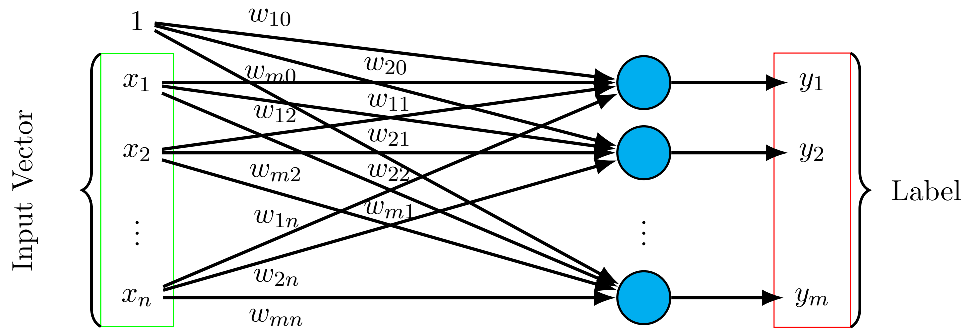


Figure 2.2: Linear Classification: Multiple perceptrons are used in a layer. The number of perceptrons coincides with the number m of possible classes.

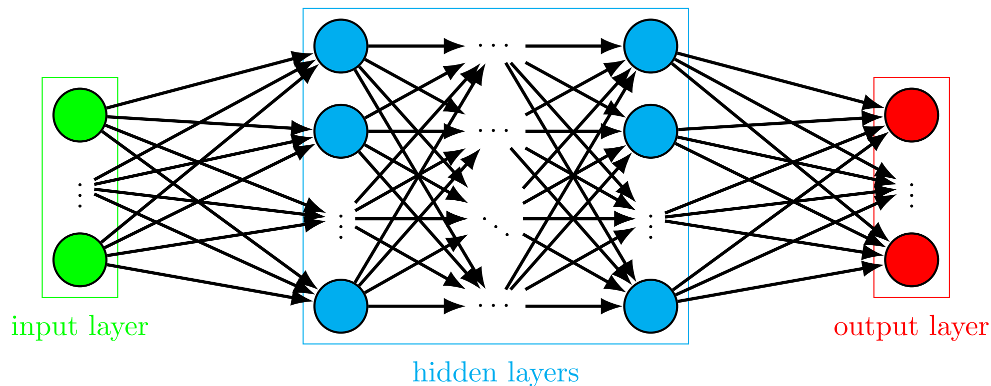


Figure 2.3: A fully connected MLP

This is still a linear equation, since weights and biases are randomly initialized and not constant during the learning process. To overcome this problem and to offer the ability to classify nonlinear connected data, an additional nonlinear function has to be applied before the output of a layer is forwarded to the next one. The function that is used here is called *activation function*. Common activation functions are the *hyperbolic tangent*[115], *sigmoid* [115], and the *rectified linear unit* (ReLU) [44, 115]. They are defined as

$$\text{sigmoid}(x) := \frac{1}{1+e^{-x}}, \quad \text{ReLU}(x) := \max(0,x), \quad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.4}$$

The output of sigmoid lies always between zero and one. ReLU leaves positive numbers unchanged and sets negative ones to zero. With activation function $activation(\cdot)$, weight matrix W_i, and bias b_i the output y_i of the i-th layer is now computed from its input x_i by

$$y_i = activation(W_i * x_i + b_i) \tag{2.5}$$

Having input dimension of n and output dimension of m, W_i has dimensions of $m \times n$ and b_i of $m \times 1$. Similar to a pipeline, the layers are stacked so that the output of one layer is the input of the next one. The neural network that arises by this procedure is called *Multilayer Perceptron* (MLP) [64, 11] (see Figure 2.3) and the perceptrons of the MLP are its *neurons*.

A MLP consists of three parts: the *input layer*, the *hidden layers*, and the *output layer*. The input layer is in this case just an input vector. In the case of graph neural networks, it is a graph. In some cases, the data are not given in the correct input format. Since

a computer cannot understand, for example, source code directly, the dataset has to be transformed by preprocessing steps in such a way that the data fit to the input layer format.

The hidden layers contain the forwarding network with the stacked layers except of the last layer. The number of layers and neurons inside the layers has to be found empirically, because it depends strongly on the task and data. Such structural parameters are called *hyperparameters*. The optimization of the hyperparameters is part of an iterative learning process. At the end of this process the neural network model is evaluated on a part of the dataset that has not been used before.

Thus, the dataset has to be divided into three subsets: the *training*, the *validation*, and the *test set*. The network weights are optimized with the data of the training set. The validation set is needed for a continuous monitoring of the performance on unseen data and for finding optimal hyperparameters. The test set is the part of the dataset that is only used at the end for the final performance test.

The last layer of the network is the output layer. The number of neurons of this layer is determined by the number of classes that are necessary for the task. Moreover, the activation function of this layer may differ from the ones used at the layers before. The reason for this is that the output shall be scaled on probability values between zero and one to determine the predicted label. Thus, depending on the task common activation function are sigmoid for multilabel classification and *Softmax* for single label classification. Having the output vector $y = (y_1, y_2, ..., y_m)^\intercal$ the Softmax function is defined as

$$\text{Softmax}(y)_i = \frac{e^{y_i}}{\sum_{j=1}^{m} e^{y_j}} \tag{2.6}$$

The outputs of the Softmax function sum to one and form a probability distribution. Therefore, each output is interpreted as the probability of one class and the predicted label is given by the class index with the highest probability. Sigmoid activations scale each of the outputs between zero and one and, thus, provide directly the probability for each class. The predicted classes are here determined by thresholds.

As stated above, the weights of neural networks are initialized randomly. However, to support and accelerate the learning process initialization functions such as the Glorot initializer [49] or the He initializer [62] have been developed to offer good starting points for the optimization procedure. The next section describes how this optimization is performed.

2.1.2 Optimization with Backpropagation

Neural networks are able to assign classes to data of a dataset. The class label is determined with the help of randomly initialized parameters. However, since in general random parameters lead to a bad classification performance, the parameters have to be adjusted in a learning or optimization phase in such a way that the network predicts the correct classes for the data. The way in which this optimization method works is now briefly explained below. More detailed versions are given in *Chapter 6 and 7* of [64], in *Chapter 2* of [95], and also in *Module 1* of [11].

As discussed in Section 2.1.1, the output of a neural network is in the case of classification tasks a probability value for each of the n possible classes. The task is to optimize the learnable parameters of the neural network in such a way that this probability distribution reflects the true distribution of the data in the dataset regarding the possible classes. This means a *maximum likelihood estimation* (see also *Chapter 19.7* of [173]) shall be performed. The true distribution is known, if the dataset has been labeled before, since then the probability for a correct class is one and the probabilities for wrong classes are zero. Thus,

the target of the optimization is to minimize or maximize a function that connects the true probabilities of the classes with the ones predicted by the neural network. This function is called *cost function* or also *loss function* [64, 67]. In a maximum likelihood estimation the loss function is given by the cross entropy loss function

$$L_{CE} = -\sum_{i=1}^{n} \hat{y}_i \log(y_i) \tag{2.7}$$

where $\hat{y}_i$ is the true and y_i the predicted probability value of the i-th class for a certain data. The cross entropy provides the error between the true and predicted probability distribution. This error has to be minimized by an update of the learnable parameters. However, for a correct update towards the minimum the partial derivatives of the loss function regarding the learnable parameters are needed.

The partial derivatives or the gradient of a function are efficiently computed by the backpropagation algorithm [32]. The backpropagation algorithm is built on the chain rule of calculus. The idea is the following: Having two differentiable functions $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$ and their composition $h(x) = g(f(x)) =: g(y) =: z$, the derivative of h with respect to x is computed as

$$\frac{dh(x)}{dx} = \frac{dz}{dx} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x} = g'(f(x)) \cdot f'(x) \tag{2.8}$$

So, the derivative of h only depends on the derivative of g at the point $f(x)$ and on the derivative of f at the point x. Since the computation path of neural networks is just a chaining of functions, the partial derivatives with respect to a certain input are obtained by a successive application of the chain rule after the computation of the forward path of the network. The following example explains how this is done.

Example for backpropagation. Given the real valued function $g(x) = b \cdot \log(ax)$, the partial derivative of g shall be computed for the input $x = 2$. During the next steps the parameters a and b may be fixed at $a = 2$ and $b = 3$. The function g is a composition of the three functions

$$g_1(x) := ax = y, \; g_2(y) := \log(y) = z, \; g_3(z) := bz = g(x) \tag{2.9}$$

and can be written as $g(x) = g_3(g_2(g_1(x)))$. The derivatives of these functions are $g_1'(x) = a$, $g_2'(y) = \frac{1}{y}$, and $g_3'(z) = b$. At first, the forwarding path for all steps in the calculation is computed where $y = g_1(2) = 4$, $z = g_2(g_1(2)) = \log(4) \approx 1.39$, and $g(2) = g_3(\log(4)) \approx 1.39 \cdot 3 = 4.17$. To obtain the numerical derivation of g at the point $x = 2$ one has to apply two times the chain rule and obtains

$$\frac{dg}{dx}(2) = \frac{\partial g_3}{\partial z}(g_2(g_1(2))) \cdot \frac{\partial g_2}{\partial y}(g_1(2)) \cdot \frac{\partial g_1}{\partial x}(2) = 3 \cdot 0.25 \cdot 2 = 1.5 \tag{2.10}$$

where $g_2(g_1(2))$ and $g_1(2)$ have already been evaluated during the forwarding path. Now, let a and b be learnable. This means that the numerical derivatives of g with respect to a and b at the point $a = 2$, $b = 3$, and input $x = 2$ have to be computed for a later update. Thus, the function $g(x, a, b)$ depends now on the three variables x, a, and b. With $g_1(a, x) = ax$, $g_2(y) = \log(y)$, and $g_3(b, z) = bz$ one obtains by a successive use of the multivariable chainrule (see for example [146])

$$\frac{dg}{db}(2, 2, 3) = \frac{\partial g_3}{\partial b}(3, g_2(g_1(2, 2))) = \log(4) \approx 1.39 \tag{2.11}$$

$$\frac{dg}{da}(2, 2, 3) = \frac{\partial g_3}{\partial z}(3, g_2(g_1(2, 2))) \cdot \frac{\partial g_2}{\partial y}(g_1(2, 2)) \cdot \frac{\partial g_1}{\partial a}(2, 2) = 3 \cdot 0.25 \cdot 2 = 1.5 \tag{2.12}$$

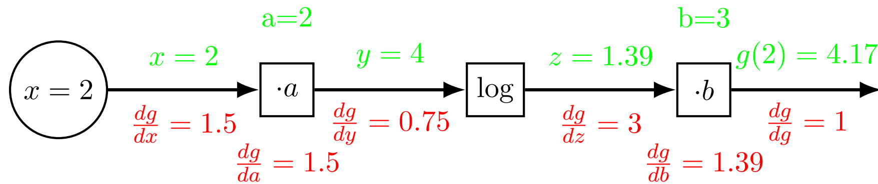


Figure 2.4: Computation Graph: Forward propagation in green, backpropagation in red; derivatives are computed with the last backpropagated value and the last forward propagated one, for example $\frac{dg}{dy} = \frac{\partial \log(y)}{\partial y}\Big|_{y=4} \cdot \frac{dg}{dz}$

Evidently, most of the values are reused, since $\frac{\partial g_3}{\partial z}(3, g_2(g_1(2,2))) \cdot \frac{\partial g_2}{\partial y}(g_1(2,2))$ is identical to the term $\frac{\partial g_3}{\partial z}(g_2(g_1(2))) \cdot \frac{\partial g_2}{\partial y}(g_1(2))$ before. Therefore, only the last derivative has to be evaluated if the results of the calculations are saved and the gradient is computed from the last applied function to the first.

This backpropagation of the gradient of g is illustrated in a computation graph in Figure 2.4. Each node in the graph defines one function or in this case operation. The results of the steps during the forwarding path are written in green above the outgoing arrows. If a function uses any learnable parameters their current values are written in green above the corresponding node. It has to be noticed that necessary analytical derivatives of a function can be already computed at its definition. Therefore, the first step for obtaining partial derivatives of the learnable parameters or the input is to evaluate the corresponding analytical derivative at the values of the forward path that are denoted above the incoming arrows. In a second step the derivative is obtained by multiplying the result with the last backpropagated derivative. Moreover, the backpropagation has to be continued in the same manner from node to node until the derivatives of all learnable parameters are determined. The derivatives regarding these parameters are written in red under the node and the towards the input backpropated derivatives under the arrows.

In neural networks, learnable parameters have more than one dimension, but their gradients are derived in the same manner with backpropagation as above. In addition to this, the nodes of the computation graph are functions that describe the layers and the last applied function is the cost function. More information about backpropagation in neural networks are provided in *Chapter 2* of [95] and *Chapter 6* of [64].

Optimization. With the gradient of the learnable parameters at hand it is possible to adjust them in the correct direction. The methods that are applied for the parameter update with help of the gradients are called *optimizers*. One of the simplest optimizer is the stochastic gradient descent (SGD) [25]

$$w_{k+1} = w_k - \lambda \nabla L_{CE}(w_k) \tag{2.13}$$

where w_k is the value of parameter w at iteration k and $\nabla L_{CE}(w_k)$ the gradient of the cross entropy loss with respect to w_k. By this update w_k is shifted towards its negative gradient which reduces the error computed by the loss function. The hyperparameter λ is called *learning rate* and indicates the intensity of this shift. Other optimizers such as Adam [74], Adagrad [33], or RMSprop [144] improve the parameter updates in such a way that the predictions of the neural network converge faster towards the true probability distribution.

The optimization of the neural network is an iterative process, since after the update not only the parameters but also the gradients change. Therefore, loss and gradients

have to be computed again for the next iteration step. However, after a few steps it can be noticed that the loss decreases and the neural network predicts the correct classes more accurately. To reduce an overfitting to the training data and support the ability for generalized predictions regularization [112] and dropout [131] can be applied. By regularization the L1 or L2 norm of the learnable parameters is added to the loss function. Then the optimization procedure keeps the learnable parameters small which reduces the prediction error. Dropout means that with a certain probability the outputs of the neurons are set to zero during the training phase. Therefore, the network has to learn a more robust model, since it is trained to predict the correct labels in a noisy process.

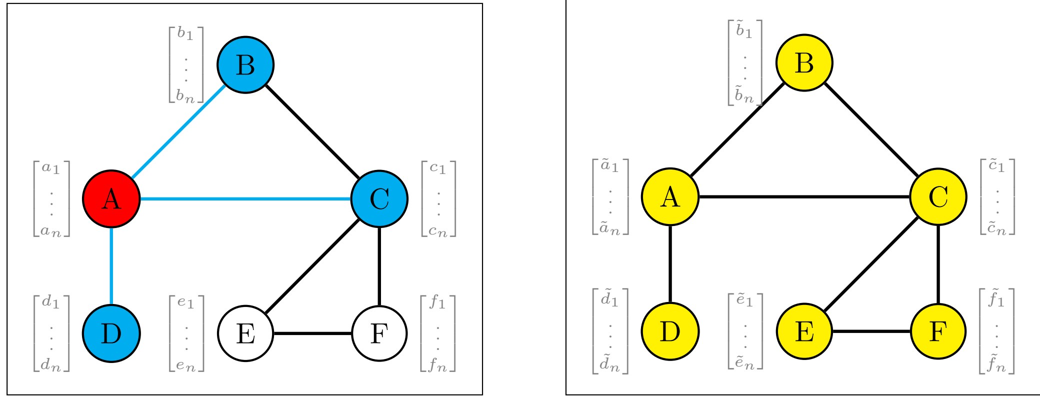
2.1.3 Graph Neural Networks

In many applications, the most suitable form of the input data are graphs. As MLPs are optimized for vector input another sort of neural networks is needed to handle graph input, a *Graph Neural Network* (GNN). An introduction to the theory of graphs can be found in the relevant literature such as [160]. The description of GNNs in this section is based on [165] and also on *Chapter 4* [136] *and 9* [99] of the book [163]. Moreover, the Stanford lecture [132, 72] provides a detailed introduction to the topic.

A graph $G = (V, E)$ consists of a set of nodes V and a set of edges E, also called links. An edge $e = (v_s, v_e)$ is determined by its start node v_s and its end node v_e and connects these two nodes with each other. Edges are either directed or undirected that means there is a connection only in one or in both directions. The connections between the nodes are deposited in the adjacency matrix where a nonzero number in row i and column j means that there is a connection from node i to node j. If there exists a sequence of edges $e_1 = (v_1, v_2)$,$e_2 = (v_2, v_3)$,...,$e_{n-1} = (v_{n-1}, v_n)$ for a number of nodes v_1,...,v_n, there is a path between v_1 and v_n. A graph is called *connected* if there is a path from any node to any other node in the graph. Otherwise the graph is called *disconnected*. Moreover, for each node $v \in V$ there exists a node feature vector X_v that describes the properties of the node. In general, node feature vectors are not unique that means two different nodes have identical node feature vectors if their properties are the same.

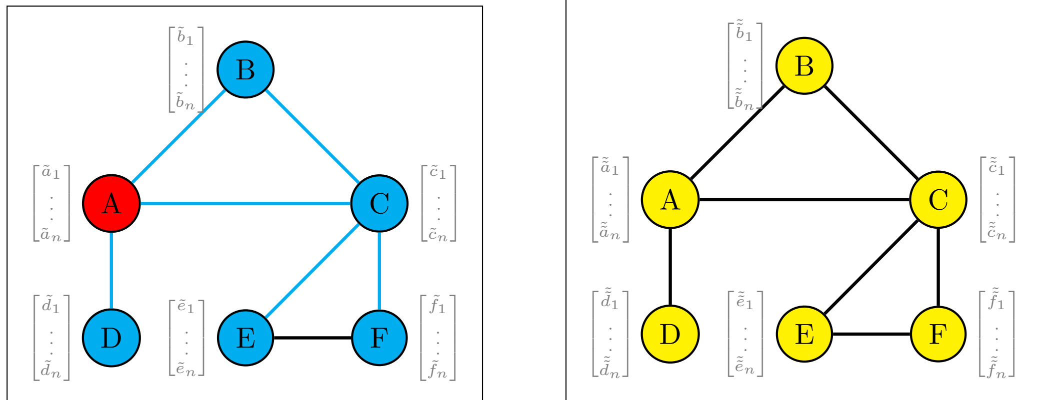
Nodes and graphs can be equipped with labels. Node labels indicate the membership of nodes to certain groups inside the graph. In a social network graph for example where nodes are people and the edges are defined as acquaintances among the people, node labels could indicate the membership of persons in certain clubs or sport teams. Similar to this, graph labels describe the belonging of graphs to particular groups. This leads to two main tasks that are performed with graphs, *node classification* and *graph classification*. Node classification means to reconstruct the node labels with the node feature vectors as input. In graph classification a feature vector that describes the graph properties is used for the label reconstruction. Usually, this graph feature vector is a combination of the node feature vectors such as the sum of all vectors. A classification network for node and graph classification is for example a MLP.

However, as the feature vectors for classification tasks have to be as informative as possible GNNs have been invented. The aim of GNNs is to adjust the feature vectors of the nodes so that they include informations about the nodes and the structure of their neighborhood that are important for a certain task. Similar to MLPs, GNNs consist of layers with learnable parameters that are optimized by backpropagation. Although many different approaches for GNN layers have been developed, there are two steps all schemes have in common, the *aggregate* and the *combine* step. Aggregate means that all nodes of the graph collect information about their neighbors and in the second step these informations are combined with the information about the corresponding node.



(a) Aggregate step: A collects informations about its neighbors

(b) First update of the node feature vectors after the combine step for all nodes



(c) Aggregate step: A collects information from its neighbors about nodes that are up to two hops away

(d) Second update of the node feature vectors after another combine step for all nodes

Figure 2.5: Illustration of the aggregation and combine steps of a two-layer GNN

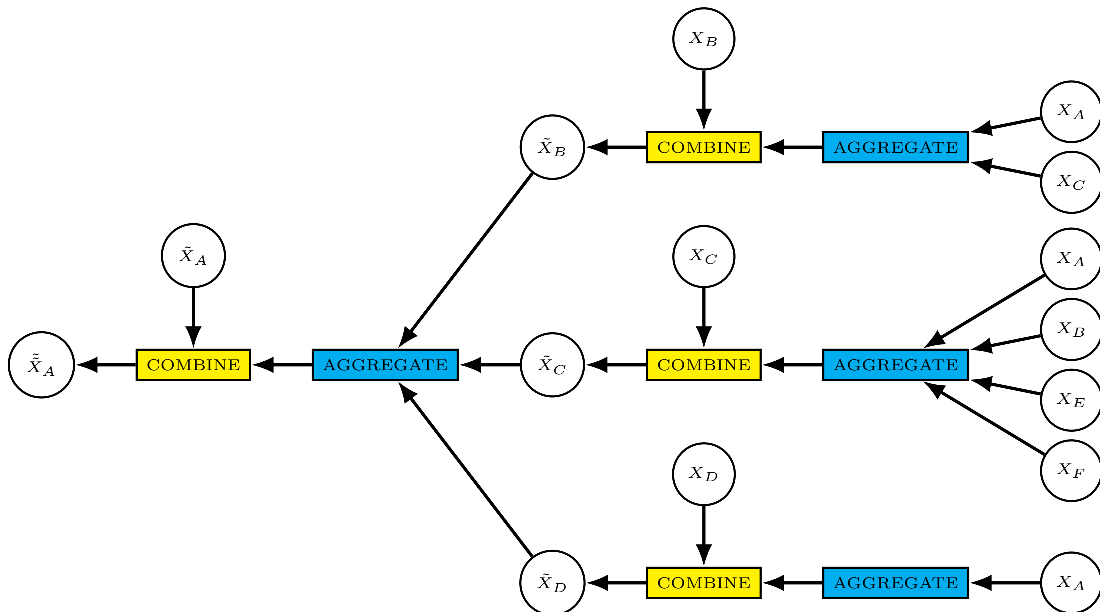


Figure 2.6: Feature vectors that are used in each computation step of Figure 2.5 at node A

The general approach of GNN layers is illustrated in Figure 2.5 at the example of a graph with six nodes and a two layer GNN. The first Figure 2.5(a) shows the aggregation step for node A. The neighboring nodes send their feature vectors to A, where they are joined to one feature vector, since, in general, it is not known how many neighbors a node has. This joining can be done, for example, by a summing or weighted averaging. Simultaneously, all the other nodes receive the feature vector informations from their neighbors. In the second step, presented in Figure 2.5(b) the feature vectors of all nodes are updated by combining their previous feature vectors with the corresponding neighbor feature vectors created in the step before. Both steps use learnable weights and their values are shared among the calculations at all nodes. Afterwards, the graph with the updated vectors is fed into the second layer and another aggregation (Figure 2.5(c)) and a combine step (Figure 2.5(d)) are performed. Since weights are only shared among the layers, during steps 2.5(c) and 2.5(d) other weights are used than in step 2.5(a) and 2.5(b). Moreover, it should be noticed that the feature vectors of the neighboring nodes in the aggregation step 2.5(c) now contain information about their own neigbors. Therefore, each additional layer extends the range of a GNN by one hop. This means that the final feature vector of a certain node includes information about all the nodes that are reached with m hops if m is the number of GNN layers. Thus, if there are enough layers all final feature vectors contain information about the complete graph.

Figure 2.6 presents which vectors take part at the computation steps of Figure 2.5 at node A. It can be noticed that if there is high number of hops between A and another node, the feature vector of this node is less often used in the calculations than feature vectors of nodes that are closer to A. Thus, the influence of nodes on A decreases with the number hops which reflects in some way the structure of the graph.

An example for a GNN is the *Graph Convolutional Network* (GCN) [75]. For a graph with N nodes, feature vector matrix X, and adjacency matrix M a layer of this network type is defined as

$$\tilde{X} = \text{ReLU}(\tilde{D}^{-\frac{1}{2}}(M + I_N)\tilde{D}^{-\frac{1}{2}}XW) \tag{2.14}$$

where I_N is the identity matrix of rank N, and W a matrix with learnable weights. The diagonal degree matrix $\tilde{D}$ is given by $\tilde{D}_{ii} = \sum_j M_{ij}$. Equivalent to this, according to [165] the output feature vector $\tilde{X}_v$ of one layer at a single node v with neighborhood $N(v)$ is computed by

$$\tilde{X}_v = \text{ReLU}(W \cdot \text{MEAN}\{X_u \forall u \in N(v) \cup \{v\}\}) \tag{2.15}$$

where MEAN denotes the elementwise mean pooling function. The aggregation step is clearly identified by the mean pooling part, although it integrates parts of the combine step by taking X_v into account. The nonlinear ReLU function enables the application of more than one layer as discussed in Section 2.1.1. An extensive list of other GNN layer types can be found in the documentation of PyTorch Geometric (PyG) [141] at *Section torch_geometric.nn* where also the corresponding papers are linked. In this work, the GNN structures are built on GCN layers.

2.2 Cryptocurrencies

Smart Contracts are strongly connected with blockchains and, therefore, cryptocurrencies. This section gives an introduction about blockchains and smart contracts. Furthermore, vulnerabilities of smart contracts are discussed.

2.2.1 Blockchain

Bitcoin [102] is known as the first worldwide successful cryptocurrency. Cryptocurrencies are characterized by the fact that no banks or other third parties are involved in money

transactions from one person to another. Therefore, there has to be a way to get informed about whether a transaction is performed correctly or not. This problem has been solved by a publicly available *blockchain*. A blockchain consists of blocks that are chained together by their blockhashes. Each block contains a few transactions, the hash of previous block in the blockchain, and some other values. The hash of the previous block ensures the ordering of the blocks in the chain and, thus, prevents double-spending of Bitcoins (or other currencies) and protects against fake transactions. Blockchains are managed by an open peer-to-peer community.

The way how blockchains are enlarged and additional blocks are added is known as *consensus mechanism* [34]. The most common consensus mechanisms are the *proof-of-work* and *proof-of-stake* protocols. Bitcoin uses proof-of-work where new blocks are added by a process called *mining*. Here, the incoming transactions are embedded into a block by finding a hash consisting of a given number of leading zeros. This is done by an iterative increasing of a variable, called the *nonce*, in the block header. Moreover, until the maximum number of Bitcoins is not reached mining creates a certain number of new Bitcoins which are given to the miner as a reward. Mining is always performed on the longest blockchain and parallel chains that arise by the contest between the miners are abandoned.

In 2022 Ethereum switched its consensus mechanism to proof-of-stake due to the high energy consumption and other disadvantages of the proof-of-work protocol. In proof-of-stake a new block is added by a limited group of validators that have to deposit a certain number of Ether, the currency of Ethereum, on a smart contract as guarantee that they behave honestly during the creation and validation process. For each block a validator is randomly selected to propose a block that means the validator is responsible for the creation and distribution of the block in the network. In addition to this, some other validators are randomly chosen to prove the validity of the proposed block. A malicious behaviour of a validator means the loss of the stake.

Evidently, smart contracts are important for the new consensus mechanism of Ethereum, since they serve as escrow accounts. The next subsection describes what smart contracts are and how they work.

2.2.2 Smart Contracts

As Bitcoin only allows simple transactions between the participants, other cryptocurrencies such as Ethereum [23, 161] established to improve this aspect. Ethereum introduces new concepts like accounts with a balance identified by an account address of 20 bytes and *smart contracts*. Smart contracts are small programs written for the *Ethereum Virtual Machine* (EVM) that make it possible to create more complex transactions such as money is only transferred to another person if a certain condition occurs, otherwise it is refunded. Additional to *external accounts* owned by real persons, each smart contract also owns an account with a balance, the so-called *internal account*. Messages or transactions that are sent from one account to another provide the ability to call contract functions, create new smart contracts, and to transfer money.

The source code of smart contracts is in most cases written in a programming language called Solidity [36]. In addition, Solidity also allows inline assembly instructions written in the language Yul. The creation process of a new smart contract starts by sending a message with compiled bytecode to an non-existent address. Then the contract address for the new smart contract is computed, the sent code is executed, and the result of the code is saved as the contract code in the internal account. Furthermore, even smart contracts are able to create new smart contracts by special commands. As the execution of programs consumes the validator's time and energy and not all programs make the same effort, the concept of *gas* has been introduced. Each message and the execution of

each command in the bytecode costs a certain amount of gas, a value that describes the computing effort independent from external influences such as the current workload. The gas is then mapped by a gas price to Ether and, thus, to real money which is transferred to the validator as fee additional to the block generation reward. So, the sender of a message has to define a gas limit and a gas price and the validator can decide if the message is worth to be included in a block or not. If a computation runs out of gas the changes are reversed, but the fee has to be paid nonetheless.

Source codes. Many smart contracts are open source. With the address of a particular smart contract, its source code can sometimes be found and downloaded from Etherscan [138]. Etherscan is a blockchain explorer that also allows browsing through the Ethereum blockchain and their testnets. Moreover, for some smart contracts it is possible to obtain their source codes from IPFS [110] if the creators published them there before. Source code is uploaded to IPFS with the Remix IDE [113] directly after compiling. To download the code the binary of the contract is needed, since it contains an IPFS hash in the metadata part (see *Section contract metadata* of [36]). This hash has to be converted to a content identifier [111] to obtain the IPFS address of a header file that contains the IPFS addresses of the source code files. As part of the blockchain binaries and contract addresses are publicly available and are obtained either from Etherscan or from an archive node [35].

2.2.3 Reentrancy and other Vulnerability Types

As explained the execution of contract bytecode costs gas and, thus, real money. So, it is important that very long or *infinite loops*, a common bug in smart contracts programming [178], are avoided, especially since the bytecode is statically included in the blockchain and cannot be changed except for stored variables or by a selfdestruction. Selfdestruction means here the removal of the entire bytecode of the contract from the blockchain.

However, loss of money can also happen due to vulnerabilities that are not related directly with gas. As the bytecode is connected to an account with its own Ether balance, and it is possible to transfer Ether from one account to another, errors in programming such as the *reentrancy* vulnerability offer a chance for attackers to steal money [37, 178]. In a reentrancy attack, an attacker deposits some Ether on the smart contract which stores the credit of users in a variable similar to a bank account. In a second step, the attacker withdraws the Ether, but due to an error in the execution order the smart contract sends the Ether first before reducing the internal balance variable. Money transactions with smart contracts have the special charactistic that after performing an Ether transfer the fallback function of the recipient account is activated. Smart contracts owned by an attacker can use this feature to repeat the withdrawing procedure before the transaction process of the sender is completed. In fact, since the balance variable in the sender account has not been reduced so far due to the error, the attacker obtains more money than he actually has deposited when starting the withdrawing procedure inside the fallback function.

In Solidity, the programmer has the choice between three ways of direct money transfer from one account to another, when designing a withdraw or a similar function. According to [37] Ether transfer is possible by the following programming constructs:

- *<CA>.call.value(x)*, or *<CA>.call{value: x}* for newer versions of Solidity
- *<CA>.transfer(x)*
- *<CA>.send(x)*

Here, <CA> indicates the contract or account address of the recipient and x describes the amount of Ether to be send. The main difference between the three functions is that by using *transfer* or *send* the gas that is forwarded to the fallback function for further processing is limited by 2300. By the use of *call* the programmer can decide individually how many gas is forwarded (see *Section "Address"* of [36]). Moreover, if an error arises in the fallback function *transfer* throws an exception in contrast to *send* or *call*. Clearly, the gas limit of *send* and *transfer* reduces the possibilities of reentering contract functions strongly and, therefore, the risk for reentrancy attacks at *call* is much higher. However, if the EVM is updated someday and the gas consumption of the commands or the gas limits of *transfer* or *send* are changed by this update, on the one hand the smart contract codes will still stay the same as before but on the other hand the risk of reentrancy by using these functions increases. Therefore, smart contract source codes have to be written in such a way that no damage can be caused.

The website SWC Registry [126] lists the known vulnerability types of smart contracts, offers examples for the vulnerabilities, and shows up ways to prevent or mitigate them. For avoiding reentrancy weaknesses two possibilities are presented: using a *reentrancy lock* [137] or performing all internal state changes before a *call* is executed. A reentrancy lock or *mutex* consists of a boolean contract variable that is switched to true when a certain contract function is entered and to false when it is left. Moreover, if this variable is already set to true when trying to enter a function the access is blocked or an exception is thrown. Thus, it is not possible to reenter a function equipped with this lock until all previous runs are completed. The second option is self-explanatory. If all internal states are changed before a call is performed at the end of the function a reentering into the function behaves as a function call after the termination of the function's control flow. Throughout this thesis source code parts are considered as vulnerable with respect to reentrancy if they do not use one of these two options to protect any direct money transfer against reentrancy. If in addition to this an attacker is able to use such a vulnerable source code part to steal money or cause other damages this code is considered to be exploitable with respect to reentrancy.

```
1  pragma solidity 0.4.24;
2
3  contract SimpleDAO {
4    mapping (address => uint) public credit;
5
6    function donate(address to) payable public{
7      credit[to] += msg.value;
8    }
9
10   function withdraw(uint amount) public{
11     if (credit[msg.sender]>= amount) {
12       require(msg.sender.call.value(amount)());
13       credit[msg.sender]-=amount;
14     }
15   }
16
17   function queryCredit(address to) view public returns(uint){
18     return credit[to];
19   }
20 }
```

Listing 2.1: Contract with reentrancy vulnerability in line 12 and 13. Source [126]

Listing 2.1 presents the source code of a smart contract that contains a reentrancy vulnerability in the withdraw function. The vulnerability is located in the lines 12 and 13, since in line 12 Ether is transferred to the address msg.sender and in line 13 a state change of

the variable credit takes place without any reentrancy lock. In fact, the variable credit describes how many Ether users have deposited at this contract and how much a user can withdraw. Due to the wrong execution order an attacker could reenter the withdraw function from his fallback function and withdraw more Ether than he owned. The update of credit takes place after the code of the fallback function is executed and, thus, the code at line 12 and line 13 is also exploitable.

2.3 Abstract Syntax Tree of Solc

Before Solidity source codes of smart contracts can be analyzed with GNN methods, they have to be transformed into a graph. In literature the entry point of many types of graph representations is the *Abstract Syntax Tree* (AST) [9, 8, 21, 167]. To create an AST, a compiler or similar program parses the source code, extracts important nodes and node attributes, and represents these nodes and their attributes in a tree form. In the case of the programming language Solidity that is used to program smart contracts for Ethereum the Solidity compiler Solc [39] is able to generate such ASTs from source code with the command

```
solc --ast-compact-json SOURCENAME.sol >> SOURCENAME.json
```

The name SOURCENAME.json indicates here the file that contains the AST in json format afterwards. Unfortunatelly, the AST format of Solc has not been documented yet [128]. Therefore, most descriptions of the AST in this thesis are obtained empirically by a comparison of source codes and resulting AST.

Listing 2.2 shows the source code of an example contract and Figure 2.7 depicts parts of the related AST created by Solc.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity ^0.8.5;
3
4  contract SomeContract {
5    uint x;
6
7    function setZero() public {
8        x=0;
9    }
10 }
```

Listing 2.2: Source code of an example contract

The AST consists of nodes that have properties such as *visibility* or *storage location*. Nodes are arranged in arrays, if they are on the same level. This is for example the case for the declaration of x and the function declaration. Moreover, nodes can be hierarchically nested inside an attribute of another node. An example for this is the *Assignment* node that has an attribute called *leftHandSide* which is also a node. Each node is equipped with an *id* number that is unique inside the AST. Nodes describe parts of the source code by their functionality and the corresponding node type (in Figure 2.7 marked with (a)) gives information about it. Other users of the Solc AST have identified 73 different node types [1]. These types are confirmed by additional analysis. However, more types can be added in future versions of Solc. Furthermore, it can be noticed that nodes that refer to variables defined at another node contain an attribute called *referencedDeclaration*, as it is the case for x in the example contract of Figure 2.7 (marked with (b)). In fact, the value of this attribute refers to the *id* of the node that defines the variable. So, in a later graph representation it is possible to connect associated variables and functions with an edge to express their dependency.

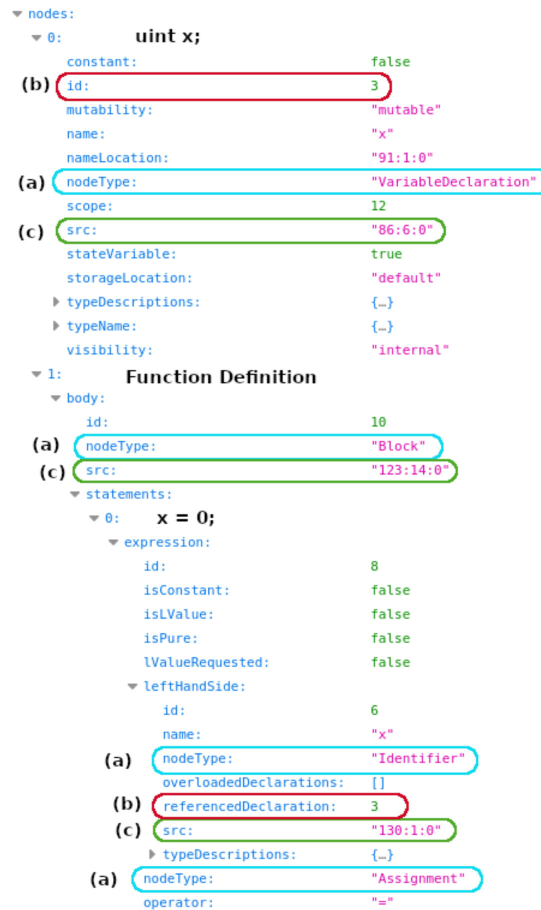


Figure 2.7: Parts of the AST of the source code of Listing 2.2

In addition to the node type, each node owns an attribute named *src*, in Figure 2.7 marked with (c). This field indicates to which source code parts the node belongs to. The first number is related to the file in the order of the import. If the file is read as bytes, the second number indicates the start position of the related code section. The third number is just the length of the code section (see this post [129] of a developer of Solc). It is crucial that the file is read as bytes and not as string, because not all control characters are represented in a string. Nevertheless, the *src* attribute is very useful. If a node is highlighted in some way it is also possible to highlight the related source code parts and vice versa. This fact will be used to highlight the source code parts that belong to nodes that are identified by a neural network as vulnerable or to provide the line numbers of vulnerable source code parts.

The nodes of the AST generated by Solc and their attributes are used in Chapter 4 to create a Solidity code graph that embedds the hierarchy of the AST, sequential processes inside the program, data dependencies, and also parts of the semantics of the source code.

3. Related Work

The thesis is about smart contract security and deals with explanation of GNN predictions. Moreover, a new approach for contract graphs is presented. This chapter describes the related works for these topics.

3.1 Smart Contract Security

Many analysis tools have been developed to detect vulnerabilities in smart contracts. Some of these approaches are described in the following.

A widely used method for identifying vulnerabilities in smart contracts is *symbolic execution*. Each input value is replaced by a *symbolic expression*, which is a variable defined on its entire input space. Then the execution paths of the smart contracts are converted into propositional logic. Subsequently, a Satisfiability Modulo Theory (SMT) solver is applied to proof if certain vulnerability conditions can be reached or not. In most cases the input of these tools is the bytecode of the smart contracts. Analysis programs that work by this procedure are Mythril [101, 17], MAIAN [106], Oyente [93], Manticore [100], and ETHBMC [40]. Many extensions and upgrades have been developed for Oyente: EthIR [3] creates a rule-based representation of the bytecode, SASC [175] provides rules for additional risks, Artemis [155] adds additional vulnerability types, and Honeybadger [150] detects the honeypot vulnerability. Moreover, since Manticore does not support all EVM instructions, SOLAR [82] closes this gap. The tool Osiris [149] detects integer bugs with symbolic execution and an additional *taint analysis*, a kind of tracking of the data across the control flow. SmarTest [127] applies symbolic execution at a language model of the smart contract and also creates transaction sequences. SAILFISH [20] uses a hybrid approach for source codes: A storage dependence graph performs analysis of side effects on the storage variables during the execution followed by the symbolic evaluation. In general, symbolic execution is very time consuming since all execution paths are checked. Therefore, the tools teEther [77] and sCompile [26] restrict the number of paths by a critical path selection before the analysis. An extension for Mythril for this feature can be found in [43].

Some static analyzers such as EtherTrust [57, 58] and EThor [118] are internally based on a reachability analysis of the bytecode with the help of Horn clauses. Furthermore, ZEUS [73] makes use of Horn clauses for symbolic model checking, but also relies on *abstract interpretation* to summarize the data changes in loops and functions. ZEUS is a source code based tool that uses LLVM as intermediate representation (IR) during the analysis.

NeuCheck [89], SmartCheck [145], and SESCon [7] transform the source code of the contract into a XML syntax tree and check this tree against vulnerability patterns. SESCon additionally performs a taint analysis with XPath queries.

Porosity [133] and Gigahorse [53] are mainly decompilers, but Porosity also offers a simple reentrancy detection method and Gigahorse a Datalog API with data-flow, dependency, and loop-semantic analysis. Securify [152, 151] extracts Datalog facts with symbolic analysis and checks these facts against vulnerability patterns with Soufflé [70]. Furthermore, Vandal [22] performs its vulnerability check by logic relations in dataflow also with Soufflé. MadMax [54, 55] is based on Vandal, but can also be applied in connection with Gigahorse, and is specialized on gas focused vulnerabilities. These vulnerabilities occur since the solidity compiler performs static analysis to estimate the gas consumption. Therefore, the real gas consumption sometimes differs. Other approaches to prevent these vulnerabilities are analysis and optimization of the source code (GasChecker [28], GASOL [2], GasReducer [30], Gasper [29], and Syrup [4]) and estimating upper boundaries for the gas consumption of the contract code (GASTAP [5], [94], and [130]).

Another method to find bugs is fuzzing. Here, sample inputs for the functions of a contract are generated automatically. Subsequently, the functions are tested with these inputs to uncover errors such as integer overflows that should not appear. Tools that employ fuzzing for smart contract vulnerability detection are ContractFuzzer [68], ILF [61], Echidna [56], ReGuard [85], GasFuzzer [13], sFuzz [104], Harvey [164], EthPloit [174], and EtherRacer [76]. Some fuzzers also generate the input with machine learning methods. Examples are ConFuzzius [148], a fuzzer with a genetic fuzzing algorithm, MTFuzz [123], and Neuzz [124], both neural network based fuzzers.

Moreover, there are other machine learning tools for vulnerability detection: ContractWard [156] is a framework with five different classification algorithms (XGBoost, AdaBoost, SVM, Random Forest, kNN). SCScan [60] is trained to detect vulnerable smart contracts with a SVM. Additionally, if a certain vulnerability exists, SCScan indicates possible locations with pattern matching algorithms in a second phase. Although SCScan provides the positions of vulnerabilities, this localization is performed in a second task which is independent of the first one. This means that the found vulnerability positions are not necessarily the reason for the positive classification result, in contrast to the positions provided by the two methods developed for this thesis. In a parallel work of this thesis, Mando [103] has been developed.

As the authors write, it provides fine-grained line level localization of vulnerabilities by a two phase approach. After the conversion of contract source codes into heterogeneous contract graphs, vulnerable contracts are identified in the first phase by graph classification. In the second phase, the vulnerable contracts found in the first phase are analyzed again by a node classification network. However, although Mando addresses the same topic as this thesis, there are some differences. Similar to SCScan, Mando is structured into two phases with independent detection and localization phases and, thus, the found vulnerability positions are again not the reason for the predicted contract label. In addition, Mando uses heterogeneous contract graphs which are only based on control flow graphs and call graphs. The methods presented in this thesis make use of homogeneous contract graphs that embedd not only the control flow or function calls, but also the hierarchy of the AST and data dependencies between the nodes. Another method, SC-VDM [176], converts the bytecode in a greyscale image and applies a convolutional neural network. Moreover, Liu et al. [178] propose to apply a graph convolutional neural network on a contract graph of the source code to detect bugs by graph classification. In the follow-up papers [87, 88] additional expert patterns are integrated to increase detection accuracy. Additional to the expert patterns, [88] also includes the ability to explain on which of the patterns the

classification result is mainly based. Giesen et al. [48] created a compiler that patches smart contracts source code with the help of a code property graph to mitigate integer and reentrancy bugs. Furthermore, Peculiar [162] extracts the dataflow of important variables of the source code in a crucial dataflow graph. Together with the source code this graph is fed into GraphCodeBERT [59] for reentrancy vulnerability classification.

3.2 Source Code Graphs

As it has been seen above, some tools convert the source code or bytecode into an intermediate representation before the bug detection starts. When using neural networks it is possible to treat source code input as text (see also [47]), but a lot of structural informations get lost in this case. An alternative approach with a graph neural network (GNN) and graph input is topic of this thesis. To use this method, the source code has to be transformed into a graph first.

The above-mentioned contract graphs of Liu et al. [178, 87, 88] mainly consist of three different node types. Major nodes represent important function invocations or critical variables. Other variables are seen as secondary nodes, and fallback nodes symbolizes a fallback function call. Edges are defined by the feasible program flow paths. Additionally, an elimination step is performed to reduce the graph size. Here, secondary and fallback nodes are removed and the edges are redirected. This final normalized graph is fed into a GNN that detects reentrancy, timestamp dependence, and infinite loop vulnerabilities.

The heterogeneous contract graphs that are applied in Mando [103] are based on call graphs and control flow graphs of the source code. A Multi-Metapaths Extractor generates metapaths from the node types and their associated edges in these graphs. Node embeddings are created with these metapaths and fused with the metapaths by a heterogeneous attention mechanism at node level. Afterwards, these node embeddings are fed into a MLP for a graph classification to check whether the contract is vulnerable. If that is the case the exact location of the vulnerability inside the contract is determined by node classification.

Allamanis et al. [9, 8] propose a connection of the source code with its abstract syntax tree (AST), a so-called *program graph*. The source code is transformed into an AST with a compiler or another helper tool. Since ASTs are a kind of nested dictionaries, the nodes have to be extracted and edges between parents and child nodes are drawn to obtain an AST graph. Then the leaves of this graph are replaced by the corresponding source code tokens and connected with edges that reflect the original execution order. To capture further control and dataflow structure, additional edges are inserted, for example, between all occurences of the same variable or in condition statements to indicate all valid paths. This graph type has been tested with a GNN for detecting variable misuse bugs and for some more program analysis tasks.

Moreover, Brauckmann et al. [21] enrich the AST produced by the C/C++ compiler Clang [139] with dataflow edges that connect identical instances of variables. Subsequently, nodes that are not structure critical are removed or merged with other nodes. Furthermore, a *Control- and Dataflow Graph* in the LLVM IR is proposed which orders the statements of the code by the control flow and also contains dataflow edges. In ASTs the order of the source code commands is usually dropped. Both models have been tested with GNNs that solve two tasks of OpenCL kernels, the CPU/GPU mapping problem and the determining of the thread coarsening factor. In both tasks, the AST-based model yields better results.

Similar to this, *Code Property Graphs* [167] are constructed from the AST by combining it with edges from Control Flow Graphs and Program Dependence Graphs. To be more specific, control flow edges are inserted for subsequent statements, loops, returns, and

similar constructs. The additional data and control dependency edges reflect influences of other variables and predicates on a certain variable such as the definition or an assignment of the variable. The graph type has been tested to uncover vulnerabilities in a Linux kernel. In further works, is has also been applied in connection with GNNs to detect vulnerabities in C functions [135, 177].

Other graph representations for source codes are the *Member Dependency Graph* and the *Component Dependency Graph*, both described in [105], with a focus on data items and methods or respectively on the components of a software system. In addition to this, the *Program Dependence Graph* [38] combines the control-flow and the value-flow of a program in one graph. It has been used in connections with GNNs to detect software vulnerabilities in C/C++ programs [31] and also for identifying malicious Android applications [166].

3.3 Methods for Explaining Machine Learning Models

Machine learning predictions are often hard to understand, since it is not clear on which facts the predictions are based. Explanation schemes for machine learning models can be of help in this case. The aim of these methods is to reveal the features of the input on which the prediction is based. The explanation is given in a human comprehensible form, for instance, as a binary or probability mask or as score values for all features. In this thesis, an explainer is needed to figure out the parts of the input that are responsible for the prediction of a contract label indicating a vulnerability. Therefore, this section gives an overview on existing approaches.

A widely applied explanation tool is LIME [114] that supports models with image or text input and treats them as black box. The method explains a single sample locally linear by generating additional samples that are slightly disturbed. Then a regression model is trained with these samples labeled with the original model's output. The scoring values of the correspondent features at the prediction task are given by the coefficients of the regression model.

The Shapley value [122, 98] is in game theory a way of measuring the average positive or negative contribution of a certain player to the average result of a game and are computed efficiently by a sampling algorithm [24]. Lipovetsky et al. [84] used this value to explain regression tasks by interpreting the feature values as players. This approach has been continued by the sampling algorithm of [153] that also handles classification tasks. Furthermore, Lundberg et al. [91] proposed SHAP, a method that combines LIME and Shapley values by a suitable weigthing of the generated samples. Consequently, Lime's scoring values that are approximated by the regression model are the Shapley values. Other variants of SHAP based on other methods converge faster, but need more structural information about the classification model such as Deep SHAP [91], a DeepLIFT [125] variant, and GradSHAP [90] that is based on the integrated gradients method [134].

Another way to explain the prediction results is to backpropagate this result to the input layer by a Layerwise Relevance Propagation (LRP) [79, 78, 6, 18]. For each input feature a relevance score is determined and successively computed for all other layers by a sort of backpropagation that needs structural information about the network. The relevance score at the input layer indicates the positive or negative share of the input features on the result. Further work discusses how to determine a prediction behavior of the network by a spectral relevance analysis (SpRAy) [80] using samples explained by LRP. Similar to this, DeepLift [125] explains the difference between the outputs of two samples with the differences of the inputs and avoids the problem of backpropagating contribution scores with zero gradients. The contribution scores of the input are again computed iteratively with certain rules. Integrated Gradients [134] improve some aspects of LRP and DeepLift

such as the implementation invariance. However, a baseline input with zero or nearly zero output at the neural network is needed. Then the path integral over the gradient of the network between the baseline input and the original input is computed to obtain the explanation.

Furthermore, Mohankumar et al. [97] propose an attention model for interpreting the result of neural networks. However, previous works [120, 65] indicate that attention models are not a reliable explainer in some cases, although there are also other opinions [159] about this.

The most explainers either need additional information about the network structure or even have to be inserted into the network. Besides LIME which is based on linear models there are also other explanation models which treat the network as black box, but use instead a nonlinear model, namely explainers based on rationalization [81]. The explainer model is trained by a reinforcement approach together with the classification model similar to encoder decoder adversarial learning. The final explanation is a binary mask for the input features. Moreover, rationalization methods are global methods that means they are only trained once and are able to explain any prediction of the model afterwards in contrast to local methods such as LIME. Additional approaches with rationalization can be found in [169, 15, 66, 108, 27, 12]. For the explanation of graph neural network prediction rationalization is not possible, since the input of size of this network type varies from sample to sample and for rationalization a fixed size is necessary.

As the above mentioned methods are built for feature values input, they can only make use of the feature vectors of the nodes of a sample when explaining the prediction of a graph neural network. Explainers that are optimized for graph neural network tasks additionally consider graph aspects such as the structure in form of edges. Moreover, there a explainers that a specialized to explain certain tasks such as link prediction, graph classification, or node classification.

With respect to the localization of vulnerabilities in source code graphs the explainers for graph classification tasks are most important. GNNExplainer [168] is a method that explains a graph neural network prediction by approximating a subgraph of the input graph that influences the predictions the most. It is a perturbation based approach where a feature selector and a mask for the edges are learned by using the output of the prediction model. GNNExplainer can be applied to link prediction, node classification, and graph classification tasks. In addition, GNNExplainer has been tested for vulnerability detection and, thus, is also used in this thesis for explaining the result of graph classification (see also Section 4.3.1). SubGraphX [172] also provides its explanations for predictions as subgraph created by a Monte Carlo tree search exploration algorithm and by a Shapley value importance measuring.

However, not all graph explainers offer a subgraph as output. The output of PGMExplainer [154] is a Bayesian network that is interpretable. Therefore, in a task where the most important nodes or features of a certain prediction have to be identified, this model has to be interpreted itself afterwards. The explanations of PGExplainer [92], GraphMask [96], and CausalScreening [158] are based on edges only. Since source code is mainly converted to node feature vectors, these three methods are of minor importance in the localization of vulnerabilities. Perturbation based explainers for node classification tasks are Gem [83], the LIME adaptation GraphLIME [63], MOO [86], and Zorro [45].

Gradient based methods for graph neural networks are mainly adaptions of other methods. Based on LRP there exists the Layerwise Relevance Visualization (LRV) [119], GNN-LRP [117], and [14] where also a guided backpropagation and a sensitivity analysis method are described. Moreover, there is a graph neural network version of Gradcam, described in [109].

There are also global methods working in a similar way as rationalization. XGNN [171] is a model level explainer consisting of a graph generator that iteratively adds edges to a subgraph of the input. The generator is trained with the feedback of the prediction model by reinforcement learning and supports graph classification tasks only. Like other explainers of this kind such as MEG [107], RC-Explainer[157], and RG-Explainer [121], XGNN is optimized for molecules and the output graph is always connected. However, for the localization of vulnerabilities it is necessary that also disconnected graphs appear as output, since one graph can contain more than one vulnerability at different locations that are not necessarily connected to each other.

4. Approach

This chapter explains in Section 4.1 the main topic the thesis deals with, the localization of vulnerabilities in Solidity source codes with machine learning methods. In the subsequent sections two developed methods are presented that solve this problem. This includes the construction of a new type of contract graphs.

4.1 Problem Statement and Goals

The goal of this master thesis is to develop new methods for detection and localization of software vulnerabilities in the source code of smart contracts using machine learning methods. In Ethereum, the source codes of smart contracts are mostly written in Solidity and its inline assembly language Yul. As explained in Section 2.2.3, these source codes can contain vulnerable parts that an attacker can exploit to steal money. Moreover, some other types of bugs such as infinite loops can result in a loss of money. However, if the source code of a contract is compiled and deposited on the blockchain it cannot be easily changed afterwards. This is also the case if the source code contains bugs that have only been found after publishing the contract. Thus, these bugs have to be revealed at the time of programming.

In Section 3.1 some methods have been discussed that are able to identify vulnerable smart contracts based on their source code or the compiled bytecode. However, many of these methods use predefined vulnerability patterns that make the application to newer vulnerability types or patterns more difficult or even impossible. Methods based on machine learning and, especially, on neural networks circumvent this restriction by learning the patterns from a large amount of vulnerable and not vulnerable data. After training, the methods are able to distinguish between vulnerable and not vulnerable contracts. Nevertheless, this approach has the disadvantage that in most cases it is not evident which part of the source code or bytecode is responsible for the prediction. Thus, in general it is not possible to determine the positions of the vulnerabilities in the code. A programmer, however, needs the exact positions to remove the vulnerability from the code and to make the contract secure. This work addresses this problem.

The two approaches presented in this chapter are based on graphs and Graph Neural Networks (GNNs) which provide some advantages regarding source codes over methods that interpret source code for example as text [47] or as greyscaled image [176]. Due to the functionality of GNNs input graphs can consist of different numbers of nodes or edges

and only the node feature vectors must have the same number of dimensions (see also Section *Advanced Mini-Batches* of [141]). Therefore, any size of contract source code that is converted to a graph can be analyzed, even if the GNN has been trained with only using small contract graphs. When using other machine learning approaches [47, 176, 60], the source code length has to be restricted to a fixed length. Thus, shorter contracts have to apply paddings and longer contracts have to be shortened. This is not necessary in approaches with graphs and GNNs. Moreover, it is possible to encode structural elements that are typical for programming lanuages such as loops or branchings directly into the structure of the graph. If the source code is treated as text input, for instance, and represented as matrix or tensor, the semantics of these elements are lost. Therefore, for both methods the Solidity source codes of smart contracts have to be converted to Solidity code graphs. The procedure for the graph generation is described in the following section. The last section of this chapter considers the two methods that are based on node classification and on graph classification with a subsequent explanation of the prediction.

4.2 Solidity Code Graph

This section describes a new graph type for Solidity source codes. In fact, this section presents how edges and node feature vectors of the graph are generated from the AST for this thesis. Information about the AST are provided in Section 2.3.

4.2.1 Edges of the Graph

As it has been discussed in Section 2.3, the AST of the source code of a smart contract contains hierarchically nested nodes that inform about the functionality of certain code parts. These nodes are extracted and serve as a sort of frame for forming the later graph. In this set of AST nodes no edges are defined. However, the hierarchy between the nodes is known from the AST. Nevertheless, edges are crucial for a later classification of the graph or of the nodes with GNNs. Therefore, edges have to be inserted always keeping in mind the structure and semantics of the underlying source code. A final graph shall represent the control flow, the hierarchy between the nodes, and also existing data dependencies between the nodes. These connections or relationships between the nodes have to be reflected by the inserted edges or by the graph structure. In some cases, the connections between the nodes are one-sided or need a strict order, for example, regarding the control flow of a function. Therefore, in principle, only directed edges are added. In a relationship between two nodes where an undirected edge is useful, two edges are inserted instead, one in each direction. The following sections describe the six different edge types that are added into the graph. Although it is in principle possible to classify graphs with more than one edge type, such a procedure is not performed by the methods presented in this thesis. Thus, all edges can be considered as equivalent. The edge types are only introduced for better traceability and comprehensibility of the graph structure. Moreover, some node types need a special embedding into the graph, for example those that are related to branchings or loops. The structure of these types is also presented in the next subsections.

Edges Representing Hierarchy of the AST

The first type of edges that are added to the graph are the edges between parent and child nodes defined by the hierarchy of the AST. As described in Section 2.3, nodes can have attributes that are nodes themselves. A node is identified by the attribute *nodeType* and also by the attribute *src* that connects the source code with its related AST node. If a node has an attribute that contains a node or even a list of nodes these nodes clearly have a parent-child relationship. A list of nodes can be found, for example, at the attribute

statements of node type *Block*, which refers to blocks in source code. This attribute lists the nodes that correspond to the statements in the source code block in the correct order.

In general, for each of the parent-child relationships two edges are added, one from the parent to the child and vice versa. In this thesis, these edges are called *AST edges*, since their origin lies in the hierarchy of the AST. Moreover, such AST edges are also part of many source code graphs that can be found in literature, for example, in the *Program Graph* of [9, 8] or the *Control- and Dataflow Graph* [21]. As AST child nodes are denoted in the attribute list of their parents, AST edges are also important for describing the properties of the nodes. The child nodes of certain statement nodes, for instance, describe the involved variables or expressions. In contract nodes, the child nodes are function nodes, nodes for contract variables, or similar. Therefore, by inserting AST edges also the main hierarchical structure of the source code is provided.

Control Flow and Ordering Edges

The final graph shall also reflect the control flow of the functions. Thus, another type of edges has to be introduced, the *control flow edges*. In the Solc AST, there are three node types that contain the statements of functions, loops, or similar, namely *Block*, *UncheckedBlock*, and *YulBlock*. The nodes corresponding to statements of the control flow are given as a list in the attribute *statements* of each of these three node types. The control flow is given by the order of these statement nodes in the list, since the nodes are ordered by the appearence of the related code parts in the source code. However, when extracting the nodes from the AST, this ordering by list is lost, since a graph contains no lists. Nevertheless, a sort of connection is needed to reflect the order of execution between the code statements or commands of source code blocks in the final graph. As indicated by the name, control flow edges are added to the graph between every two consecutive statement nodes of the list to preserve the execution order. Figure 4.1 shows how the graph structure of the three block node types looks like. Moreover, it should be noted that statement nodes mean all kind of nodes that can appear in the control flow including nodes refering to loops or branchings. Edges that refer to the control flow are part of the *Code Property Graph* [167], of the *Control- and Dataflow Graph* [21], and of many other types of source code graphs.

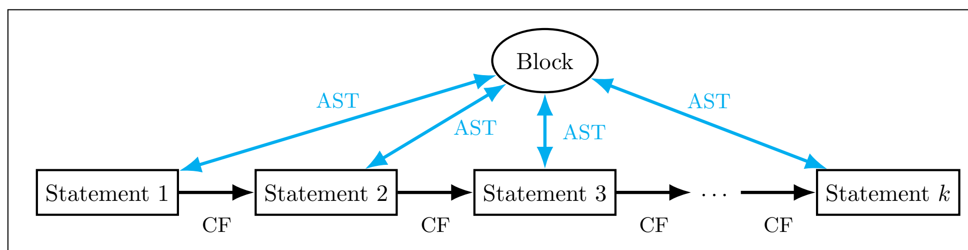


Figure 4.1: Control flow edges between k statements inside a block

The control flow is not the only case where a strict ordering of the nodes is needed. In some cases the nodes have two or more attributes that contain nodes of the same node type. An example for this is the node type *Mapping*. In Solidity, a mapping is a variable that connects keys of a certain type with values of another or the same type. In the AST, the type of the key node is connected by the attribute *keyType* with the mapping node, and the value type by the attribute *valueType*. Since the attribute connections are replaced by AST edges when generating the graph from the AST, it is no longer possible to distinguish between value type and key type, neither for a human nor for a computer. Therefore, a new edge type is introduced, the *ordering edge*, that clarifies the belonging to key or value by the direction of the edge. The edge is drawn from the *keyType* node to the *valueType*

node, for instance. There are eight other node types where this problem appears, namely at *IndexAccess*, *IndexRangeAccess*, *FunctionCall*, *FunctionTypeName*, *Assignment*, *BinaryOperation*, *FunctionDefinition*, and *YulFunctionDefinition*. For *IndexAccess* an ordering edge is added from the node at property *baseExpression* to the node at *indexExpression*. Similar to this, *IndexRangeAccess* receives an edge from the node at *baseExpression* to the one at *startExpression*, and an edge from the node at *startExpression* to the node at *endExpression*. Node type *FunctionCall* is equipped with an ordering edge from the node at property *arguments* to the node at *expression*. *FunctionTypeName* receives an ordering edge from the node at *parameterTypes* to the node at property *returnParameterTypes*. For the node types *Assignment* and *BinaryOperation* a distinction has to be made between left and right hand side. So, in both cases an ordering edge is inserted from the node at property *leftHandSide* to the node at property *rightHandSide*. At node types *FunctionDefinition* and *YulFunctionDefinition*, the input and the output arguments of the functions cannot be distinuished from each other without any additional edge. Therefore, an ordering edge is added from the node at property *parameters* to the node at *returnParameters* or at *returnVariables* for *YulFunctionDefinition*.

Edges Related to Variables and Functions

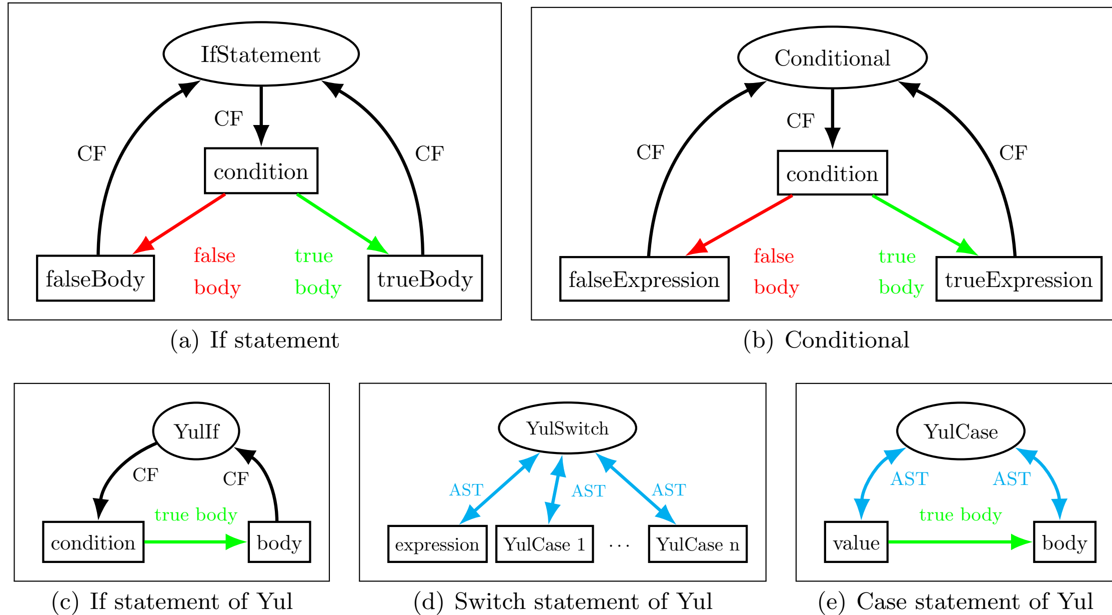
A basic concept of programming languanges is the usage of variables. In source code analysis, however, a problem due to different variable names may arise. Although the functionality is the same, the source code of two contracts differs if the variable names are changed. Nevertheless, the vulnerability analysis result of two contract codes that are identical except of the variable names have to be equal. In the AST, definitions of variables are replaced by a declaration node and further usages of this variable in the source code are mapped to nodes of type *Identifier*. Like all other nodes, a declaration node has the attribute *id* that uniquely identifies the node inside the AST. Moreover, identifier nodes have a special attribute called *referencedDeclaration*. The value of this attribute is identical to the id of the node where the related variable is declared. To reflect the data dependency between all usages of the same variable without a variable name, identifier nodes are linked with their declaration node by the new edge type *reference edge*. Reference edges are always drawn in both directions, meaning from the identifier node to the declaration node and vice versa. In many source code graphs known from the literature, edges are drawn between multiple usages of the same variable to represent the data dependency [162, 21, 9, 8]. However, in most cases the variable nodes are connected in the order of their appearence in the code to create a dataflow graph, which is different from the approach used in this thesis.

Similar to variables, functions can have different names, although their functionality is the same. However, nodes that refer to function calls also contain the *referencedDeclaration* attribute. Therefore, nodes with type function definition and nodes that reflect calls on this function are treated in the same manner as it is done with identifier and variable declaration nodes: Reference edges are drawn from the call node to its referenced definition and vice versa. In addition to the attribute *referencedDeclaration*, there are three other attribute types that refer to the ids of other nodes in the graph: *functionReturnParameters*, *baseFunctions*, and *superFunction*. If any of these attributes appear as property of a node, reference edges are added between the node and its referenced node to reflect the dependency.

Branching Edges

There are some types of nodes that have to be treated separately such as types that refer to branches in the control flow. If the control flow reaches some kind of branching, a condition has to be evaluated and depending on the result different blocks with code are

executed. In Solidity, such conditional branchings are represented by if statements or conditions of the form *<condition> ? <true expression> : <false expression>*. When generating the AST, the code of the branching is just mapped to a node that contains subnodes for the code components such as the condition or the blocks. Evidently, the real context between these subnodes and their meaning for the control flow is not reflected in the AST. However, in a graph the true order of execution of branches during the control flow can be mapped into the structure. Figure 4.2 shows how this is done for the different branch node types generated by Solc.



(a) If statement (b) Conditional

(c) If statement of Yul (d) Switch statement of Yul (e) Case statement of Yul

Figure 4.2: Graph structure of the node types referring to branchings

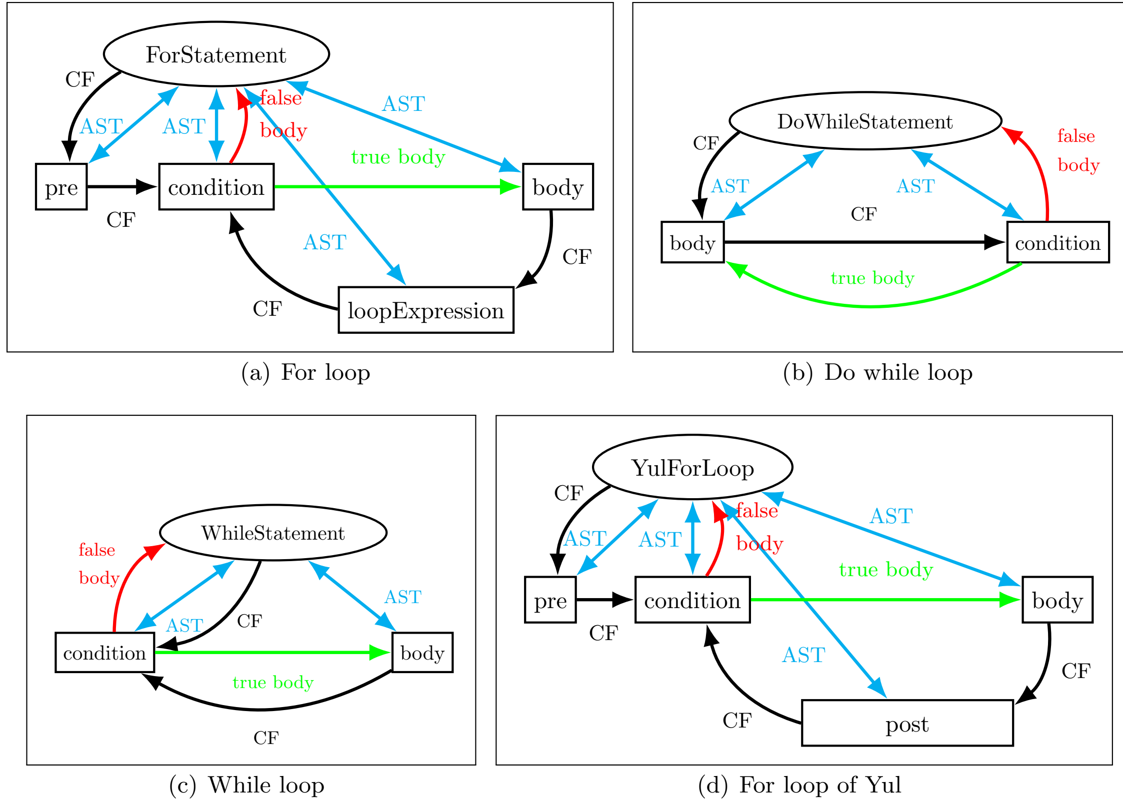
In the figures, nodes are represented by ellipses and the values of their attributes by rectangles. In this case the attribute values are nodes, too. Both, if statement 4.2(a) and conditional nodes 4.2(b) have three subnodes, one that refers to the condition and one each for the true and false branch. Thus, two new edge types are introduced: *true body edges* that connect the condition node with the statements or expressions of the true branch and *false body edges* for the connection with the false branch. Moreover, instead of using AST edges, control flow edges are added between if statement and conditional nodes and their children.

In the inline assembly Yul, there exist two programming constructs that refer to branches, the if statement and the switch case statement. In contrast to Solidity's if statements, the if statements of Yul only own a true branch. If the condition is evaluated as false, the statements in the true body are just ignored and the next statement after the branch is executed. This is reflected in the corresponding graph structure 4.2(c) by dropping the false body part. The switch case statement basically consists of a switch statement holding the expression that has to be evaluated and a number of case statements 4.2(d). The body of a case statement is executed if the result of the expression at switch matches the value given at case. Switch statements are mapped in the AST to nodes of type *YulSwitch* and case statements to nodes of type *YulCase*. In the graph 4.2(e) AST edges are drawn between the *YulCase* node and the two subnodes at the properties *body* and *value*. Moreover, a true body edge from value to body reflects the condition for the execution of the body. Nodes with type *YulCase* are connected with their related *YulSwitch* node by AST edges as well as the corresponding expression node as Figure 4.2(d) shows. The way how if statements are mapped into the graph structure in this thesis is similar to the handling of

if statements in Code Property Graphs [167]. The embedding of switch case statements into the graph structure has not been considered in literature.

Loop-related Edges

Moreover, the functionality of loops has to be integrated into the structure of the graph. In a AST of Solc, four different loop types can appear: The for, the do while, and the while loop, also known from other programming languages, and a for loop type for the Yul assembly. The resulting graph structure of these loops is shown in Figure 4.3.



(a) For loop

(b) Do while loop

(c) While loop

(d) For loop of Yul

Figure 4.3: Graph structure of the four loop types

The node type *ForStatement* 4.3(a) has four attributes where the values are nodes. These four nodes are connected with the *ForStatement* node by AST edges. When entering a for loop, at first, an initialization statement takes place, for example, the initialization of a certain loop parameter. This statement is saved in the attribute *pre* and connected with the *ForStatement* node by a control flow edge. Afterwards, the loop condition is evaluated. In the graph this is mapped by a control flow edge from the *pre* node to the *condition* node. Similar to the case of branching, the next step depends on the evaluation result of the condition. In the case of false, the loop is exited, which is reflected by a false body edge inserted in between the *condition* and *ForStatement* nodes and pointing towards the *ForStatement* node. Otherwise, the statements of the *body* node are executed and, therefore, a true body edge is drawn from *condition* to *body*. When this execution is finished, the loop update takes place and the loop condition is evaluated again. Thus, two more edges are added to the graph, one control flow edge from the *body* node to the *loopExpression* node and one from the *loopExpression* node to the *condition* node. Nodes of type *YulForLoop* 4.3(d) are handled identically to the *ForStatement* node of Solidity. However, the loop update is given here by the attribute *post* instead of the attribute *loopExpression*. The node types *DoWhileStatement* 4.3(b) and *WhileStatement* 4.3(c)

have only two attributes containing child nodes, namely *body* and *condition*. The nodes behind those attributes are connected with the loop nodes by AST edges. In a while loop, first the *condition* is evaluated and, depending on the result, either the loop is left or the statements of the *body* are executed.

In a do while loop, first the *body* is executed and afterwards the *condition* is evaluated. Therefore, in the graph structure of both node types there is an false body edge from *condition* either to the *DoWhileStatement* or to the *WhileStatement* node, a true body edge from *condition* to the *body* node, and a control flow edge from the *body* to the *condition* node. To mirror the entry point in while loops, a control flow edge is drawn from the *WhileStatement* node to the *condition* node and in do while loops a control flow edge from the *DoWhileStatement* node to the *body* node. In fact, if these last two control flow edges were removed, both node types would be mapped to the same graph structure. The Code Property Graph [167] also treats control statements such as for loops or while loops separately and constructs a preliminary control flow graph for them. However, the exact designs of these graph embeddings are not described.

Edges for Break, Continue, and Return Statements

Additional to the regular way of leaving a loop by the evaluation of the condition to false, there are six more node types that have to be taken into account: The node types *Break*, *Continue*, and *Return* and their Yul counterparts *YulBreak*, *YulContinue*, and *YulLeave*. Those node types can only appear as statements of the control flow and they are mapped into the graph structure similar to their functionality. If a node with type *Break* or *YulBreak* appears, a control flow edge is drawn from this node to the loop node, since a break causes that a loop is left. *Continue* or *YulContinue* means that the body of a loop is left, the loop update is done, and the execution afterwards proceeds in the normal order. Thus, control flow edges from nodes of these types are drawn to the loop update node. The function is left by nodes with node types *YulLeave* or *Return*. This is represented by control flow edges from nodes of this type to the function definition node. Program graphs [9, 8] handle the semantics of return nodes in the same manner. The Code Property Graph [167] considers breaks and continues as part of the control flow in the graph structure of loops. However, the embedding of break and continue in the graph structure of loops is not described in more detail.

4.2.2 Node Feature Vectors

After defining the edges of the graph, a closer look is taken on the nodes, more exactly on the node feature vectors. As it has been discussed in Section 2.3 each node of the AST owns a list of attributes, possibly containing subnodes. However, not all values of attributes are subnodes and, therefore, the attributes without subnodes give information about the properties of each node. The most important property is the attribute *nodeType* that describes the main function of the node or the source code part the node refers to. According to [1], there are 73 node types that have been verified empirically, since, as explained in Section 2.3, currently there exist no documentation for the AST. Regarding node feature vectors the first idea is to use one hot encoding of the 73 node types for each node. Then the feature vector of a node has 73 dimensions, one for each node type, and all dimensions are filled with zeros except of the one that indicates the type to which the node belongs. However, since some types are very similar in meaning and functionality and such one hot vectors consume a lot of memory, some of the dimensions are combined. This means that there is, for example, one dimension for the different types of loop nodes, one dimension for all nodes that are related to branchings and so on. In order to be able to differentiate between the individual types, inside these groups or dimensions different

values are set. A node of type *WhileStatement*, for example, contains in dimension zero of the node feature vector a two, in contrast to nodes of type *ForStatement* where in dimension zero of the node feature vector a three is set. Additional to the attribute *nodeType*, other properties of the node such as the *visibility* or the *storage location* are mapped to the feature vector. However, not all properties are used for the feature vectors. Variables are only mapped to their types, while the assigned values are omitted. Moreover, function or variable names and other strings or values are ignored except if there is a limited number of possibilities for them. A limited number of string values has, for example, the property *kind* of node type *Literal* that indicates to which type of the five possible literal types the node belongs to. In addition, nodes of the types *StructuredDocumentation*, *PragmaDirective*, *ImportDirective*, and *UsingForDirective* are not embedded into the graph, since they do not contain any important information for the vulnerability analysis.

Table 4.1 presents the meaning of all 29 dimensions and of their values. The first 20 dimensions refer to the different node types and the next eight ones to other node properties. The last dimension is some kind of special. The functions *send* and *transfer* are always mapped to a node of type *MemberAccess*, since they are functions belonging to the Solidity type *address payable*. However, the only indication on one of these functions inside the member access node is a string with the function name. Since in general infinitely many of these function names can appear, such strings are normally ignored in the feature vector. Nevertheless, some of these functions belong to the Solidity language and are important for vulnerabilities. Thus, they have to be integrated into a graph in some way if necessary for a certain vulnerability. The last dimension is meant for these types of nodes. Moreover, it can be noticed that for some node types, such as *Assignment*, more than one value inside of a dimension is used. In some cases, it is possible to divide the node type into subtypes by taking another property into account. For the type *Assignment* this more specific differentiation is performed by the attribute *operator*.

Table 4.1: Description of the node feature vectors

Dim	Values	Node Type	Further Description
0	1	DoWhileStatement	
	2	WhileStatement	
	3	ForStatement	
	4	YulForLoop	
1	1	IfStatement	
	2	Conditional	
	3	YulIf	
	4	YulSwitch	
	5	YulCase	
2	1	VariableDeclaration	
	2	Mapping	
	3	StructDefinition	
	4	ParameterList	
	5	YulIdentifier	
	6	YulVariableDeclaration	
3	$\in \mathbb{Z}$	Identifier / IdentifierPath	Value is 1 if id of referenced node is inside graph, otherwise equals referenced declaration id of external node
4	1	InheritanceSpecifier	
	2	OverrideSpecifier	

<table>
<tr><td>5</td><td>1
2
3</td><td>MemberAccess
IndexAccess
IndexRangeAccess</td><td>Member access with send or transfer function is not included here</td></tr>
<tr><td>6</td><td>1
2
3
4
5</td><td>SourceUnit
ContractDefinition
ModifierDefinition
FunctionDefinition
YulFunctionDefinition</td><td></td></tr>
<tr><td>7</td><td>1
2
3
4</td><td>Block
UncheckedBlock
InlineAssembly
YulBlock</td><td></td></tr>
<tr><td>8</td><td>1
2
3
4
5
6
7</td><td>VariableDeclarationStatement
ExpressionStatement
PlaceholderStatement
YulExpressionStatement
NewExpression
TupleExpression
ElementaryTypeNameExpression</td><td></td></tr>
<tr><td>9</td><td>1
2</td><td>EnumDefinition
EnumValue</td><td></td></tr>
<tr><td>10</td><td>1
2
3
4</td><td>ModifierInvocation
FunctionCall
FunctionCallOptions
YulFunctionCall</td><td></td></tr>
<tr><td>11</td><td>1-12</td><td>Assignment</td><td>Values describe different types of assignment operators: 1 → '& =', 2 → '| =', 3 → '^=', 4 → '<<=', 5 → '% =', 6 → '>>=', 7 → '+ =', 8 → '=', 9 → '* =', 10 → '/ =', 11 → '− =', 12 → unknown or not available operator</td></tr>
<tr><td>12</td><td>1-8</td><td>UnaryOperation</td><td>Values describe different types of unary operations: 1 → '−', 2 → '+', 3 → '−−', 4 → '++', 5 → '!', 6 → '∼', 7 → 'delete', 8 → unknown or not available operator</td></tr>
<tr><td>13</td><td>1-20</td><td>BinaryOperation</td><td>Values describe different types of binary operations: 1 → '|', 2 → '*', 3 → '>', 4→ '^', 5 → '%', 6 → '/', 7 → '&&', 8 → '! =', 9 → '||', 10 → '**', 11 → '<', 12 → '<=', 13 → '<<', 14 → '+', 15 → '>=', 16 → '>>', 17 → '&', 18 → '==',19 → '−', 20 → unknown or not available operator</td></tr>
</table>

14	1-5 6-9	Literal YulLiteral	Types of Literal: 1 → ’bool’, 2 → ’hexString’, 3 → ’number’, 4→ ’unicodeString’, 5 → ’string’ Types of YulLiteral: 6 → ’bool’, 7 → ’string’, 8 → ’number’, 9 → unknown or not available type
15	1 2 3 4 5 6	Break Continue Return YulBreak YulContinue YulLeave	
16	1 2 3 4 5	ErrorDefinition RevertStatement EventDefinition EmitStatement Throw	
17	1 2	TryCatchClause TryStatement	
18	1-6 7 8 9 10 11	ElementaryTypeName ArrayTypeName FunctionTypeName UserDefinedValueTypeDefinition UserDefinedTypeName YulTypedName	Types of ElementaryTypeName: 1 → ’int’, 2 → ’uint’, 3 → ’byte’, 4→ ’bool’, 5 → ’string’, 6 → ’address’
19	1	Unknown node type	Dimension for nodes whose node type is not in the list
20	1-5	All types	Different values of *visibility*: 1 → ’internal’, 2 → ’external’, 3 → ’private’, 4→ ’public’, 5 → unknown value
21	1-5	All types	Different values of *stateMutability*: 1 → ’payable’, 2 → ’view’, 3 → ’nonpayable’, 4→ ’pure’, 5 → unknown value attribute *payable* is true → 1 attribute *payable* is false → 3 attribute *isPure* is true → 4
22	1-4	All types	Different values of *mutability*: 1 → ’constant’, 2 → ’immutable’, 3 → ’mutable’, 4 → unknown value attribute *isConstant*, *isConstant*, or *constant* is true → 1
23	1-5	All types	Different values of *storageLocation*: 1 → ’memory’, 2 → ’storage’, 3 → ’default’, 4 → ’calldata’, 5 → unknown value attribute *stateVariable* is true → 2 attribute *isLValue* is true → 1

24	1-15	All types	Different values of *kind*: 1 → 'memory', 2 → 'modifierInvocation', 3 → 'baseConstructorSpecifier', 4 → 'typeConversion', 5 → structConstructorCall, 6 → 'functionCall', 7 → 'fallback', 8 → 'freeFunction', 9 → 'constructor', 10 → receive, 11 → 'function' Different values of *contractKind*: 12 → 'contract', 13 → 'library', 14 → 'interface', 15 → unknown value attribute *isConstructor* is true → 9
25	1-3	All types	attribute *implemented* is true → 1 attribute *fullyImplemented* is true → 2 attribute *abstract* is true → 3
26	1-3	All types	attribute *lValueRequested* is true → 1
27	1-6	All types	attribute *indexed* is true → 1 attribute *virtual* is true → 2 attribute *tryCall* is true → 3 attribute *isInlineArray* is true → 4 attribute *prefix* is true → 5 attribute *anonymous* is true → 6
28	-1, 1	MemberAccess	attribute *memberName* equals 'transfer' → 1 attribute *memberName* equals 'send' → -1

4.3 Vulnerability Localization with Machine Learning

This section presents two GNN-based methods to detect code vulnerabilities and to precisely identify their locations. The first method classifies source code graphs at the graph level and the second one at the node level. In both approaches the idea that is followed is to find the nodes that are related with the vulnerability. If these nodes are identified they can easily be mapped back to source code by the attribute *src* of each node. Therefore, these atttributes have to be saved with their corresponding nodes when the graph is created. The advantage of the first method over the second one is that only graph labels are needed for training, meaning the distinction between vulnerable and not vulnerable contracts. There already exist some datasets where smart contract source codes are labeled this way, for example, the dataset used in [162]. The second method, however, needs exact line numbers of the vulnerability locations in the source code. Currently, such a dataset with a representative number of labeled source codes does not exist. Therefore, it had to be assembled and labeled for this thesis as described in Section 6.2. Nevertheless, the first method has the disadvantage that an additional program, an explainer, is needed to determine the vulnerability locations in the code. The second method does not need additional tools, but provides the vulnerability positions and graph label in a single phase and, therefore, is more efficient.

4.3.1 Graph Classification with Explanation

The first method identifies vulnerable contracts by graph classification with a GNN. If a vulnerable smart contract is found, an explainer is used to determine the vulnerability positions. A similar approach also has been applied in [46] for vulnerability localization of source code of the programming languages C and C++.

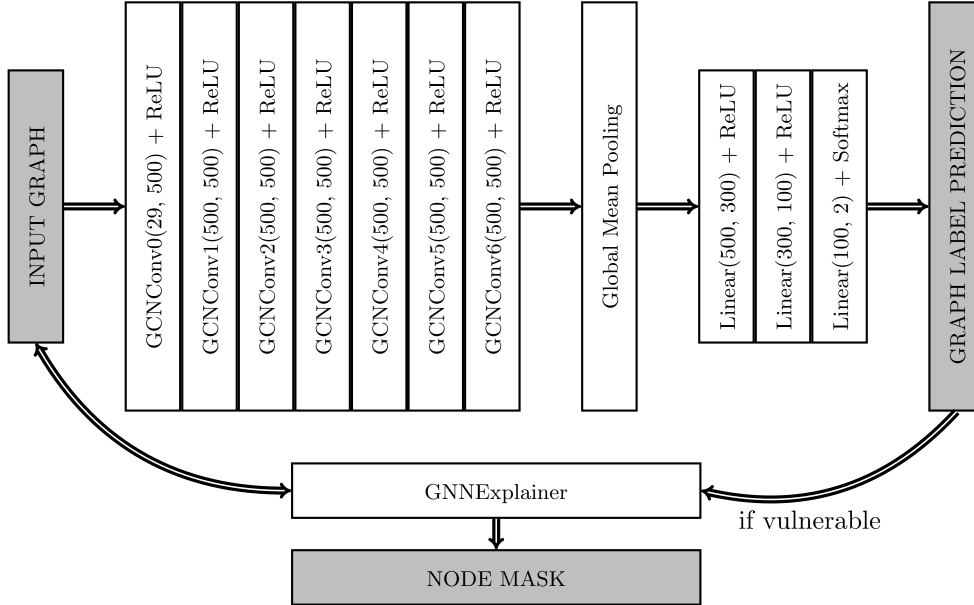


Figure 4.4: Structure of the method based on graph classification and explanation

In Figure 4.4 the structure of the method is shown. The method is divided into two parts, a GNN for graph classification and a part concerning the explanation. The GNN consists of seven GCN [75] layers with ReLU activation, a global mean pooling layer, and a MLP with two hidden layers and ReLU activation. Input of this GNN are Solidity code graphs that have to be generated before from smart contract source codes. It has to be noticed that a GCN layer is only able to handle one type of edges. However, as described in Section 4.2, Solidity code graphs can contain up to six different edge types. Therefore, to obtain a graph with only one edge type, the types of all edges are ignored and multiple edges between a node pair are reduced to one. During graph classification the feature vectors of the graph are expanded in the GCN layers from 29 dimensions to 500 dimensions. This increases the capacity and performance of the network, since more learnable parameters are provided. Moreover, seven GCN layers are used to ensure that the nodes are able to receive enough informations about the graph. The global mean pooling layer combines all feature vectors of a graph into one vector by taking the mean of all features. This vector represents the properties of the graph or the smart contract, respectively, and is fed into the MLP for being classified into vulnerable and not vulnerable contract. In the first step, this graph classification network is trained with smart contracts and their provided vulnerability labels. During training, a dropout layer with dropout probability of 25% is applied after each of the seven GCN layers and each of the two hidden layers of the MLP. After finishing the training, the network is able to detect vulnerable contracts.

When a vulnerable contract is detected, the goal is to find out the reason why the network classifies this contract or graph as vulnerable. More exacly, the nodes on which the prediction result is based have to be figured out. This is done in the second part of the method with the help of an explainer. An explainer determines the importance of features, nodes, or edges regarding a classification decision of a neural network and presents them

in a human readable form, for example, by score values. For this approach, an explainer is needed that works on the basis of nodes or even features, since these can be assigned to nodes. However, as there are explainers that consider aspects of the graph and a greater data basis normally produces better results, an explainer is preferred that explains graph classification tasks on the basis of nodes or of a subgraph. In literature (see Section 3.3) only two explainers have been found that fulfill this condition, namely GNNExplainer [168] and SubGraphX [172]. Furthermore, in [46] nine explainers have been tested on different GNN architectures and GNNExplainer is recommended in this paper for the detection of vulnerabilities. Nevertheless, there is one disadvantage of GNNExplainer. It uses a machine learning approach that becomes specialized on the input of the GNN during training to predict the nodes that are basis for the graph classification. Thus, to explain the prediction result for another graph GNNExplainer has to be trained again. However, existing global methods such as XGNN [171] only provide nodes of a connected subgraph as explanation. For a vulnerability localization this is insufficient, because a graph can contain more than one vulnerability and they do not necessarily lie close together. Therefore, when a vulnerability is predicted by the graph classification scheme, GNNExplainer is chosen to determine the nodes that are mainly responsible for the vulnerability label. As the vulnerabilities are responsible for the label, too, it is assumed that the determined nodes are the positions of the vulnerabilities inside the graph. For better readability and understandability, these nodes are mapped back to source code pieces with the node attribute *src*.

4.3.2 Node Classification

The first method uses graph classification to identify vulnerable contracts graphs and applies an explainer to figure out the nodes that are responsible for the vulnerability label on the basis of prediction. The second method swaps the roles and the graph or contract label predictions are determined by the presence or absence of vulnerable nodes. A prerequisite for this method is that the exact positions of the vulnerabilities for all contracts in the training set are known. Vulnerability position means in this case the lines that contain the vulnerability. With the help of the attribute *src* that all nodes of the Solc AST and, therefore, of the Solidity code graph own, these positions are transformed into node labels by the following procedure. If the related source code part of a node is fully contained in lines that are marked as vulnerable, then this node is labeled with one, otherwise it is labeled with zero. The idea is now to train a GNN on the task of node classification with the labeled Solidity code graphs of the training set. By this procedure the model learns how the vulnerabilities are embedded into the code graph. After training the GNN model is used to predict the node labels of before unseen code graphs. As all graphs are generated by the same procedure and, thus, are very similar, a kind of transfer learning is applied here. This means that knowledge about vulnerability patterns is generated in few graphs. Then this knowledge about the nodes related to vulnerabilities is applied or transferred to other graphs that have not the identical but a similar structure as the graphs the knowledge has been generated from.

The architecture of the model that is used for the node classification is shown in Figure 4.5. In fact, the model is similar to the one that is applied for the graph classification in Section 4.3.1. The only difference is that no global pooling layer exists that combines the node feature vectors to one vector for describing the graph properties. As the structure contains GCN layers that cannot handle more than one edge type, the input code graphs have to be adapted. Analogous to Section 4.3.1 the different edge types are removed by considering all types as equivalent and reducing multiple numbers of edges between two nodes to one. The absence of the pooling layer has the effect that a label can be predicted for all feature vectors of the input graph. Normally, node classification is performed on

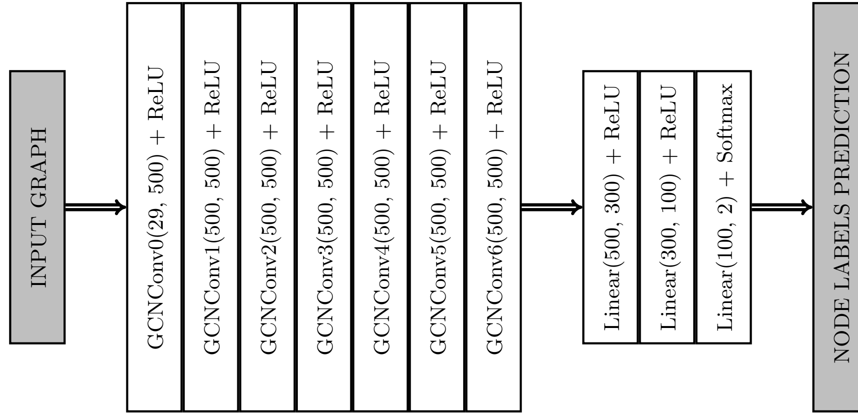


Figure 4.5: Architecture of the GNN for node classification and transfer learning

large scaled graphs where a part of the nodes are labeled and serve as training data. After training, the labels of the unlabeled nodes are predicted. However, in the case of the here described method the training and evaluation procedures are more similar to the ones done in graph classification tasks. The model learns to predict the node labels from the structures and features of all nodes of the graphs of the training set. This is achieved by continuously exchanging the graph that the model is learning from. Afterwards, the model predicts the labels of the nodes of the unknown graphs in the validation or test set. In fact, the output of the node classification is a binary mask that indicates the positions of the vulnerabilities in the graph. Like in the first method, described in Section 4.3.1, vulnerable nodes are mapped back to source code parts with the node attribute *src*.

5. Implementation

This chapter is devoted to the implementation of the two methods that are presented in Chapter 4. The chapter is structured as follows: Section 5.1 discusses the software packages and frameworks that are required for running and testing the program. The following Section 5.2 explains the structure and functionality of the program. Here, the program is also divided into its four main functional parts. The implementation of these parts is focus of the Sections 5.3 - 5.6.

5.1 Software Prerequisites

The implementation of the methods is written in Python 3.8.10, but should also work with higher Python versions. The code has been implemented and tested on a Linux Mint 20.3. This is important, since some used commands are related to the Linux shell, which may also work identically on other Linux or Unix operating systems, but possibly not correctly on Microsoft systems. Parts that may be affected are those that use the compiler Solc for AST creation and those that focus on the presentation of marked contracts on the console. Moreover, different versions of the Solidity compiler Solc are used and can be downloaded from the GitHub repository [39].

Furthermore, some additional Python modules are needed: The graph neural network and machine learning parts are written with *PyTorch* [147] version 1.9.0 and *PyTorch Geometric* (PyG) [141] 2.0.4. The GNNs are designed for training and evaluation with Cuda [42], but work also with CPU if no Cuda-supported graphic card is available. The code has been tested in an Anaconda [10] enivronment, too, and works without problems. Installation guides for installing PyTorch and PyG can be found on the referenced documentation web pages. In addition to this, the program uses the Python module *colorama* [69] for the print of marked smart contract source code on the console. Moreover, the scikit-learn [143] module is needed for the evaluation of the prediction results of node and graph classification. These two modules can be installed from the console via

```
pip3 install -U colorama scikit-learn
```

Additional necessary modules such as NumPy [140] are installed together with PyTorch and PyG and do not need to be considered individually.

5.2 Program Architecture

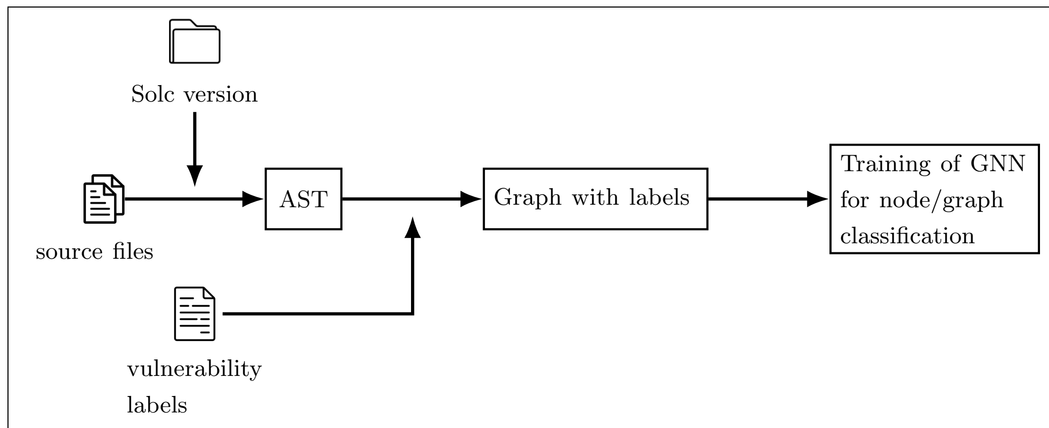
In Chapter 4 two methods for the localization of vulnerabilities in smart contracts have been proposed. These two methods have also been implemented and tested. This section gives a main overview on the functionality and the structure of the program. The detailed implementations of the individual components of the program are the subject of the following sections.

Figure 5.1 presents the procedures of the tasks the program is able to perform. These are the training and evaluation of GNNs that classify smart contracts source code with suitable vulnerability labels by node or graph classification 5.1(a) and the prediction and presentation of vulnerabilities in before unseen smart contract source code either by node classification 5.1(b) or by graph classification with explainer 5.1(c).

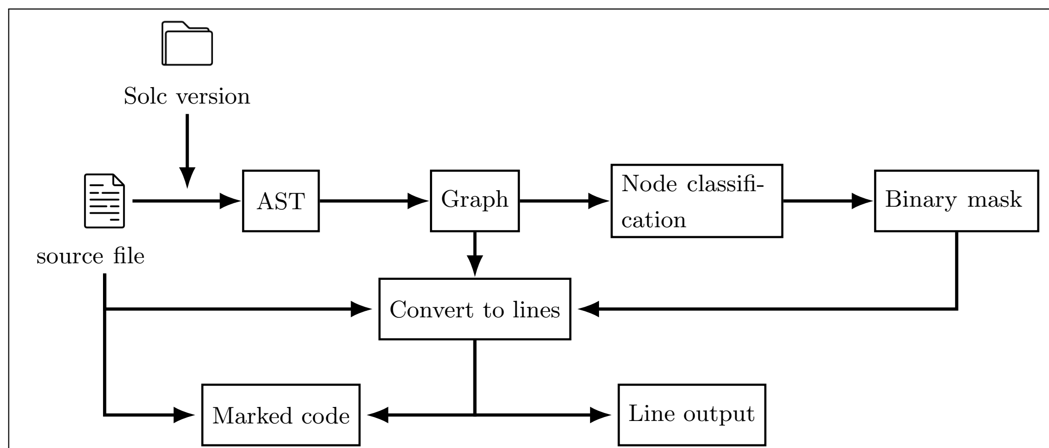
For all of these tasks it is assumed that a source code file always contains the complete source code of a smart contract. Thus, if the source code of a contract is split into more than one file these files have to be merged into one. Moreover, for the training and evaluation of the GNNs a complete dataset of smart contract source codes is needed. Additionally, these source codes have to be labeled with respect to a certain vulnerability type. For the here descibed program these labels are deposited in a comma separated CSV file in the following form. Each line in the file represents a contract. The first column of each line contains the name of the contract and each of the subsequent columns is filled with one line number that contains a part of the vulnerability. If there are no known vulnerabilities in a contract of the training or evaluation set the related line of the CSV file contains only the name. Contracts that are not mentioned in the CSV file, but are used for training, are always considered as not vulnerable.

The first step in the training process 5.1(a) is to construct ASTs from all the source code files. This is done with the help of different versions of Solc [39] that are located in a subfolder of the project directory. In a next step, these ASTs are used to construct graphs following the descriptions of Section 4.2. It should be noticed that the graph objects that are built here are *torch_geometric.data.Data* objects that means they are directly compatible with the PyG framework. Moreover, a graph of the training or evaluation set must also contain either the vulnerability labels of its nodes or the label of the graph depending on the task. As the labels are given as line numbers, the lines are converted into node labels during graph construction. A graph is considered as vulnerable if at least one of its nodes is vulnerable. In fact, both label types are saved in each of the graphs independently of the classification type, since they are useful for the evaluation of the methods. Moreover, dataset objects that are based on PyGs dataset class *torch_geometric.data.Dataset* are generated to group the graphs of the training, evaluation, or of the test set. Afterwards, the GNNs for node or graph classification are trained with the graphs of the training set by backpropagation and evaluated with the graphs from the validation and test set using certain evaluation metrics.

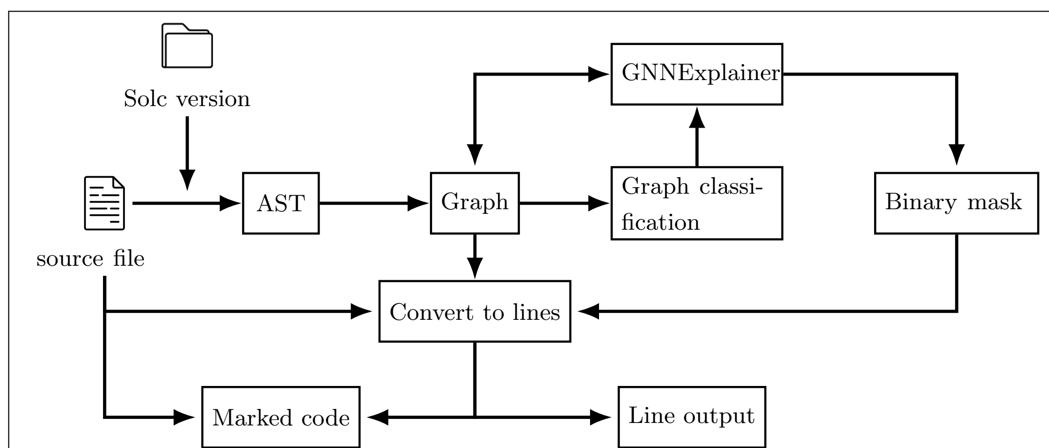
Having a trained GNN for node or graph classification, it can be applied to predict the location of vulnerabilities in 5.1(b) and 5.1(c). Similar to the steps done before in the training, a source code file is converted into an AST by Solc and then into a graph. This time, the graph does not necessarily contain any labels. If this graph is fed into the GNN for node classification 5.1(b), a binary mask is obtained where the nodes that belong to vulnerabilities are described by ones. If no ones appear in this mask, the contract is predicted as free of vulnerabilities. In a further step, the vulnerable nodes are converted back into line numbers. This can be seen as reversed step to the conversion of line numbers to node labels during training and evaluation process. Depending on the decision of the user, these line numbers are either the output or the basis of a marking of vulnerable code lines. In fact, there is a variable which switches between the two output modes.



(a) Steps to train GNNs for node or graph classification



(b) Prediction of vulnerability positions in source code by node classification



(c) Prediction of vulnerability positions in source code by graph classification and explanation

Figure 5.1: Overview on the program

The graph classification does not provide a binary mask directly, it only detects if a source contains a vulnerability or not. Therefore, if there is a vulnerability, GNNExplainer [168] is applied to obtain a score value for each node that indicates their responsibility of each node for the classification result. To obtain a binary mask from the score values, a thresholding is applied. The value of the threshold is a hyperparameter and is determined experimentally with the graphs of the validation set. Similar to the vulnerability localization with node classification, the binary mask is converted to line numbers, which are again depending on the user decision either the output or used to mark the positions of vulnerabilities in the source code.

As it can be concluded from the description of the functionalities, the implementation of the two methods is divided into four major parts: The two preprocessing steps, namely AST and graph generation, the machine learning part with the two GNNs and the explainer, and the conversion of line numbers to graph nodes and vice versa. In the remaining part of this chapter a closer look is taken on the implementation of these four parts.

5.3 Preprocessing Step 1 - Abstract Syntax Tree

The entry point of all tasks that are presented in the previous section is the creation of ASTs with the Solidity compiler Solc [39]. However, there are many versions of Solc, and the language Solidity has changed over time by removing or adding of some commands or by other changes in the semantics or syntax. Therefore, there exists no Solc version that is able to compile the source code of all contracts of the Ethereum blockchain. The version pragma that is part of smart contract source code files informs in most cases about the Solidity version the contract has been programmed with. It has the form

```
pragma solidity x.x.xx;
```

where x.x.xx stands for the Solidity version the contract has been programmed with or a higher version. Clearly, it is possible to create the AST of a few source code files manually by checking the pragma version, choosing the correct compiler, and typing the command for the AST creation (see Section 2.3) into the console. However, for a dataset consisting of a few thousand source code files this procedure can be a very time consuming activity. Therefore, the generation of ASTs has to be automated. Evidently, for this automation all versions of Solidity compilers are needed that can create ASTs. In fact, versions of Solc below version 0.4.12 are not able to provide a compact json AST, but use a different AST format. As only very few contracts are written with these versions, these contracts are not taken into account by the implementation. For all other versions the compiler Solc is downloaded and placed in the working directory subfolder './Compiler/'. The implementation of the automated AST creation process is located in the file 'solcAst.py'. There are five functions defined in this file. The first function 'getPragmaVersion' takes the path of the source code file as input, searches for version pragma in the file, parses the version, and returns this version as string. If there are more than one version pragmas in the code, the highest version is returned. Afterwards, the version string is used in the function 'createAst' together with the path of the source file and a path for the AST file for the automatic generation of the AST. In some cases, a contract has not been compiled with the given Solc version, but instead with a higher version. Then it is possible that the syntax also belongs to a higher version and the AST cannot be created and instead produces an AST file with zero size. In this case, the function 'giveAlternativeVersion' takes a Solc version as input and provides the last version of a release as alternative Solc version.

The json files that are created by Solc contain a few lines that indicate that an AST has been created by Solc and from which source file the AST is generated from. As these lines

violate the format of json, they can be removed with the function 'adapt2Json' that takes the path of the AST as input. The last function that belongs to the AST creation part is 'generateAST' that summarizes the creation procedure of ASTs. In addition to the path of the source code, it has a second parameter that changes the location of the AST. The two locations are needed to differentiate between the ASTs of source codes that are used for training or evaluation and the ASTs of source codes that are only used for tests, such as a time consumption test. The function returns true if the AST has been successfully created and a false otherwise. In the case of false it has also been tried to generate the AST with an alternative version of Solc given by the function 'giveAlternativeVersion'. The ASTs produced by this function are located either in './Data/AST/' or in './temp/'.

5.4 Preprocessing Step 2 - Graph Generation

The second preprocessing step for all program functions is the generation of the Solidity code graphs from the ASTs. The source code for this phase is located in the file 'graph.py'. In a first step during the graph creation of a contract, the corresponding AST json file is read as string. This string is converted to a python dictionary with the command 'loads' of the package json. A Python dictionary is a collection of key-value pairs and, thus, allows browsing through the complete nested AST tree with the keys. The command <dictionary>.keys() reveals the keys that are available in the dictionary and <dictionary>[key] returns the associated value which can also be a dictionary again. In fact, the keys mirror the attributes of the AST. Therefore, they offer a possibility to extract the nodes and create edges by a recursive programming approach.

The node extraction and edge creation step is mainly implemented in the function 'dicToNodes'. The function has six input parameters: The dictionary that holds a part of the AST, Python lists for the nodes, the edges, and the edge types, a dictionary for mapping AST node ids to new graph node ids, and a Python list for indicating breaks, continues, or returns in substructures. In Python, lists and dictionaries that are passed to a function and modified there are permanently modified. This offers the possibility of transferring data between the instances of a recursion. In fact, the function 'dicToNodes' is a recursive function, which means it calls itself inside the function body. One purpose of the function is to check by the node type property if the passed dictionary is an AST node. If this is the case, the node is added to the node list and its list index becomes its new node id. The mapping between the new id and the old AST id is needed in a later process and, therefore, denoted in the corresponding dictionary. Moreover, if the value of the dictionary at a certain key is also of type dicitionary, the function recursively calls itself to identify subnodes. The function returns the boolean value true and the index of the subnode in the node list if it finds a subnode, otherwise false and -1 as index. In addition to the identification of nodes, 'dicToNodes' handles the different node types and adds all corresponding edges except for reference edges (cf. Section 4.2.1). The representations of edges are identical to the in PyG [141] defined ones. This implies that the edge list has two sublists, one for the ids of the starting nodes and one for the ids of the ending nodes of the edges. The nodes that belong to a certain edge are identified by the same list index. At the end of the recursion, all important information about the nodes and edges of the future Solidity code graph is in the lists and dictionaries with which the function 'dicToNodes' has been called, since they are modified during the process.

The second function 'createReferenceEdges' receives as input parameters node and edge lists and a dictionary that connects AST ids with the indices of the node list. This function is designed to add reference edges in the sense of Section 4.2.1 to a graph.

Although the structure of the graph is finished by a successive execution of the 'dicToNodes' and 'createReferenceEdges' functions, the elements of the node list are still Python

dictionaries. The function 'createNodeFeatureVectors' converts this list to a list of node feature vectors as they are presented in Section 4.2.2. Moreover, this function extracts the 'src' attribute of each node in another list. Again, the same list index refers to the same node. Both lists, the feature vector list and the 'src' list, are the return values of the function. In fact, the resulting node feature vector list, the list of edges, and the list of edge types form the Solidity code graph.

The graph generation procedure is summarized by the function 'generateGraph'. The main argument of this function is a path to the source code file the graph is generated from. The function searches automatically in the AST directory './Data/AST' for the corresponding AST json file. Moreover, a second argument indicates whether the generated graph is saved as json file in the './Data/Processed/' folder. If the graph of a source code is saved and the function is called again at a later time with the same source code name, the graph is loaded from the json file instead of being generated again. This saves time, especially for large datasets of source codes or graphs. Furthermore, if a dictionary with line labels for contracts is passed as argument, the graph is enriched with node and graph labels. The derivation of these labels is discussed in more detail in Section 5.6. To ensure compatibility with PyG, the function returns the Solidity code graph in form of a *torch_geometric.data.Data* object.

Datasets. A dataset that is used for machine learning is organized as a training, a validation, and a test set. In the implementation, the source codes that belong to these three subsets, are registered by their name in the files './Data/Train', './Data/Validation', or './Data/Test'. The related source codes are located in the './Data/Raw/' folder. To keep the partitioning of the dataset into its three subsets the class SCDataset of the file dataset.py provides a way to group a number of graphs. It inherits from PyG dataset class *torch_geometric.data.Dataset* and mainly manages the graph generation procedure of the source codes registered in a dataset file such as './Data/Train'. This procedure includes the import of a CSV file with source code labels, as described in Section 5.2. However, the graphs in the directory './Data/Processed/' have to be deleted or moved to another folder if a labeling of another vulnerability type is used. This is necessary, since all generated graphs are saved and the dataset and its graphs offer only the possibility for the labeling by one vulnerability type.

5.5 Graph Neural Network and Explainer

The graphs that have been constructed from the source codes are used either for the training or for the evaluation of GNNs in a node or graph classification task. Moreover, the trained networks are able to predict vulnerable source codes and the positions of vulnerabilities in the code. To obtain vulnerability positions by graph classification, an explainer is needed in addition to the trained network. For this thesis, GNNExplainer [168] has been chosen as explainer. GNNExplainer is already part of the PyG [141] package and, therefore, the implementation of PyG is used to explain graph classification results. The results are scores for each node that indicate the responsibility of a node for the prediction in relation to the other nodes. A threshold that is figured out empirically transforms this score mask to a binary mask. This binary mask represents the predicted node labels.

The implementations of the two GNNs are located in the files 'classifyGraph.py' and 'classifyNodes.py'. The structure of both files is very similar. At first, a class is defined that inherits from *torch.nn.Module* and contains the GNN model as described in Section 4.3.1 or Section 4.3.2, respectively. The classes of both files have a constructor that initializes the learnable parameters of the GNN and MLP layers. In addition to this, a forward function is defined that describes the structure of the forward path with the output of the softmax function as return value. A function for computing the loss by cross entropy

completes the classes of both files. Moreover, each of the files contains two more functions: A function for training the network and a function for evaluating a trained model with labeled validation or test data. The training function loads the train and validation set as defined by the files './Data/Train' and './Data/Validation' and sets the optimizer as Adam [74]. Afterwards, the learnable parameters GNN are optimized by backpropagation with the labeled graphs of the training set as input. In fact, the input of the network are batches of graphs produced by a dataloader instance of the class *torch_geometric.loader.DataLoader* of PyG. A batch of n graphs is generated by merging the node feature vectors and edges of all graphs and adapting the node indices the edges refer to. In addition to this, a list with numbers from 0 to $n-1$ is created to indicate to which graph a node belongs to (see Section *Advanced Mini-Batching* of [141] for more details). This list is necessary for the creation of a graph feature vector by the global mean pooling layer in graph classification tasks. During the optimization process, statistics about the prediction are computed and printed on the console or into a file. The metrics that are used for the analyzing of the prediction are explained in more detail in Section 6.1. The second function that is part of the files and meant for evaluation computes the statistics without training loop.

The training, evaluation, and prediction processes are started by the main function of the file 'main.py'. The here defined prediction and evaluation processes by graph classification also include the explainer part described above.

5.6 Line and Node Label Conversion

The last part of the implementation concerns the conversion of line labels to node labels and vice versa. These conversions are necessary, since the nodes of the graph cannot be labeled by a human directly. Moreover, a node mask does not help a human to find a detected and localized vulnerability in the source code.

```
1  def setNodeLabels(sourceFilePath,node2sourceCode,GTlabels):
2      labels=[]
3      filename = sourceFilePath.split('/')[-1].split('.sol')[0]
4      with open(sourceFilePath, 'rb') as f:
5          contentb = f.read()
6      try:
7          pIndex = GTlabels[0].index(filename)
8      except:
9          return [0] * len(node2sourceCode)
10
11     for src in node2sourceCode:
12         src = src.split(':')
13         start = int(src[0])
14         end = start + int(src[1])
15
16         startLine = len(contentb[:start].decode("utf-8").splitlines())
17         endLine = len(contentb[:end].decode("utf-8").splitlines())
18         codeLines=[*range(startLine, endLine+1,1)]
19
20         if(all(x in GTlabels[1][pIndex] for x in codeLines)):
21             labels.append(1)
22         else:
23             labels.append(0)
24
25     return labels
```

Listing 5.1: Conversion from line labels to node labels

In Listing 5.1 the Python code of the function is shown that converts line labels to node labels. Input arguments are the path of the source code file, the list node2sourceCode with

the 'src' values of the corresponding graph nodes, and a three dimensional array named GTlabels. This array contains the ground truth labels from the CSV file described in Section 5.2. Thus, it is equipped with a list of source code names. In addition to this, for each name a list of line numbers is given that contains parts of vulnerabilities. More exactly, the array is structured as follows: The first dimension consists of two elements and distinguishes between source code name or line number access. The second dimension indicates which source code is selected. With the third dimension one can go through the list of line numbers of a certain contract. The function body starts by extracting the filename of the source code. Then in line 4 and 5 the source code file is read into a variable as bytes. Afterwards, in line 7 it is tried to find the source code name inside GTlabels. If the search is successful, the variable pIndex indicates the corresponding contract number in the array. Otherwise it is assumed that the contract contains no vulnerability and a zero mask is returned as label. The nodes of the related graph are implicitly given by the variable node2sourceCode. Therefore, for each of the nodes in 12 to 18 the lines of the source code are determined to which the node belongs. As the 'src' attribute always refers to byte positions and the labels are given as line numbers, a byte to string conversion has to take place here. In line 20 to 23, the node is labeled as vulnerable or 1 if all of the determined lines are also marked as vulnerable in GTlabels. Otherwise it is labeled as not vulnerable or 0. The resulting label mask is returned as result.

Similar to this conversion from lines to nodes the conversion from nodes to lines is performed. The corresponding function needs in addition to the source code path and the list with the 'src' values the predicted node labels. At first, the source code file is read as bytes as above. Then for each of the as vulnerable marked nodes the corresponding line numbers are determined with the same code as in lines 12 to 18 of Listing 5.1. The line numbers of all as vulnerable marked nodes are saved in an array and deduplicated before they are returned. Evidently, this direction is much more efficient, since only the line numbers of the marked nodes have to be figured out and not the line numbers of all nodes. The source code is marked by printing it line by line on the console and using a different font or background color for line numbers with predicted vulnerabilities.

6. Evaluation

This chapter presents the evaluation of the two discussed methods that detect and localize vulnerabilities. Section 6.1 introduces the metrics applied for this evaluation. Section 6.2 describes the hardware on which the GNNs are trained and the tests are performed. Moreover, the section provides detailed informations about the dataset that is used for the tests. The next two sections present the results of the tests that have been performed to measure the effectiveness of the methods. In addition to this, Section 6.5 considers the efficiency of both methods. The last two sections of this chapter contain a comparison of the results with methods from literature and some example contracts with localized vulnerabilities.

6.1 Metrics for Evaluation

The two methods presented in this thesis have to be tested to measure the quality of the outcome. Both methods are classification models based on GNNs and provide vulnerability labels for the graph and for each node. In node classification, the output of the GNN are predictions for the node labels. The graph is considered as vulnerable if at least one node is vulnerable. The method that is based on graph classification predicts the graph label. Then an explainer and a threshold generate the node labels. The vulnerability labels have two classes, vulnerable and not vulnerable. Thus, when classifying a sample contract and comparing the predicted node or graph labels with the true ones, the following four cases can appear according to [52]:

- **True positive (TP):** A vulnerable graph or node that is correctly classified as vulnerable by the model.
- **True negative (TN):** A not vulnerable graph or node that is correctly classified as not vulnerable by the model.
- **False positive (FP):** A not vulnerable graph or node that is incorrectly classified as vulnerable by the model.
- **False negative (FN):** A vulnerable graph or node that is incorrectly classified as not vulnerable by the model.

The 2×2 *confusion matrix* [52] counts how often these four cases appear when a dataset is classified. The first row of this matrix indicates the number of true positives in the first column and in the second one the number of false positives during classification. The second row consists of the number of false negatives in the first column and of the number of true negatives in the second column. The structure of the confusion matrix is illustrated in Figure 6.1.

TP **FP**

FN **TN**

Figure 6.1: Confusion matrix of a model with binary outcome.

During the training process of the two GNNs, the confusion matrix of the training and validation set are computed and noted. However, the values of this matrix depend strongly on the number of the samples that are evaluated. Therefore, the confusion matrix is not an adequate measure for the performance of the methods. To obtain measures that are independent of the number of samples, the following metrics are computed from the entries of the confusion matrix.

Accuracy [50]. The accuracy indicates the proportion of correctly identified graphs or nodes. It is computed by

$$\text{Accuracy} := \frac{TP + TN}{TP + FP + TN + FN} \tag{6.1}$$

Recall [51]. The recall indicates the proportion of the actual vulnerable graphs or nodes that are correctly identified. It is computed by

$$\text{Recall} := \frac{TP}{TP + FN} \tag{6.2}$$

Precision [51]. The precision indicates the proportion of graphs or nodes that are correctly identified as vulnerable. It is computed by

$$\text{Precision} := \frac{TP}{TP + FP} \tag{6.3}$$

F1 score [142]. The F1 score combines precision and recall in a harmonic mean. It is computed by

$$\text{F1} := \frac{2 * (precision * recall)}{(precision + recall)} \tag{6.4}$$

As described above, both methods provide predictions for graph and node labels. Therefore, the metrics can be computed on the graph level and on the node level. If the metrics are computed on the graph level, they describe the ability of a method to detect vulnerable contracts. Based on these metrics, the methods can be compared to other detection methods found in literature, since almost all methods in literature only provide vulnerability labels for the entire contract. Metrics computed on the node level measure the quality of a method to localize the vulnerability positions in the source code.

6.2 Dataset and Hardware

The two GNNs are trained on a server with NVIDIA A16 graphic card consisting of four GPUs of 16 GB RAM each. The server has 24 CPUs and 128 GB RAM. The methods are evaluated on a laptop with NVIDIA RTX 2070 graphic card with one GPU of 8 GB RAM that runs with Cuda version 11.6. Furthermore, the laptop has 64 GB RAM and a Intel i7-10875H CPU with 8 cores and 16 hardware supported parallel threads.

Dataset. The dataset is based on the dataset that has been used in Peculiar [162], a former version of the SmartBugs Wild Dataset [71]. The smart contracts have been redownloaded from Etherscan and relabeled, since a more precise line labeling is necessary to evaluate the two methods presented in this thesis. Moreover, the authors of [162] only provide labels for the subtype *call* of the reentrancy vulnerability. However, this thesis also considers the subtypes *send* and *transfer*, as described in Section 2.2.3.

Although the dataset has been deduplicated prior to this work, only those contracts that are exact copies of other contracts in the dataset have been removed. This means, even if all source code files contain different strings, there are still duplicates in this dataset in the following sense. In some contract codes additional white lines are added to the copy, or sometimes only the comments are changed. Moreover, many programmers use only different names for contracts, variables, or functions, or assign different values to the variables, but do not change more parts of the code. In these cases, the source code parts that are important for a static vulnerability assessment are in fact a copy of another contract in the dataset. Thus, if a machine learning method is trained with a contract that has such a copy in the dataset, it is not surprising, if the method can identify the label of the copy with the same quality. However, if the copy is in the test set, this reduces the quality of the evaluation, since the goal of the evaluation is to determine the performance of a method in finding vulnerabilities in before unseen contracts. In this case, the result of such an evaluation cannot be seen as a general result.

Therefore, this dataset has again been deduplicated in the following way. From all of the contracts the Solidity code graphs are constructed as described in Chapter 4. The compiler removes white spaces characters when building the AST and maps comments to its own node type which is ignored for Solidity code graphs. Furthermore, the graph does not own any immediate values or names of variables, functions, or similar due to construction. Afterwards, the source codes are divided into groups by the sha256 hashes of their related Solidity code graph. To finally deduplicate the dataset, one representative of each group is taken, resulting in only 22,237 contracts, less than half of the original dataset containing 46,057 contracts. In fact, the 23,820 contracts are duplicates of 3,684 contracts.

For this thesis, many of the contracts of the deduplicated set are labeled with respect to the reentrancy vulnerability subtypes *call*, *send*, and *transfer* (see Section 2.2.3). The contracts without any of the strings '.send(', '.transfer(', and '.call.value(' (or '.call{value:', respectively) are considered as not vulnerable. In the remining contracts, the strings concerning the three subtypes have been searched and it is checked manually, if a state change is performed after the money transfer has taken place and no reentrancy lock is used. If so, the lines that contain the vulnerability are noted into a CSV file as described in Section 5.2. For each of the three subtypes different CSV files are created. The author of this thesis has examined the locations with respect to type call and send and about 2,000 of the contract source codes concerning the transfer type. Many of the remainable contracts of type transfer have been labeled by a student assistant (HiWi). By including the contracts without any money transfer as not vulnerable samples, the final dataset contains 13,773 smart contract source codes that are line-labeled with respect to reentrancy vulnerabilities. The dataset is divided into training, validation, and test set. The composition of this set related to each subtype can be found in Table 6.1. In the tables the number of graphs is

equivalent to the number of contracts and the number of nodes refers to the total number of nodes in the corresponding Solidity code graphs.

Table 6.1: Composition of the dataset

(a) Composition regarding subtype *call*

	vulnerable graphs	not vulnerable graphs	vulnerable nodes	not vulnerable nodes
training set	573	11,040	28,900	11,468,676
validation set	121	959	5,430	1,447,293
test set	118	962	6,603	1,401,601

(b) Composition regarding subtype *send*

	vulnerable graphs	not vulnerable graphs	vulnerable nodes	not vulnerable nodes
training set	1,067	10,546	61,508	11,436,068
validation set	200	880	10,678	1,442,045
test set	226	854	13,207	1,394,997

(c) Composition regarding subtype *transfer*

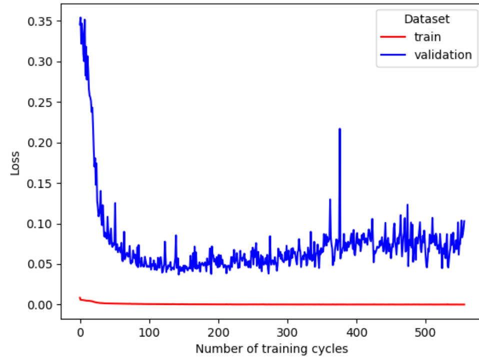
	vulnerable graphs	not vulnerable graphs	vulnerable nodes	not vulnerable nodes
training set	1,572	10,041	83,469	11,414,107
validation set	320	760	16,782	1,435,941
test set	346	734	18,213	1,389,991

6.3 Evaluation of the Graph Classification Approach

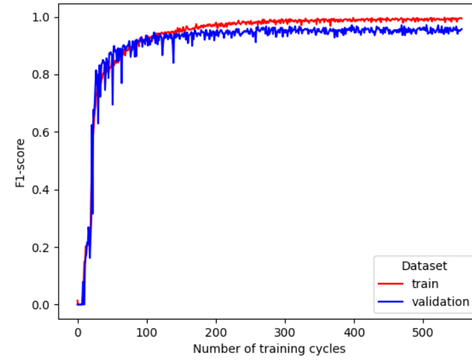
This section describes the training process and the results of the tests that have been performed to evaluate the method based on graph classification. Basis for this is the dataset presented in Section 6.2. The tests are performed on the graph and on the node level.

The first step for testing the method is to train the GNN. During training the results on the validation set are monitored and the metrics recall, precision, and f1-score are computed. Accuracy is not a good metric for monitoring the results at imbalanced datasets, which is the case for the validation set due to the large number of not vulnerable samples. In fact, the accuracy is for all three subtypes over 80%, if the model classifies all samples as not vulnerable. Therefore, the accuracy is only computed at the final test of the validation and test set. During a few of these training processes the hyperparameters are optimized by grid search [16] to obtain high values of the metrics. One result of this hyperparameter optimization is that the model does not need more than seven GCN layers, since the three metrics do not improve, if more layers are used. It is clear that such a maximum number exists, since after a certain number of layers each node has received information of all the other nodes in the graph. Thus, additional layers will only bring redundant information to the nodes. In addition, the learning rate is set to the highest possible value that does not cause the results to oscillate too much, in this case 0.0001. The size of the batches is set to 30.

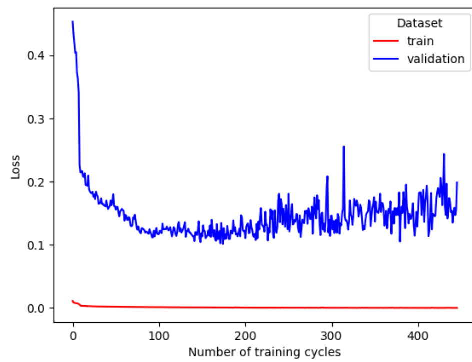
The loss and the computed metrics of the last training process are shown in Figures 6.2(a) - 6.2(f) and Figures 6.3(a) - 6.3(f). It can be seen that the f1-score for detecting vulnerable graphs is for all three subtypes higher than 85%, for subtype *call* and *send* even over 90%. Moreover, for all subtypes the recall is nearly on the same level as the precision.



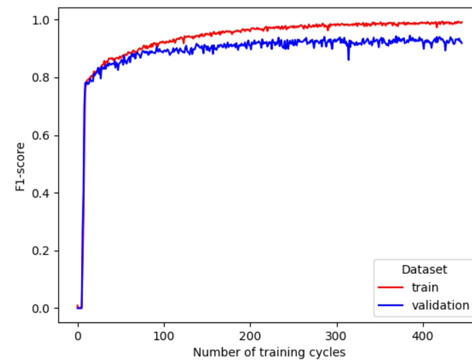
(a) Loss regarding subtype *call*



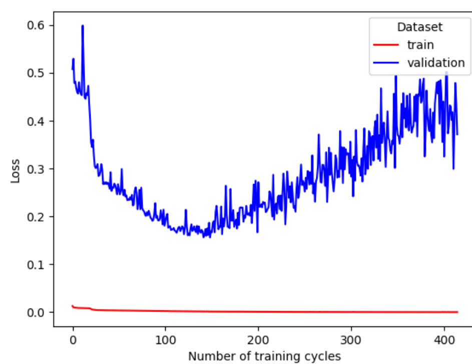
(b) F1-score regarding subtype *call*



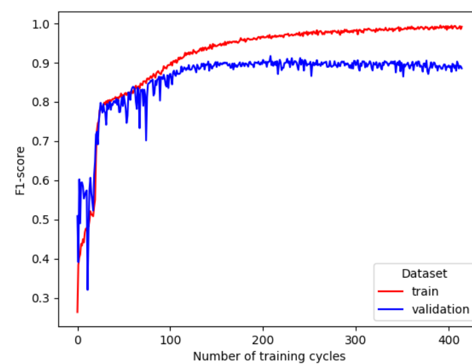
(c) Loss regarding subtype *send*



(d) F1-score regarding subtype *send*

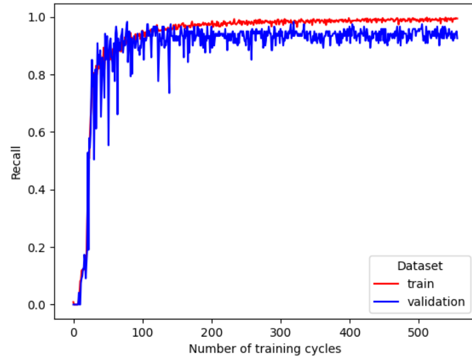


(e) Loss regarding subtype *transfer*

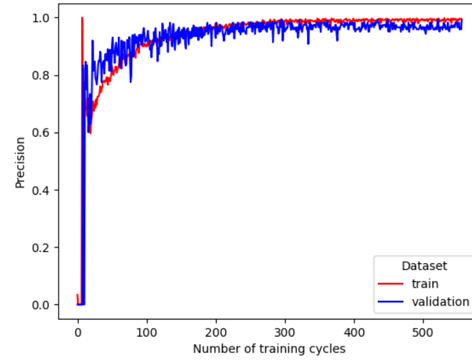


(f) F1-score regarding subtype *transfer*

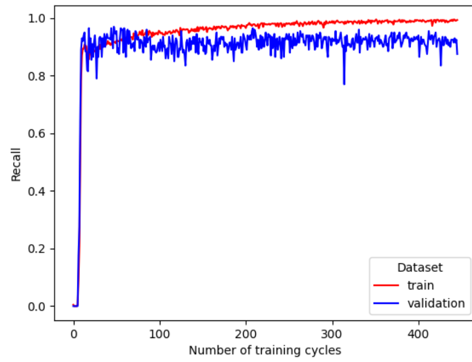
Figure 6.2: Graph level losses and f1-scores during training of the graph classification method regarding the three reentrancy subtypes.



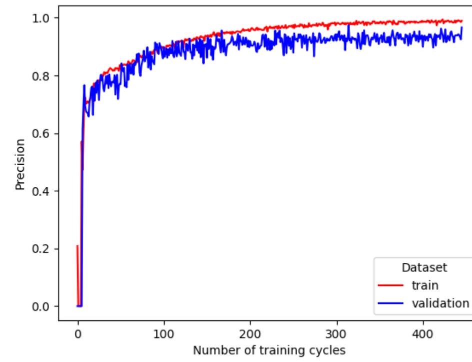
(a) Recall regarding subtype *call*



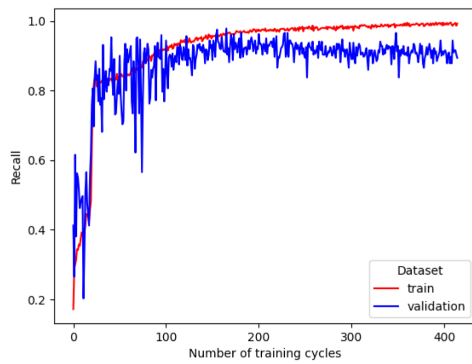
(b) Precision regarding subtype *call*



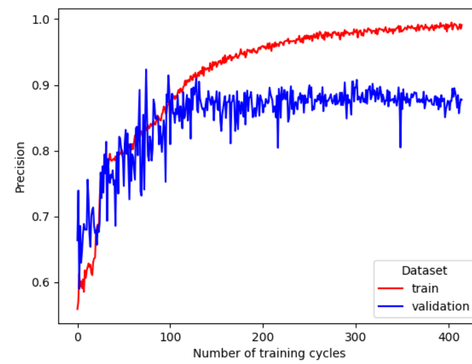
(c) Recall regarding subtype *send*



(d) Precision regarding subtype *send*



(e) Recall regarding subtype *transfer*



(f) Precision regarding subtype *transfer*

Figure 6.3: Graph level precision and recall during training of the graph classification method regarding the three reentrancy subtypes.

As the graph classification provides directly the prediction results on the graph level, the model is now tested on this level. For the evaluation of the test set on the graph level these values of the learnable weights are taken that yield the highest f1-score in the last training process. The f1-score is a suitable measure, since it is a harmonic mean of recall and precision. The results of these tests are given in Table 6.2. It can be seen that vulnerable contracts are reliably identified with a recall of over 94% for all subtypes. However, the values of precision are a little worse, especially for the subtype *transfer*. This means that many contracts are detected as vulnerable but are in fact not vulnerable.

Table 6.2: Graph level results of graph classification for the three reentrancy subtypes.

(a) Results on validation and test set regarding reentrancy subtype *call*.

	accuracy	precision	recall	f1-score
validation set	0.9935	0.9672	0.9752	0.9712
test set	0.9815	0.8952	0.9407	0.9174

(b) Results on validation and test set regarding reentrancy subtype *send*.

	accuracy	precision	recall	f1-score
validation set	0.9796	0.945	0.945	0.945
test set	0.9713	0.9149	0.9513	0.9328

(c) Results on validation and test set regarding reentrancy subtype *transfer*.

	accuracy	precision	recall	f1-score
validation set	0.9306	0.8284	0.9656	0.8918
test set	0.9444	0.8743	0.9653	0.9176

The next step is to evaluate the method on the node level. Graphs that are predicted as not vulnerable cannot have any vulnerable nodes. Thus, in this case all node labels are set to not vulnerable. For the node labels of graphs identified as vulnerable GNNExplainer [168] is used to reveal the nodes on which this result is based. GNNExplainer returns score values for each node. A threshold is needed to obtain a node mask or the node labels from the score values. As the scores are not limited, the threshold has to depend on the maximal value of the scores. Therefore, the threshold is set to be a multiple of this maximal value. The unknown multiplicator is figured out by creating score masks for all vulnerable graphs in the validation set. For multiplicators in the range of 0.5 to 0.99 the resulting mask is determined and with the ground truth labels of the validation set the f1-score is computed. It turns out that for most graphs the highest f1-score is reached if the multiplicator has a value between 0.70 and 0.90. Thus, the multiplicator is set to 0.80 and the threshold to $0.8 * max(node_scores)$. GNNExplainer is trained for 100 epochs with a learning rate of 0.007.

The test results of the node level for all three reentrancy subtypes can be seen in Table 6.3. Nearly all metrics have small values. The accuracy is high due to a large number of not vulnerable nodes that have been correctly identified. For all subtypes, only every fourth or fifth node that is part of a vulnerability has been found, which can be inferred from the value of the recall. Moreover, many not vulnerable nodes have been incorrectly classified as vulnerable, as shown by the extremely low precision values. Probably, by choosing another way of labeling, for example by only labeling the lines belonging to the money transfer, the values could increase. In addition, the low number of vulnerable contracts could cause that many incorrect patterns are found by the network. If so, the predictions are partly based on patterns that are more or less random. Thus, nodes are responsible for the prediction that do not belong to the vulnerability and this would explain the high

number of false positives. However, this has to be examined in connection with other explainer types and, therefore, in a future work.

Table 6.3: Node level results of graph classification with GNNExplainer for the three reentrancy subtypes.

(a) Results on validation and test set regarding reentrancy subtype *call*.

	accuracy	precision	recall	f1-score
validation set	0.9694	0.0266	0.2015	0.0470
test set	0.9661	0.0300	0.1992	0.0522

(b) Results on validation and test set regarding reentrancy subtype *send*.

	accuracy	precision	recall	f1-score
validation set	0.9587	0.0537	0.2781	0.0900
test set	0.9481	0.0506	0.2550	0.0844

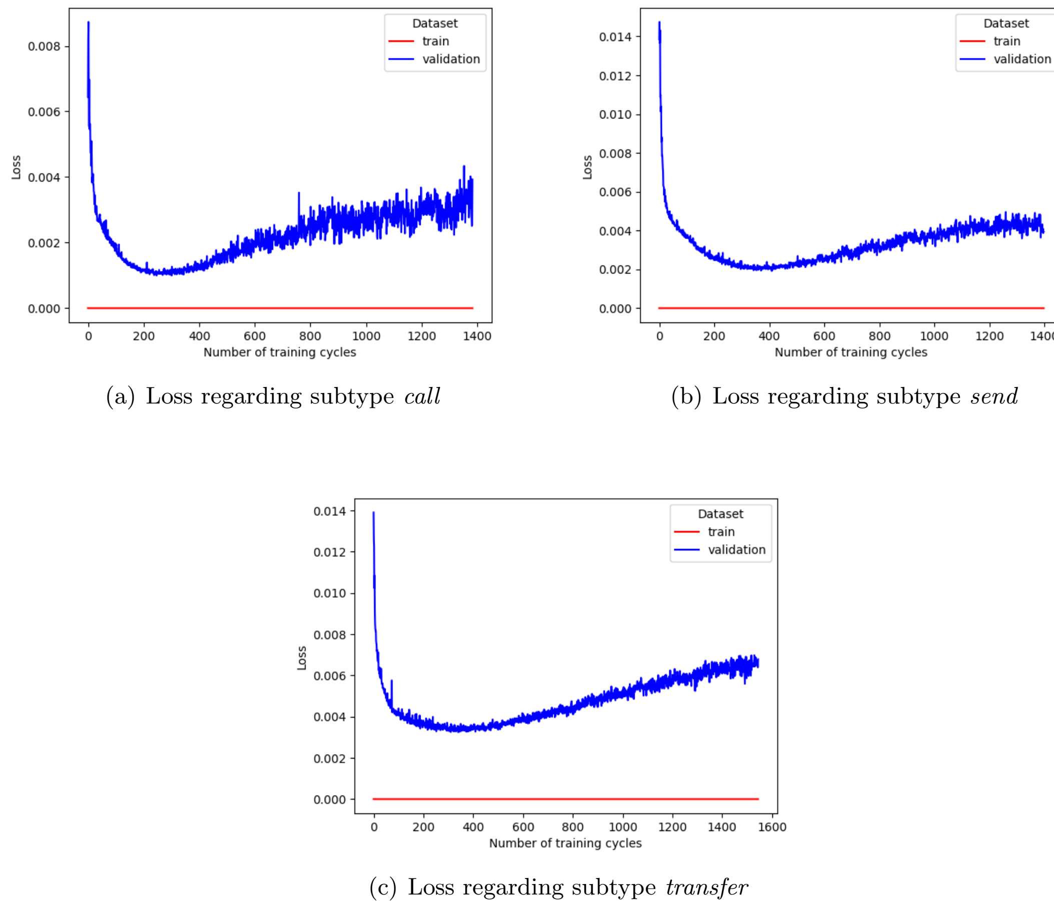
(c) Results on validation and test set regarding reentrancy subtype *transfer*.

	accuracy	precision	recall	f1-score
validation set	0.8789	0.0258	0.2579	0.0469
test set	0.8760	0.0316	0.2894	0.0569

6.4 Evaluation of the Node Classification Approach

Similar to the method with graph classification, the GNN embedded in the node classification method has to be trained and the hyperparameters have to be optimized. Therefore, as in the first approach, the performance of the method must be monitored by metrics during training. In contrast to the graph classification method, metrics are given at both levels, graph and node level, during training. The node level masks are the output of the node classification tasks and the metrics can be computed from this mask and the true labels. A graph is assumed to be vulnerable if at least one of its nodes it vulnerable. This is applied to the predicted mask and the true labels to obtain the metrics on the graph level. As before precision, recall, and f1-score are the metrics that are monitored.

As the network trains on the same data the same number of GNN layers is needed as in graph classification. Moreover, the number of training parameters in these layers seems to affect mainly to the duration of the training but not the performance. The learning rate is again set to 0.0001 and the size of the batches at 100. The losses of the final training process for the three reentrancy subtypes are shown in Figure 6.4. It can be seen that the losses regarding the validation set increase again at some point. A reason for this could be the proportion of not vulnerable nodes to vulnerable ones. There are much more not vulnerable nodes in the graphs and, in addition the GNN predicts at the beginning nearly all nodes as not vulnerable. However, when the network learns to classify the vulnerable nodes correctly, the probability of some similar not vulnerable nodes decreases and the average loss increases due to the proportion. The precision for all three types is shown in Figures 6.5(a) - 6.5(f). At node level for subtype *call* and *send* the saturation level is around 90% and for *transfer* it is a little lower at 80%. Thus, the number of false positives is much lower than in graph classification with explainer. Moreover, the prediction of the graph label by the above described procedure seems to work, too. The recall and the f1-score are shown in Figures 6.6(a) - 6.6(f) and Figures 6.7(a) - 6.7(f). The recall at node level looks similar to the precision and, therefore, the f1-score, too. At graph level it can be seen that almost all vulnerable graphs or contracts are identified. However, this has to be proved with the final test.



(a) Loss regarding subtype *call*

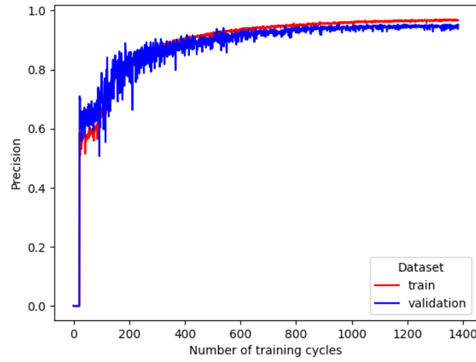
(b) Loss regarding subtype *send*

(c) Loss regarding subtype *transfer*

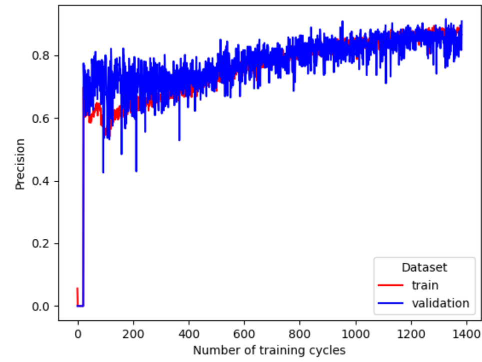
Figure 6.4: Losses per node during training of the node classification method regarding the three reentrancy subtypes.

For the final tests again those values for the learnable weights are taken that yield the highest f1-score evaluated on the validation set. The results of the final tests are presented in Table 6.4. Evidently, the recall at the graph level is nearly at 1 for all subtypes. This means that nearly all vulnerable graphs or contracts are identified. However, as the precision value of 74% to 87% show there are many false positives. At node level, which refers to the localization of the vulnerabilities in the graph, there is not much difference between precision and recall values. The values of the metrics regarding subtype *transfer* tend to be slightly lower than those of the other two subtypes. Probably, this is caused by a higher number of vulnerable nodes in the validation and test set. The accuracy at the node level for all types is almost 1 due to the high number of not vulnerable nodes. On the graph level, the method achieved accuracy values between 89.9% and 98.3%, which is quite good for a method that is not trained on this level.

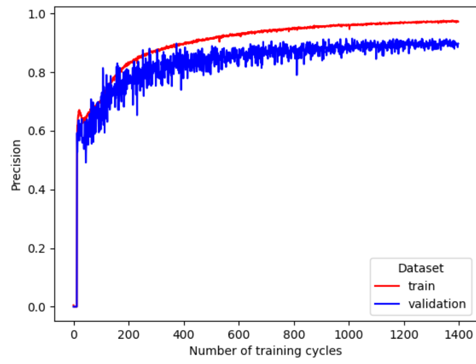
In summary, it is concluded that this node classification method is very accurate and, therefore, seems to be useful for the detection of vulnerable contracts and the localization of the vulnerability within the contract. However, in addition to the tests in this section it has to be tested how fast the method can predict the vulnerabilities. This test is performed in the next section.



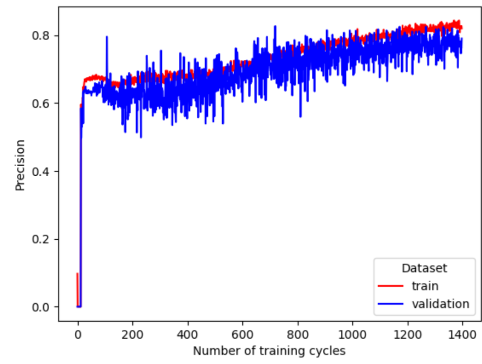
(a) Subtype *call* at node level



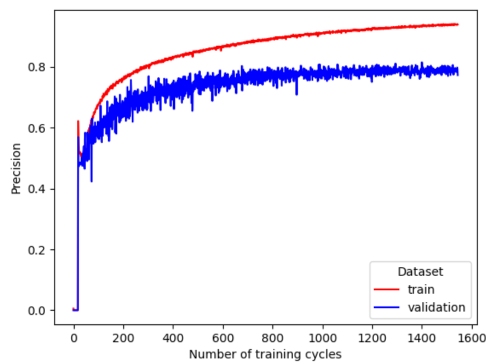
(b) Subtype *call* at graph level



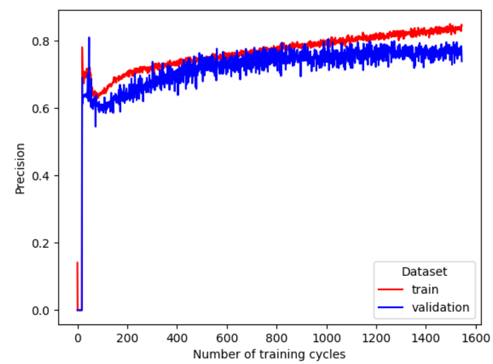
(c) Subtype *send* at node level



(d) Subtype *send* at graph level

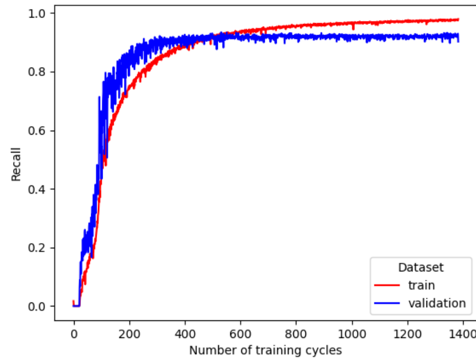


(e) Subtype *transfer* at node level

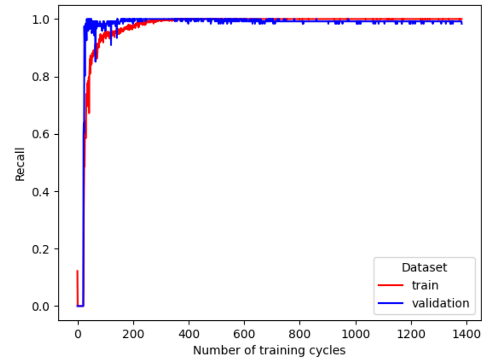


(f) Subtype *transfer* at graph level

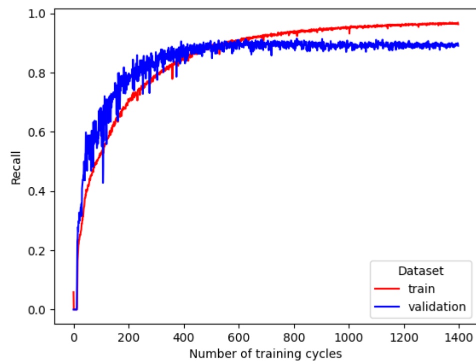
Figure 6.5: Graph and node level precision during training of the node classification method regarding the three reentrancy subtypes.



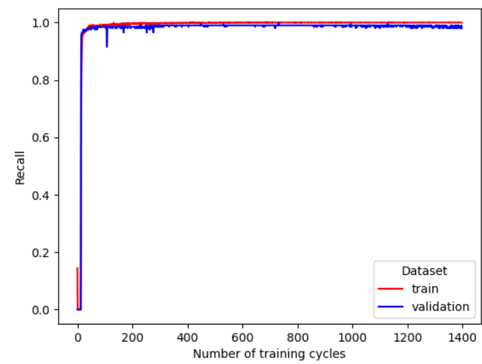
(a) Subtype *call* at node level



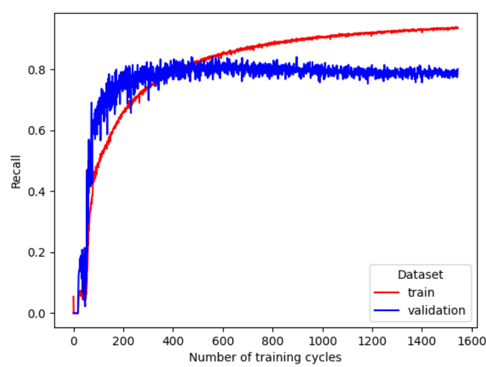
(b) Subtype *call* at graph level



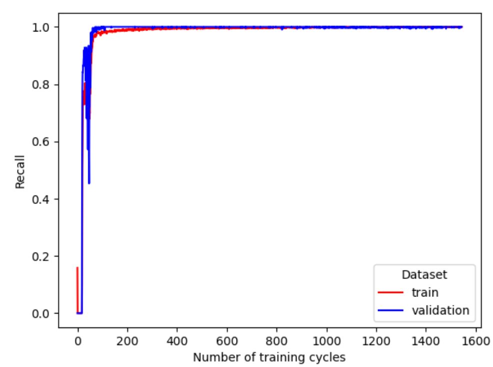
(c) Subtype *send* at node level



(d) Subtype *send* at graph level

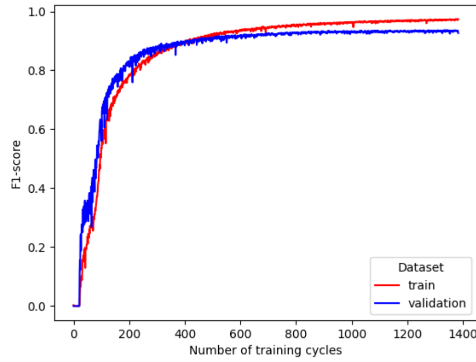


(e) Subtype *transfer* at node level

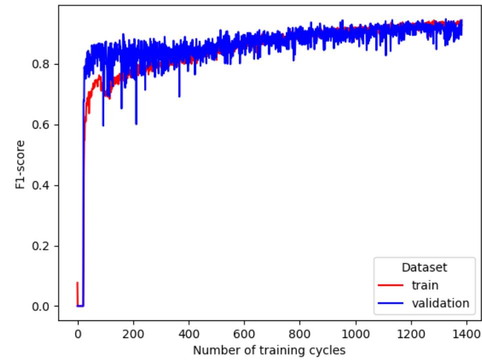


(f) Subtype *transfer* at graph level

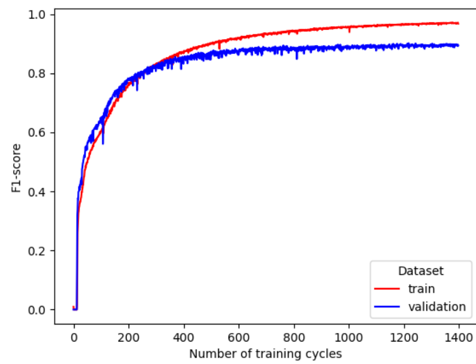
Figure 6.6: Graph and node level recall during training of the node classification method regarding the three reentrancy subtypes.



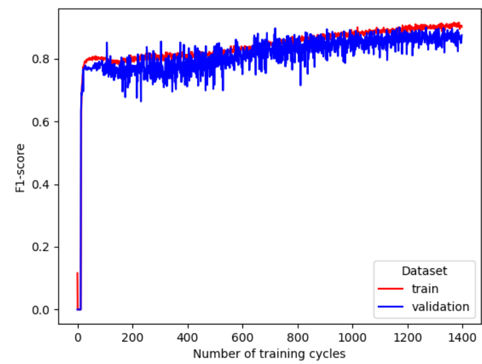
(a) Subtype *call* at node level



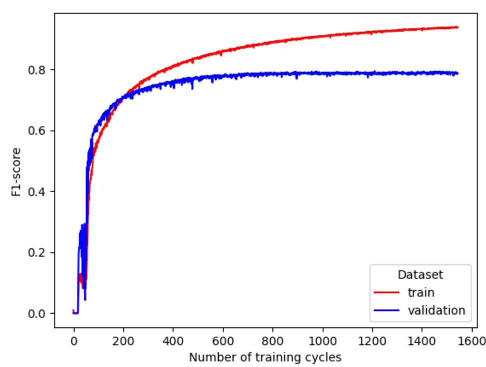
(b) Subtype *call* at graph level



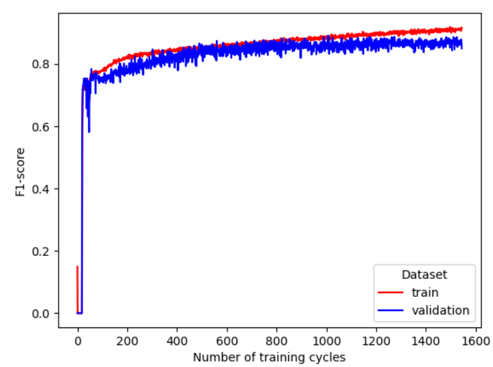
(c) Subtype *send* at node level



(d) Subtype *send* at graph level



(e) Subtype *transfer* at node level



(f) Subtype *transfer* at graph level

Figure 6.7: Graph and node level f1-scores during training of the node classification method regarding the three reentrancy subtypes.

Table 6.4: Results of the node classification method for the three reentrancy subtypes.

(a) Results on validation and test set regarding reentrancy subtype *call*.

	level	accuracy	precision	recall	f1-score
validation set	node	0.9995	0.9488	0.9252	0.9369
test set	node	0.9992	0.9323	0.8926	0.9120
validation set	graph	0.9833	0.8759	0.9917	0.9302
test set	graph	0.9731	0.8027	1.0000	0.8906

(b) Results on validation and test set regarding reentrancy subtype *send*.

	level	accuracy	precision	recall	f1-score
validation set	node	0.9986	0.9134	0.8905	0.9018
test set	node	0.9977	0.9255	0.8265	0.8732
validation set	graph	0.9472	0.8107	0.9850	0.8894
test set	graph	0.9546	0.8007	0.9956	0.8876

(c) Results on validation and test set regarding reentrancy subtype *transfer*.

	level	accuracy	precision	recall	f1-score
validation set	node	0.9952	0.7958	0.7919	0.7939
test set	node	0.9953	0.8361	0.7892	0.8119
validation set	graph	0.8991	0.7471	0.9969	0.8541
test set	graph	0.9074	0.7821	0.9855	0.8721

6.5 Time Consumption Test

The generation of the AST, the graph creation, and the prediction of the node labels are processes that take some time. After predicting the graph label by graph classification, GNNExplainer is applied to obtain the node labels. However, GNNExplainer is a neural network itself and needs to be trained for each sample. In fact, GNNExplainer uses the input graph and the trained graph classification model to approximate the parts of the input graph that are responsible for result. As this approximation or training time depends on the number of training steps, it is in general not useful to measure the time that is needed for a prediction with this method. Moreover, the predicition time is also influenced by the hardware that is used for training. An evaluation of the validation set with 100 training steps of GNNExplainer for each contract takes on the CPU about one and a half hour and on the GPU less than five minutes. Nevertheless, the time consumption of the second method with node classification for predicting node and graph labels is reliably measurable.

For this test a few contracts have been chosen with different contract sizes. The contract size is in this case given by the number of lines of the source code file. While classifying these contracts by node classification, the time is measured that is needed for the three main phases: AST generation, graph creation, and the classification process itself. The latter includes the time between entering the GNN and returning the line numbers. The classification process is measured with and without GPU support. The Table 6.5 shows that the entire procedure usually takes less than three seconds. With GPU support it is even faster. Moreover, it is noticeable that larger source code files also require more time for each of the three phases. This may be due to the fact that the compiler need more time to parse larger file. Furthermore, larger files usually contain more AST nodes which results in larger graphs and, thus, in more processing time for graph creation and classification. However, the method is still quite fast even with larger files.

Table 6.5: Time consumption of graph creation and prediction by node classification for different lengths of contract codes.

ID	Lines	Time for AST generation	Time for graph creation	Time for Prediction (CPU)	Time for Prediction (GPU)
1	6151	0.85070586 sec	0.19148588 sec	1.52419233 sec	0.10825682 sec
2	180	0.03155947 sec	0.00817943 sec	0.02713013 sec	0.00830913 sec
3	5817	0.89934659 sec	0.20776343 sec	1.22108746 sec	0.09555745 sec
4	7	0.00233436 sec	0.00083113 sec	0.00328064 sec	0.00481200 sec
5	5	0.00360012 sec	0.00119042 sec	0.00642180 sec	0.00505376 sec
6	5815	0.90198112 sec	0.17808032 sec	1.24745345 sec	0.09526324 sec
7	245	0.05589867 sec	0.01292014 sec	0.06594872 sec	0.01415920 sec
8	248	0.03695369 sec	0.01243591 sec	0.06062937 sec	0.01209402 sec
9	5034	0.84993196 sec	0.19599771 sec	1.14327025 sec	0.09277701 sec
10	197	0.03129363 sec	0.00604439 sec	0.02359438 sec	0.00996661 sec
11	6078	0.81858921 sec	0.18368864 sec	1.35770512 sec	0.10321331 sec
12	7	0.00288796 sec	0.00079179 sec	0.00292635 sec	0.00538826 sec
13	5	0.00284767 sec	0.00078106 sec	0.00263023 sec	0.00423813 sec
14	248	0.04019380 sec	0.01232624 sec	0.06026840 sec	0.01176572 sec
15	6	0.00303888 sec	0.00100374 sec	0.00319648 sec	0.00446224 sec

6.6 Result Comparison with Methods from Literature

Although the results of the node classification method seem to be quite good, it is still not clear how good they are compared to other methods known from literature. All vulnerability detection methods found in literature that are based on machine learning only consider reentrancy subtype *call* as reentrancy vulnerability. Among these methods there are only two methods that are able to predict the position of vulnerabilities, Mando [103] and SCScan [60]. SCScan has not been evaluated on line labels, but on only on the graph level. Thus, Mando is the only method that can be compared to the methods presented in this thesis regarding the localization performance. Mando is a graph based method that is also evaluated on the node level. However, the f1-score for the node level Mando provides in the paper is not higher than 86.4%. With 91.2% the node classification method elaborated in this thesis outperforms Mando. Nevertheless, the performance of the graph classification method is with a f1-score of not more than 9.0% much worse than the performance of Mando.

Table 6.6: Comparison of contract level results regarding reentrancy subtype *call*

method	line labels	accuracy	f1-score
Node Classification	✓	97.31%	89.06%
Graph Classification	✓	98.15%	91.74%
Mando [103]	✓	-	75.80%
DR-GCN [178]	-	81.47%	76.39%
TMP [178]	-	84.48%	78.11%
CGE [87]	-	89.15%	86.41%
AME [88]	-	90.19%	87.94%
Peculiar [162]	-	-	92.1%

Other vulnerability detection methods based on machine learning only provide labels for the entire contract or graph. The results of some selected methods regarding the reentrancy vulnerability are presented in Table 6.6. All methods shown are source code based and

use a graph approach to detect vulnerable contracts. Detection methods that are not based on a graph approach such as Mythril [101] or Oyente [93] are not listed in the table. Nevertheless, their detection performance is even lower than the performance of the here listed methods. As it can be noticed only the f1-score of Peculiar [162] is better than the f1-score of the two methods presented in this work. However, Peculiar does not provide any locations of the vulnerability in the source code, but only the information whether a contract is vulnerable or not. It is also striking that even the node classification method outperforms the other methods. This is interesting, because, in contrast to the other methods, node classification has only been trained on the node level and not on the graph level. Therefore, node classification not only offers more functionalities than other methods, but also shows higher performance in detecting contracts vulnerable to reentrancy.

6.7 Prediction Examples

The node classification method provides predicted node labels that can be transformed back into line number. These line number are used to mark the positions of a vulnerability in the contract. This section shows some examples of contracts where this is done. The ground truth in form of line labels is colored with a blue background. The lines that are marked by node classification are written in red. All examples of this section are labeled and analyzed with respect to reentrancy subtype *call*.

Figure 6.8(a) shows the first example contract that has been analyzed by the node classification method. It contains a line with false positive nodes. The line 1301 has been forgotten in the ground truth markings. However, this left out line is accidental and the classifier marked it anyway.

Figure 6.8(b) presents a contract that includes eight lines marked by the classifier only. In fact, these markings are false positives, since there is no money tranfer in the functions and, therefore, no reentrancy vulnerability. Probably, in the corresponding graph the node structure related to these lines is very similar to node structure belonging to vulnerabilities. Thus, these code sections looked like a vulnerability for the method and it marked them.

The contract of Figure 6.8(c) includes a money transfer, but there is no state change of variables afterwards. Nevertheless, line 353 is marked by the node classification method. This false positive marking is not that problematic, since the method provides the positions of the vulnerability.

The last Figure 6.8(d) shows a contract that seems to be marked equivalent to the ground truth. However, the accuracy on this contract is not 100% since not all nodes of the two lines are marked. It has to be noticed that *emit* statements are seen as state changes, although they do not affect any state of the EVM. However, these statements change the state of the blockchain, since the content of the messages that are emitted are saved there.

Most of the contracts are predicted and marked identically to the ground truth. A look into the contracts clarifies that the predicted markings of the remaining contracts differ only slightly from the markings made by hand and do not prevent the vulnerabilities from being found. In addition, contracts classified as false positive are of minor importance, since the source code section of the vulnerability can be inspected.

```
1281        // pay 3% out to community rewards
1282        uint256 _p1 = _eth / 100;
1283        uint256 _com = _eth / 50;
1284        _com = _com.add(_p1);
1285
1286        uint256 _p3d;
1287        if (!address(admin).call.value(_com)())
1288        {
1289            // This ensures Team Just cannot influence the outcome of FoMo3D with
1290            // bank migrations by breaking outgoing transactions.
1291            // Something we would never do. But that's not the point.
1292            // We spent 2000$ in eth re-deploying just to patch this, we hold the
1293            // highest belief that everything we create should be trustless.
1294            // Team JUST, The name you shouldn't have to trust.
1295            _p3d = _com;
1296            _com = 0;
1297        }
1298
1299
1300        // distribute share to affiliate
1301        uint256 _aff = _eth / 10;
1302
1303        // decide what to do with affiliate share of fees
1304        // affiliate must not be self, and must have a name registered
1305        if (_affID != _pID && plyr_[_affID].name != '') {
1306            plyr_[_affID].aff = _aff.add(plyr_[_affID].aff);
1307            emit F3Devents.onAffiliatePayout(_affID, plyr_[_affID].addr, plyr_[_affID].name, _rID, _pID, _aff, now);
1308        } else {
1309            _p3d = _aff;
1310        }
1311
1312        // pay out p3d
1313        _p3d = _p3d.add((_eth.mul(fees_[_team].p3d)) / (100));
1314        if (_p3d > 0)
1315        {
1316            // deposit to divies contract
1317            uint256 _potAmount = _p3d / 2;
1318
1319            admin.transfer(_p3d.sub(_potAmount));
1320
1321            round_[_rID].pot = round_[_rID].pot.add(_potAmount);
1322
1323            // set up event data
1324            _eventData_.P3DAmount = _p3d.add(_eventData_.P3DAmount);
1325        }
1326
1327        return(_eventData_);
1328    }
1329
1330    /**
1331     * @dev distributes eth based on fees to gen and pot
1332     */
1333    function distributeInternal(uint256 _rID, uint256 _pID, uint256 _eth, uint256 _team, uint256 _keys, F3Ddatasets.EventRetu
1334        private
```

(a) Example 1: The method marks one additional line, which has not been labeled before.

Figure 6.8: Some Prediction Examples - Part 1 of 4

```
459         change(_callDatas, _starts);
460
461         uint mtknTotalSupply = _mtkn.totalSupply(); // optimization totalSupply
462         uint256 bestAmount = uint256(-1);
463         for (uint i = _mtkn.tokensCount(); i > 0; i--) {
464             ERC20 token = _mtkn.tokens(i - 1);
465             if (token.allowance(this, _mtkn) == 0) {
466                 token.approve(_mtkn, uint256(-1));
467             }
468
469             uint256 amount = mtknTotalSupply.mul(token.balanceOf(this)).div(token.balanceOf(_mtkn));
470             if (amount < bestAmount) {
471                 bestAmount = amount;
472             }
473         }
474
475         require(bestAmount >= _minimumReturn, "buy: return value is too low");
476         _mtkn.bundle(msg.sender, bestAmount);
477     }
478
479     function buyFirstTokens(
480         IMultiToken _mtkn,
481         bytes _callDatas,
482         uint[] _starts // including 0 and LENGTH values
483     )
484         public
485         payable
486     {
487         change(_callDatas, _starts);
488
489         uint tokensCount = _mtkn.tokensCount();
490         uint256[] memory amounts = new uint256[](tokensCount);
491         for (uint i = 0; i < tokensCount; i++) {
492             ERC20 token = _mtkn.tokens(i);
493             amounts[i] = token.balanceOf(this);
494             if (token.allowance(this, _mtkn) == 0) {
495                 token.approve(_mtkn, uint256(-1));
496             }
497         }
498
499         _mtkn.bundleFirstTokens(msg.sender, msg.value.mul(1000), amounts);
500     }
501
```

(b) Example 2: No money transfer, but the code is marked by the method.

Figure 6.8: Some Prediction Examples - Part 2 of 4

```
335
336         // fire event
337         emit onNewName(_pID, _addr, _name, _isNewPlayer, _affID, plyr_[_affID].addr, plyr_[_affID].name, msg.value, now);
338     }
339 //==============================================================================
340 //     _|_ _  _ | _  .
341 //      | (_)(_)|_\  .
342 //==============================================================================
343
344      function updateFundAddress(address _newAddress)
345         onlyAdmin()
346         public
347     {
348         FundEIF = _newAddress;
349     }
350
351
352     function payFund() public {   //Registration fee goes to EIF - function must be called manually so enough gas is sent
353         if(!FundEIF.call.value(address(this).balance)()) {
354             revert();
355         }
356     }
357
358
359     function determinePID(address _addr)
360         private
361         returns (bool)
362     {
363         if (pIDxAddr_[_addr] == 0)
364         {
365             pID_++;
366             pIDxAddr_[_addr] = pID_;
367             plyr_[pID_].addr = _addr;
368
369             // set the new player bool to true
370             return (true);
371         } else {
372             return (false);
373         }
374     }
375 //==============================================================================
376 //   _    _|_ _  _ _  _ |    _ _ || _  .
377 //  (/_>< | (/_| | |(_||   (_(_|||_\  .
378 //==============================================================================
379     function getPlayerID(address _addr)
380         isRegisteredGame()
381         external
382         returns (uint256)
```

(c) Example 3: A false positive line which contains a money transfer but without state change afterwards.

Figure 6.8: Some Prediction Examples - Part 3 of 4

```
313         playerBalance[msg.sender] = 0;
314         msg.sender.transfer(_amount);
315
316         emit WithdrewBalance(msg.sender, _amount);
317     }
318
319     // PayThrone
320     // Sends thronePot to SnailThrone
321
322     function PayThrone() public {
323         uint256 _payThrone = thronePot;
324         thronePot = 0;
325         if (!SNAILTHRONE.call.value(_payThrone)()){
326             revert();
327         }
328
329         emit PaidThrone(msg.sender, _payThrone);
330     }
331
332     // fallback function
333     // Feeds the jackPot
334
335     function() public payable {
336         jackPot = jackPot.add(msg.value);
337
338         emit BoostedPot(msg.sender, msg.value);
339     }
340
341     //-- CALCULATIONS --
342
343     // ComputeEtherShare
344     // Returns ETH reward for a claim
345     // Reward = 0.00000002 ETH per treeSize per day
346
347     function ComputeEtherShare(address adr) public view returns(uint256) {
348
349         //Get time since last claim
350         uint256 _timeLapsed = now.sub(lastClaim[adr]);
```

(d) Example 4: Although the lines of the contract are identically marked as the ground truth, not all nodes derived from the lines are marked by the method.

Figure 6.8: Some Prediction Examples - Part 4 of 4

7. Conclusion

The observant reader may remember Alice who wanted to explore the capabilities of Ethereum smart contracts and, thus, wrote a contract detected as vulnerable. Alice could not find the bug and finally lost a lot of money because of that. If Alice had used the node classifier developed in this work, she would surely have found the vulnerability in the code.

In this thesis two methods for the detection of vulnerabilities in smart contracts are presented. Both methods are based on graphs and graph neural networks and are able to localize the positions of the vulnerabilities in the source code. Therefore, this work also provides a new type of contract graphs for smart contract source codes. The methods are tested on a dataset of 13,773 smart contract source codes. All source codes that can contain a reentrancy vulnerability are checked manually and if a vulnerability is present the positions are marked. The first method consisting of a graph neural network for graph classification and an explainer detects contracts containing reentrancy vulnerabilities with a f1-score of up to 97.12%, but shows poor performance in localizing the vulnerability with a f1-score of not more than 9.0%. The second method, a graph neural network for node classification, outperforms most detection methods with a f1-score of up to 93.02% at the identification of contracts with reentrancy vulnerabilities. Moreover, it localizes the vulnerability positions in the contract with a f1-score of up to 93.69% and returns them to the user as line numbers or marked sections in the source code. The method is fast with less than three second running time and can analyze contracts of different lengths. Therefore, programmers like Alice can find vulnerabilities in smart contract source codes much easier and faster in the future by using the node classification method.

In future works, this method can also be tested for other vulnerability types of smart contracts. Moreover, since the method makes currently use of a graph restricted to only one edge type, it can also be tried to extend the method in such a way that a restriction is no longer necessary. Furthermore, the underlying approach for localizing vulnerabilities can be transferred to other programming languages.

List of Figures

2.1 Functionality of a perceptron: An input vector is multiplied by weights and the output is generated by summing the results and a bias w_0. Then the label is determined by a threshold. 4
2.2 Linear Classification: Multiple perceptrons are used in a layer. The number of perceptrons coincides with the number m of possible classes. 5
2.3 A fully connected MLP 5
2.4 Computation Graph: Forward propagation in green, backpropagation in red; derivatives are computed with the last backpropagated value and the last forward propagated one, for example $\frac{dg}{dy} = \frac{\partial \log(y)}{\partial y}\Big|_{y=4} \cdot \frac{dg}{dz}$ 8
2.5 Illustration of the aggregation and combine steps of a two-layer GNN 10
2.6 Feature vectors that are used in each computation step of Figure 2.5 at node A 10
2.7 Parts of the AST of the source code of Listing 2.2 16

4.1 Control flow edges between k statements inside a block 25
4.2 Graph structure of the node types referring to branchings 27
4.3 Graph structure of the four loop types 28
4.4 Structure of the method based on graph classification and explanation . . . 34
4.5 Architecture of the GNN for node classification and transfer learning 36

5.1 Overview on the program 39

6.1 Confusion matrix of a model with binary outcome. 46
6.2 Graph level losses and f1-scores during training of the graph classification method regarding the three reentrancy subtypes. 49
6.3 Graph level precision and recall during training of the graph classification method regarding the three reentrancy subtypes. 50
6.4 Losses per node during training of the node classification method regarding the three reentrancy subtypes. 53
6.5 Graph and node level precision during training of the node classification method regarding the three reentrancy subtypes. 54
6.6 Graph and node level recall during training of the node classification method regarding the three reentrancy subtypes. 55
6.7 Graph and node level f1-scores during training of the node classification method regarding the three reentrancy subtypes. 56
6.8 Some Prediction Examples - Part 1 of 4 60
6.8 Some Prediction Examples - Part 2 of 4 61
6.8 Some Prediction Examples - Part 3 of 4 62
6.8 Some Prediction Examples - Part 4 of 4 63

List of Tables

4.1 Description of the node feature vectors 30

6.1 Composition of the dataset 48
6.2 Graph level results of graph classification for the three reentrancy subtypes. 51
6.3 Node level results of graph classification with GNNExplainer for the three reentrancy subtypes. 52
6.4 Results of the node classification method for the three reentrancy subtypes. 57
6.5 Time consumption of graph creation and prediction by node classification for different lengths of contract codes. 58
6.6 Comparison of contract level results regarding reentrancy subtype *call* . . . 58

Listings

2.1 Contract with reentrancy vulnerability in line 12 and 13. Source [126] . . . 14
2.2 Source code of an example contract 15

5.1 Conversion from line labels to node labels 43

Acronyms

MLP Multilayer Perceptron

GNN Graph Neural Network

EVM Ethereum Virtual Machine

AST Abstract Syntax Tree

GCN Graph Convolutional Network

PyG PyTorch Geometric

ReLU rectified linear unit

Bibliography

[1] blitz 1306: Repository solc-typed-ast. https://github.com/ConsenSys/solc-typed-ast, Last Access on December 26, 2022

[2] Albert, E., Fernández, J.C., Gordillo, P., Román-Díez, G., Rubio, A.: GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. TACAS **12079** (2020), https://arxiv.org/abs/1912.11929

[3] Albert, E., Gordillo, P., Livshits, B., Rubio, A., Sergey, I.: EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. Lect. Notes Comput. Sci. (2018), http://dx.doi.org/10.1007/978-3-030-01090-4_30

[4] Albert, E., Gordillo, P., Rubio, A., Schett, M.A.: Synthesis of Super-Optimized Smart Contracts Using Max-SMT. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. Springer International Publishing (2020)

[5] Albert, E., Gordillo, P., Rubio, A., Sergey, I.: Running on Fumes. In: Ganty, P., Kaâniche, M. (eds.) Verification and Evaluation of Computer and Communication Systems. Springer International Publishing (2019), https://www.springerprofessional.de/running-on-fumes/17375358

[6] Alexander Binder and Grégoire Montavon and Sebastian Bach and Klaus-Robert Müller and Wojciech Samek: Layer-wise Relevance Propagation for Neural Networks with Local Renormalization Layers. CoRR (2016), http://arxiv.org/abs/1604.00825

[7] Ali, A., Abideen, Z.U., Ullah, K., Ullah, F.: SESCon: Secure Ethereum Smart Contracts by Vulnerable Patterns' Detection. Secur. Commun. Networks **2021** (2021), https://doi.org/10.1155/2021/2897565

[8] Allamanis, M.: Graph Neural Networks in Program Analysis. In: Wu, L., Cui, P., Pei, J., Zhao, L. (eds.) Graph Neural Networks: Foundations, Frontiers, and Applications. Springer, Singapore (2021), https://graph-neural-networks.github.io/gnnbook_Chapter22.html

[9] Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to Represent Programs with Graphs. CoRR **abs/1711.00740** (2017), http://arxiv.org/abs/1711.00740

[10] Anaconda Inc.: Anaconda. https://www.anaconda.com/, Last Access on December 26, 2022

[11] Andrej Karpathy: CS231n: Convolutional Neural Networks for Visual Recognition - Course Notes. http://cs231n.github.io/, Last Access on December 26, 2022

[12] Antognini, D., Faltings, B.: Rationalization through Concepts. CoRR (2021), https://arxiv.org/abs/2105.04837

[13] Ashraf, I., Ma, X., Jiang, B., Chan, W.K.: GasFuzzer: Fuzzing Ethereum Smart Contract Binaries to Expose Gas-Oriented Exception Security Vulnerabilities. IEEE Access **8** (2020), https://doi.org/10.1109/ACCESS.2020.2995183

[14] Baldassarre, F., Azizpour, H.: Explainability Techniques for Graph Convolutional Networks. CoRR (2019), http://arxiv.org/abs/1905.13686

[15] Bastings, J., Aziz, W., Titov, I.: Interpretable Neural Predictions with Differentiable Binary Variables. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. ACL, Florence, Italy (2019), https://aclanthology.org/P19-1284

[16] Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. J. Mach. Learn. Res. **13** (feb 2012), https://dl.acm.org/doi/10.5555/2188385.2188395

[17] Bernhard Mueller and Daniel Luca: Advances in Automated EVM Smart Contract Vulnerability Detection and Exploitation. In: DEF CON 27. Las Vegas, CA, USA (2019), https://github.com/b-mueller/smashing-smart-contracts/blob/master/DEFCON27-EVM-Smart-Contracts-Mueller-Luca.pdf

[18] Binder, A., Bach, S., Montavon, G., Müller, K.R., Samek, W.: Layer-Wise Relevance Propagation for Deep Neural Network Architectures. In: Kim, K.J., Joukov, N. (eds.) Information Science and Applications (ICISA). Springer Singapore, Singapore (2016)

[19] Bishop, C.M.: Pattern Recognition and Machine Learning. Springer (2006)

[20] Bose, P., Das, D., Chen, Y., Feng, Y., Kruegel, C., Vigna, G.: SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds. CoRR **abs/2104.08638** (2021), https://arxiv.org/abs/2104.08638

[21] Brauckmann, A., Goens, A., Ertel, S., Castrillon, J.: Compiler-Based Graph Representations for Deep Learning Models of Code. In: Proceedings of the 29th International Conference on Compiler Construction. CC 2020, ACM, New York, NY, USA (2020), https://doi.org/10.1145/3377555.3377894

[22] Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., Scholz, B.: Vandal: A Scalable Security Analysis Framework for Smart Contracts. CoRR (2018), http://arxiv.org/abs/1809.03981

[23] Buterin, V.: Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform (2013), https://github.com/ethereum/wiki/wiki/White-Paper

[24] Castro, J., Gómez, D., Tejada, J.: Polynomial calculation of the Shapley value based on sampling. Computers and Operations Research **36**(5) (2009), https://www.sciencedirect.com/science/article/pii/S0305054808000804

[25] Cauchy, A., et al.: Méthode générale pour la résolution des systemes d'équations simultanées. Comp. Rend. Sci. Paris **25**(1847) (1847)

[26] Chang, J., Gao, B., Xiao, H., Sun, J., Cai, Y., Yang, Z.: sCompile: Critical Path Identification and Analysis for Smart Contracts. In: Ait-Ameur, Y., Qin, S. (eds.) Formal Methods and Software Engineering. Springer International Publishing (2019)

[27] Chang, S., Zhang, Y., Yu, M., Jaakkola, T.S.: Invariant Rationalization. CoRR (2020), https://arxiv.org/abs/2003.09772

[28] Chen, T., Feng, Y., Li, Z., Zhou, H., Luo, X., Li, X., Xiao, X., Chen, J., Zhang, X.: GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts. IEEE Trans. Emerg. Topics Comput. **9**(3) (2021), https://doi.org/10.1109/TETC.2020.2979019

[29] Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: 24th IEEE SANER (2017), https://doi.org/10.1109/SANER.2017.7884650

[30] Chen, T., Li, Z., Zhou, H., Chen, J., Luo, X., Li, X., Zhang, X.: Towards Saving Money in Using Smart Contracts. In: IEEE/ACM 40th ICSE-NIER (2018), `https://ieeexplore.ieee.org/document/8444844`

[31] Cheng, X., Wang, H., Hua, J., Xu, G., Sui, Y.: DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. ACM Trans. Softw. Eng. Methodol. **30**(3) (2021), `https://doi.org/10.1145/3436877`

[32] David E. Rumelhart and Geoffrey E. Hinton and Ronald J. Williams: Learning representations by back-propagating errors. Nature **323** (1986)

[33] Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res. **12** (2011), `http://dl.acm.org/citation.cfm?id=1953048.2021068`

[34] Ethereum Foundation: Ethereum Documentation, Consensus mechanisms, `https://ethereum.org/en/developers/docs/consensus-mechanisms/`, Last Access on December 26, 2022

[35] Ethereum Foundation: Ethereum Documentation, Nodes and Clients, `https://ethereum.org/en/developers/docs/nodes-and-clients/`, Last Access on December 26, 2022

[36] Ethereum Foundation: Solidity Documentation, `https://docs.soliditylang.org/en/v0.8.11/`, Last Access on December 26, 2022

[37] Fatima Samreen, N., Alalfi, M.H.: Reentrancy vulnerability identification in ethereum smart contracts. In: 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE) (2020), `https://doi.org/10.1109/IWBOSE50093.2020.9050260`

[38] Ferrante, J., Ottenstein, K.J., Warren, J.D.: The Program Dependence Graph and Its Use in Optimization. ACM Trans. Program. Lang. Syst. **9**(3) (1987), `https://doi.org/10.1145/24039.24041`

[39] Foundation, E.: Repository Solidity Compiler Solc, `https://github.com/ethereum/solidity/releases`, Last Access on December 26, 2022

[40] Frank, J., Aschermann, C., Holz, T.: ETHBMC: A Bounded Model Checker for Smart Contracts. In: 29th USENIX Security Symposium. USENIX Association (2020), `https://www.usenix.org/conference/usenixsecurity20/presentation/frank`

[41] Frank Rosenblatt: The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review **65(6)** (1958)

[42] Fred Oh: What is CUDA, `https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/`, Last Access on December 31, 2022

[43] Fu, M., Wu, L., Hong, Z., Zhu, F., Sun, H., Feng, W.: A Critical-Path-Coverage-Based Vulnerability Detection Method for Smart Contracts. IEEE Access **7** (2019), `https://ieeexplore.ieee.org/document/8867880`

[44] Fukushima, K.: Cognitron: A self-organizing multilayered neural network (sep 1975), `https://doi.org/10.1007/BF00342633`

[45] Funke, T., Khosla, M., Anand, A.: Hard Masking for Explaining Graph Neural Networks (2021), `https://openreview.net/forum?id=uDN8pRAdsoC`

[46] Ganz, T., Härterich, M., Warnecke, A., Rieck, K.: Explaining graph neural networks for vulnerability discovery. In: Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security. AISec '21, ACM, New York, NY, USA (2021), https://doi.org/10.1145/3474369.3486866

[47] Gao, Z., Jiang, L., Xia, X., Lo, D., Grundy, J.: Checking Smart Contracts With Structural Code Embedding. IEEE Transactions on Software Engineering **47**(12) (2021), http://dx.doi.org/10.1109/TSE.2020.2971482

[48] Giesen, J.R., Andreina, S., Rodler, M., Karame, G.O., Davi, L.: Practical mitigation of smart contract bugs. ArXiv (2022), http://arxiv.org/abs/2203.00364

[49] Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Teh, Y.W., Titterington, M. (eds.) Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics. Proceedings of Machine Learning Research, vol. 9. PMLR, Chia Laguna Resort, Sardinia, Italy (2010), https://proceedings.mlr.press/v9/glorot10a.html

[50] Google Developers: Classification: Accuracy. https://developers.google.com/machine-learning/crash-course/classification/accuracy, Last Access on December 26, 2022

[51] Google Developers: Classification: Precision and Recall. https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall, Last Access on December 26, 2022

[52] Google Developers: Classification: True vs. False and Positive vs. Negative. https://developers.google.com/machine-learning/crash-course/classification/true-false-positive-negative, Last Access on December 26, 2022

[53] Grech, N., Brent, L., Scholz, B., Smaragdakis, Y.: Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In: 41st IEEE/ACM ICSE (2019), https://doi.org/10.1109/ICSE.2019.00120

[54] Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: MadMax: Surviving Out-of-gas Conditions in Ethereum Smart Contracts. Proc. ACM Program. Lang. **2** (2018), http://doi.acm.org/10.1145/3276486

[55] Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: MadMax: Analyzing the out-of-Gas World of Smart Contracts. Commun. ACM **63**(10) (2020), https://doi.org/10.1145/3416262

[56] Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2020, ACM, New York, NY, USA (2020), https://doi.org/10.1145/3395363.3404366

[57] Grishchenko, I., Maffei, M., Schneidewind, C.: EtherTrust: Sound Static Analysis of Ethereum bytecode (2018), https://www.netidee.at/sites/default/files/2018-07/staticanalysis.pdf

[58] Grishchenko, I., Maffei, M., Schneidewind, C.: Foundations and Tools for the Static Analysis of Ethereum Smart Contracts. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. Springer International Publishing (2018), https://link.springer.com/chapter/10.1007/978-3-319-96145-3_4

[59] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S.K., Clement, C.B., Drain, D., Sundaresan, N., Yin, J., Jiang, D., Zhou, M.: GraphCodeBERT: Pre-training Code Representations with Data Flow. CoRR (2020), https://arxiv.org/abs/2009.08366

[60] Hao, X., Ren, W., Zheng, W., Zhu, T.: SCScan: A SVM-Based Scanning System for Vulnerabilities in Blockchain Smart Contracts. In: 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) (2020), https://doi.org/10.1109/TrustCom50675.2020.00221

[61] He, J., Balunović, M., Ambroladze, N., Tsankov, P., Vechev, M.: Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. CCS '19, ACM, New York, NY, USA (2019), https://doi.org/10.1145/3319535.3363230

[62] He, K., Zhang, X., Ren, S., Sun, J.: Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. CoRR (2015), http://arxiv.org/abs/1502.01852

[63] Huang, Q., Yamada, M., Tian, Y., Singh, D., Yin, D., Chang, Y.: GraphLIME: Local Interpretable Model Explanations for Graph Neural Networks. CoRR (2020), https://arxiv.org/abs/2001.06216

[64] Ian Goodfellow and Yoshua Bengio and Aaron Courville: Deep Learning. MIT Press (2016)

[65] Jain, S., Wallace, B.C.: Attention is not Explanation. In: NAACL (2019)

[66] Jain, S., Wiegreffe, S., Pinter, Y., Wallace, B.C.: Learning to Faithfully Rationalize by Construction. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. ACL (2020), https://aclanthology.org/2020.acl-main.409

[67] Jason Brownlee: Loss and Loss Functions for Training Deep Learning Neural Networks. https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/, Last Access on December 26, 2022

[68] Jiang, B., Liu, Y., Chan, W.K.: ContractFuzzer: fuzzing smart contracts for vulnerability detection. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (2018), http://dx.doi.org/10.1145/3238147.3238177

[69] Jonathan Hartley: Colorama. https://pypi.org/project/colorama/, Last Access on December 26, 2022

[70] Jordan, H., Scholz, B., Subotić, P.: Soufflé: On Synthesis of Program Analyzers. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. Springer International Publishing (2016), https://link.springer.com/chapter/10.1007/978-3-319-41540-6_23

[71] João F. Ferreira: SmartBugs Wild Dataset. https://github.com/smartbugs/smartbugs-wild, Last Access on December 26, 2022

[72] Jure Leskovec: Stanford CS224W: Machine Learning with Graphs - Lecture Notes, http://web.stanford.edu/class/cs224w/, Last Access on December 26, 2022

[73] Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: Analyzing Safety of Smart Contracts. In: 25th NDSS. San Diego, CA, USA (2018), https://subodhvsharma.github.io/publication/ndss18/

[74] Kingma, D., Ba, J.: Adam: A method for stochastic optimization. ICLR (2014)

[75] Kipf, T.N., Welling, M.: Semi-Supervised Classification with Graph Convolutional Networks. CoRR (2016), http://arxiv.org/abs/1609.02907

[76] Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., Saxena, P.: Exploiting the Laws of Order in Smart Contracts. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2019, ACM, New York, NY, USA (2019), https://doi.org/10.1145/3293882.3330560

[77] Krupp, J., Rossow, C.: teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In: 27th USENIX Security Symposium. USENIX Association, Baltimore, MD (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[78] Lapuschkin, S., Binder, A., Montavon, G., Müller, K.R., Samek, W.: The LRP Toolbox for Artificial Neural Networks. JMLR **17**(114) (2016), http://jmlr.org/papers/v17/15-618.html

[79] Lapuschkin, S., Binder, A., Montavon, G., Klauschen, F., Müller, K.R., Samek, W.: On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation. PLoS ONE **10** (2015), https://doi.org/10.1371/journal.pone.0130140

[80] Lapuschkin, S., Wäldchen, S., Binder, A., Montavon, G., Samek, W., Müller, K.: Unmasking Clever Hans Predictors and Assessing What Machines Really Learn. CoRR (2019), http://arxiv.org/abs/1902.10178

[81] Lei, T., Barzilay, R., Jaakkola, T.: Rationalizing Neural Predictions. In: Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing. ACL, Austin, Texas (2016), https://aclanthology.org/D16-1011

[82] Li, A., Long, F.: Detecting Standard Violation Errors in Smart Contracts. CoRR (2018), http://arxiv.org/abs/1812.07702

[83] Lin, W., Lan, H., Li, B.: Generative Causal Explanations for Graph Neural Networks. CoRR (2021), https://arxiv.org/abs/2104.06643

[84] Lipovetsky, S., Conklin, M.: Analysis of Regression in Game Theory Approach. Applied Stochastic Models in Business and Industry **17** (2001), https://doi.org/10.1002/asmb.446

[85] Liu, C., Liu, H., Cao, Z., Chen, Z., Chen, B., Roscoe, B.: ReGuard: Finding Reentrancy Bugs in Smart Contracts. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. ICSE '18, ACM, New York, NY, USA (2018), https://doi.org/10.1145/3183440.3183495

[86] Liu, Y., Chen, C., Liu, Y., Zhang, X., Xie, S.: Multi-objective Explanations of GNN Predictions. In: 2021 IEEE International Conference on Data Mining (ICDM) (2021), https://doi.org/10.1109/ICDM51629.2021.00052

[87] Liu, Z., Qian, P., Wang, X., Zhuang, Y., Qiu, L., Wang, X.: Combining Graph Neural Networks with Expert Knowledge for Smart Contract Vulnerability Detection. IEEE Transactions on Knowledge and Data Engineering **1**(01) (2021), https://doi.org/10.1109/TKDE.2021.3095196

[88] Liu, Z., Qian, P., Wang, X., Zhu, L., He, Q., Ji, S.: Smart Contract Vulnerability Detection: From Pure Neural Network to Interpretable Graph Feature and Expert Pattern Fusion. In: Zhou, Z.H. (ed.) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21. International Joint Conferences on Artificial Intelligence Organization (2021), https://doi.org/10.24963/ijcai.2021/379

[89] Lu, N., Wang, B., Zhang, Y., Shi, W., Esposito, C.: NeuCheck: A more practical Ethereum smart contract security analysis tool. Softw Pract Exp **51**(10) (2021), https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2745

[90] Lundberg, S.: Implementation of SHAP, https://github.com/slundberg/shap, Last Access on December 26, 2022

[91] Lundberg, S., Lee, S.I.: A Unified Approach to Interpreting Model Predictions (2017), http://arxiv.org/abs/1705.07874

[92] Luo, D., Cheng, W., Xu, D., Yu, W., Zong, B., Chen, H., Zhang, X.: Parameterized Explainer for Graph Neural Network. In: Proceedings of the 34th International Conference on Neural Information Processing Systems. NIPS'20, Curran Associates Inc., Red Hook, NY, USA (2020)

[93] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making Smart Contracts Smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16, ACM, New York, NY, USA (2016), https://doi.org/10.1145/2976749.2978309

[94] Marescotti, M., Blicha, M., Hyvärinen, A.E.J., Asadi, S., Sharygina, N.: Computing Exact Worst-Case Gas Consumption for Smart Contracts. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. Springer International Publishing (2018), https://link.springer.com/chapter/10.1007/978-3-030-03427-6_33

[95] Michael A. Nielsen: Neural Networks and Deep Learning. Determination Press (2015)

[96] Michael Sejr Schlichtkrull and Nicola De Cao and Ivan Titov: Interpreting Graph Neural Networks for NLP With Differentiable Edge Masking. CoRR (2020), https://arxiv.org/abs/2010.00577

[97] Mohankumar, A.K., Nema, P., Narasimhan, S., Khapra, M.M., Srinivasan, B.V., Ravindran, B.: Towards Transparent and Explainable Attention Models. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. ACL (2020), https://aclanthology.org/2020.acl-main.387

[98] Molnar, C.: Interpretable Machine Learning (2019), https://christophm.github.io/interpretable-ml-book/, Last Access on December 26, 2022

[99] Morris, C.: Graph Neural Networks: Graph Classification. In: Wu, L., Cui, P., Pei, J., Zhao, L. (eds.) Graph Neural Networks: Foundations, Frontiers, and Applications. Springer, Singapore, Singapore (2022), https://graph-neural-networks.github.io/gnnbook_Chapter9.html

[100] Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A.: Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In: 34th IEEE/ACM ASE (2019), https://doi.org/10.1109/ASE.2019.00133

[101] Mueller, B.: Smashing Ethereum Smart Contracts for Fun and Real Profit. In: 9th HITBSecConf. Amsterdam, Netherlands (2018), https://github.com/b-mueller/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf

[102] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2009), http://www.bitcoin.org/bitcoin.pdf

[103] Nguyen, H.H., Nguyen, N.M., Xie, C., Ahmadi, Z., Kudendo, D., Doan, T.N., Jiang, L.: Mando: Multi-level heterogeneous graph embeddings for fine-grained detection of smart contract vulnerabilities. ArXiv (2022), https://arxiv.org/abs/2208.13252

[104] Nguyen, T.D., Pham, L.H., Sun, J., Lin, Y., Minh, Q.T.: sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. 42nd IEEE/ACM ICSE (2020), https://doi.org/10.1145/3377811.3380334

[105] Nguyen, V.H., Tran, L.M.S.: Predicting Vulnerable Software Components with Dependency Graphs. In: Proceedings of the 6th International Workshop on Security Measurements and Metrics. MetriSec '10, ACM, New York, NY, USA (2010), https://doi.org/10.1145/1853919.1853923

[106] Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In: Proceedings of the 34th Annual Computer Security Applications Conference. ACSAC '18, ACM, New York, NY, USA (2018), https://doi.org/10.1145/3274694.3274743

[107] Numeroso, D., Bacciu, D.: MEG: Generating Molecular Counterfactual Explanations for Deep Graph Networks. In: 2021 International Joint Conference on Neural Networks (IJCNN) (2021), https://doi.org/10.1109/IJCNN52387.2021.9534266

[108] Paranjape, B., Joshi, M., Thickstun, J., Hajishirzi, H., Zettlemoyer, L.: An Information Bottleneck Approach for Controlling Conciseness in Rationale Extraction. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP). ACL (2020), https://aclanthology.org/2020.emnlp-main.153

[109] Pope, P.E., Kolouri, S., Rostami, M., Martin, C.E., Hoffmann, H.: Explainability Methods for Graph Convolutional Neural Networks. In: 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) (2019), https://doi.org/10.1109/CVPR.2019.01103

[110] Protocol Labs: IPFS, https://ipfs.tech/, Last Access on December 26, 2022

[111] ProtoSchool: What is a CID, https://proto.school/anatomy-of-a-cid/01, Last Access on December 26, 2022

[112] rashida048: Understanding Regularization in Plain Language, https://regenerativetoday.com/understanding-regularization-in-plain-language-l1-and-l2-regularization/, Last Access on December 26, 2022

[113] Remix Project Team: Remix IDE, https://remix-project.org/, Last Access on December 26, 2022

[114] Ribeiro, M.T., Singh, S., Guestrin, C.: "Why Should I Trust You?": Explaining the Predictions of Any Classifier. CoRR (2016), http://arxiv.org/abs/1602.04938

[115] SAGAR SHARMA: Activation Functions in Neural Networks. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6, Last Access on December 26, 2022

[116] Sagar Sharma: What the Hell is Perceptron. https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53, Last Access on December 26, 2022

[117] Schnake, T., Eberle, O., Lederer, J., Nakajima, S., Schütt, K.T., Müller, K., Montavon, G.: XAI for Graphs: Explaining Graph Neural Network Predictions by Identifying Relevant Walks. CoRR (2020), https://arxiv.org/abs/2006.03589

[118] Schneidewind, C., Grishchenko, I., Scherer, M., Maffei, M.: EThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. CCS '20, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3372297.3417250

[119] Schwarzenberg, R., Hübner, M., Harbecke, D., Alt, C., Hennig, L.: Layerwise Relevance Visualization in Convolutional Text Graph Classifiers. In: Proceedings of the Thirteenth Workshop on Graph-Based Methods for Natural Language Processing (TextGraphs-13). ACL, Hong Kong (2019), https://aclanthology.org/D19-5308

[120] Serrano, S., Smith, N.A.: Is Attention Interpretable? In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. ACL, Florence, Italy (jul 2019), https://aclanthology.org/P19-1282

[121] Shan, C., Shen, Y., Zhang, Y., Li, X., Li, D.: Reinforcement Learning Enhanced Explainer for Graph Neural Networks. In: Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems. Curran Associates, Inc. (2021), https://proceedings.neurips.cc/paper/2021/file/be26abe76fb5c8a4921cf9d3e865b454-Paper.pdf

[122] Shapley, L.S.: A Value for N-Person Games. RAND Corporation, Santa Monica, CA" (1952), https://doi.org/10.7249/P0295

[123] She, D., Krishna, R., Yan, L., Jana, S., Ray, B.: MTFuzz: Fuzzing with a Multi-Task Neural Network. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2020, ACM, New York, NY, USA (2020), https://doi.org/10.1145/3368089.3409723

[124] She, D., Pei, K., Epstein, D., Yang, J., Ray, B., Jana, S.S.: NEUZZ: Efficient Fuzzing with Neural Program Smoothing. IEEE Symposium on Security and Privacy (SP) (2019), https://doi.org/0.1109/SP.2019.00052

[125] Shrikumar, A., Greenside, P., Kundaje, A.: Learning Important Features Through Propagating Activation Differences. CoRR (2017), http://arxiv.org/abs/1704.02685

[126] SmartContractSecurity: SWC Registry, https://swcregistry.io/, last Access on December 26, 2022

[127] So, S., Hong, S., Oh, H.: SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In: 30th USENIX Security Symposium. USENIX Association (2021), https://www.usenix.org/conference/usenixsecurity21/presentation/so

[128] Solc Developers: Add AST output. https://github.com/ethereum/solidity/pull/7387, Last Access on December 26, 2022

[129] Solc Developers: What is the meaning of src in solc –ast-json output. https://github.com/ethereum/solidity/issues/11921, Last Access on December 26, 2022

[130] Soto, D., Bergel, A., Hevia, A.G.: Fuzzing to Estimate Gas Costs of Ethereum Contracts. IEEE ICSME (2020), https://doi.org/10.1109/ICSME46990.2020.00073

[131] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: A simple way to prevent neural networks from overfitting. JMLR **15** (2014), http://dl.acm.org/citation.cfm?id=2627435.2670313

[132] Stanford Online: Stanford CS224W: Machine Learning with Graphs, https://www.youtube.com/playlist?list=PLoROMvodv4rPLKxIpqhjhPgdQy7imNkDn, Last Access on December 26, 2022

[133] Suiche, M.: Porosity: A Decompiler For Blockchain-Based Smart Contracts Bytecode. In: DEF CON 25. Las Vegas, CA, USA (2017), https://github.com/comaeio/porosity/tree/master/defcon2017

[134] Sundararajan, M., Taly, A., Yan, Q.: Axiomatic Attribution for Deep Networks. In: Proceedings of the 34th International Conference on Machine Learning - Volume 70. ICML'17, JMLR.org, Sydney, NSW, Australia (2017)

[135] Suneja, S., Zheng, Y., Zhuang, Y., Laredo, J., Morari, A.: Learning to map source code to software vulnerability using code-as-a-graph. CoRR (2020), https://arxiv.org/abs/2006.08614

[136] Tang, J., Liao, R.: Graph Neural Networks for Node Classification. In: Wu, L., Cui, P., Pei, J., Zhao, L. (eds.) Graph Neural Networks: Foundations, Frontiers, and Applications. Springer, Singapore, Singapore (2022), https://graph-neural-networks.github.io/gnnbook_Chapter4.html

[137] Team, E.S.C.B.P.: Reentrancy, https://consensys.github.io/smart-contract-best-practices/attacks/reentrancy/, last Access on December 26, 2022

[138] The Etherscan Team: Etherscan, The Ethereum Blockchain Explorer. https://etherscan.io/

[139] The LLVM Team: Clang Compiler. https://clang.llvm.org/, Last Access on December 26, 2022

[140] The NumPy Team: NumPy. https://numpy.org/, Last Access on December 26, 2022

[141] The PyG Team: PyTorch Geometric. https://www.pyg.org/, Last Access on December 26, 2022

[142] The scikit-learn Team: F1-score. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html, Last Access on December 26, 2022

[143] The scikit-learn Team: Scikit-learn. https://scikit-learn.org/stable/index.html, Last Access on December 26, 2022

[144] Tieleman, T., Hinton, G.: Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude (2012)

[145] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: SmartCheck: Static Analysis of Ethereum Smart Contracts. In: 1st IEEE/ACM WETSEB (2018), https://ieeexplore.ieee.org/document/8445052

[146] Timothy Prescott: The Multivariable Chain Rule. https://sites.und.edu/timothy.prescott/apex/web/apex.Ch13.S5.html, Last Access on December 26, 2022

[147] Torch Contributors: PyTorch. https://pytorch.org/, Last Access on December 26, 2022

[148] Torres, C.F., Iannillo, A.K., Gervais, A., State, R.: ConFuzzius: Towards Smart Hybrid Fuzzing for Smart Contracts. CoRR **abs/2005.12156** (2020), https://arxiv.org/abs/2005.12156

[149] Torres, C.F., Schütte, J., State, R.: Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference. ACSAC '18, ACM, New York, NY, USA (2018), https://doi.org/10.1145/3274694.3274737

[150] Torres, C.F., Steichen, M., State, R.: The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. In: 28th USENIX Security Symposium. USENIX Association, Santa Clara, CA, USA (2019), https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira

[151] Tsankov, P.: Security Analysis of Smart Contracts in Datalog. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. Springer International Publishing (2018)

[152] Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.T.: Securify: Practical Security Analysis of Smart Contracts. CoRR (2018), http://arxiv.org/abs/1806.01143

[153] Štrumbelj, E., Kononenko, I.: Explaining Prediction Models and Individual Predictions with Feature Contributions. Knowl. Inf. Syst. **41**(3) (2014), https://doi.org/10.1007/s10115-013-0679-x

[154] Vu, M.N., Thai, M.T.: PGM-Explainer: Probabilistic Graphical Model Explanations for Graph Neural Networks. In: Proceedings of the 34th International Conference on Neural Information Processing Systems. NIPS'20, Curran Associates Inc., Red Hook, NY, USA (2020)

[155] Wang, A., Wang, H., Jiang, B., Chan, W.K.: Artemis: An Improved Smart Contract Verification Tool for Vulnerability Detection. 7th DSA (2020), https://doi.org/10.1109/DSA51864.2020.00031

[156] Wang, W., Song, J., Xu, G., Li, Y., Wang, H., Su, C.: ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts. IEEE Transactions on Network Science and Engineering **8**(2) (2021), https://doi.org/10.1109/TNSE.2020.2968505

[157] Wang, X., Wu, Y., Zhang, A., Feng, F., He, X., Chua, T.S.: Reinforced Causal Explainer for Graph Neural Networks. IEEE Transactions on Pattern Analysis and Machine Intelligence (2022), https://doi.org/10.1109/TPAMI.2022.3170302

[158] Wang, X., Wu, Y., Zhang, A., He, X., seng Chua, T.: Causal Screening to Interpret Graph Neural Networks (2021), https://openreview.net/forum?id=nzKv5vxZfge

[159] Wiegreffe, S., Pinter, Y.: Attention is not not Explanation. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP). ACL, Hong Kong, China (2019), https://aclanthology.org/D19-1002

[160] Wilson, R.J.: Introduction to Graph Theory. Prentice Hall/Pearson, New York (2010)

[161] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (2014), https://ethereum.github.io/yellowpaper/paper.pdf

[162] Wu, H., Zhang, Z., Wang, S., Lei, Y., Lin, B., Qin, Y., Zhang, H., Mao, X.: Peculiar: Smart Contract Vulnerability Detection Based on Crucial Data Flow Graph and Pre-training Techniques. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE). IEEE (2021), https://shangwenwang.github.io/files/ISSRE-21.pdf

[163] Wu, L., Cui, P., Pei, J., Zhao, L.: Graph Neural Networks: Foundations, Frontiers, and Applications. Springer Singapore, Singapore (2022)

[164] Wüstholz, V., Christakis, M.: Harvey: A Greybox Fuzzer for Smart Contracts. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, New York, NY, USA (2020), https://doi.org/10.1145/3368089.3417064

[165] Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? CoRR (2018), http://arxiv.org/abs/1810.00826

[166] Xu, P., Khairi, A.E.: Android-COCO: Android Malware Detection with Graph Neural Network for Byte- and Native-Code (2021), https://arxiv.org/abs/2112.10038v1

[167] Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and Discovering Vulnerabilities with Code Property Graphs. In: 2014 IEEE Symposium on Security and Privacy (2014), https://doi.org/10.1109/SP.2014.44

[168] Ying, R., Bourgeois, D., You, J., Zitnik, M., Leskovec, J.: GNNExplainer: Generating Explanations for Graph Neural Networks. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems. Curran Associates Inc., Red Hook, NY, USA (2019)

[169] Yu, M., Chang, S., Zhang, Y., Jaakkola, T.: Rethinking Cooperative Rationalization: Introspective Extraction and Complement Control. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP). ACL, Hong Kong, China (2019), https://aclanthology.org/D19-1420

[170] Yuan, G.X., Ho, C.H., Lin, C.J.: Recent advances of large-scale linear classification (09 2012)

[171] Yuan, H., Tang, J., Hu, X., Ji, S.: XGNN: Towards Model-Level Explanations of Graph Neural Networks. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, New York, NY, USA (2020), https://doi.org/10.1145/3394486.3403085

[172] Yuan, H., Yu, H., Wang, J., Li, K., Ji, S.: On Explainability of Graph Neural Networks via Subgraph Explorations. CoRR (2021), https://arxiv.org/abs/2102.05152

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

[174] Zhang, Q., Wang, Y., Li, J., Ma, S.: EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts. 27th IEEE SANER (2020), https://doi.org/10.1109/SANER48275.2020.9054822

[175] Zhou, E., Hua, S., Pi, B., Sun, J., Nomura, Y., Yamashita, K., Kurihara, H.: Security Assurance for Smart Contract. In: 9th IFIP NTMS (2018), https://doi.org/10.1109/NTMS.2018.8328743

[176] Zhou, K., Cheng, J., Li, H., Yuan, Y., Liu, L., Li, X.: SC-VDM: A Lightweight Smart Contract Vulnerability Detection Model. In: Tan, Y., Shi, Y., Zomaya, A., Yan, H., Cai, J. (eds.) Data Mining and Big Data. Springer Singapore, Singapore (2021)

[177] Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In: NeurIPS (2019), https://proceedings.neurips.cc/paper/2019/file/49265d2447bc3bbfe9e76306ce40a31f-Paper.pdf

[178] Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X., He, Q.: Smart Contract Vulnerability Detection using Graph Neural Network. In: Bessiere, C. (ed.) Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20. International Joint Conferences on Artificial Intelligence Organization (2020), https://doi.org/10.24963/ijcai.2020/454

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Würzburg, 02. January 2023

. . . .
(Alexander Hefter)