Master Thesis

Detection of Software Vulnerabilities in Smart Contracts using Deep Learning

Oliver Lutz
Department of Computer Science
Chair of Computer Science II (Secure Software Systems)

Prof. Dr.-Ing. Alexandra Dmitrienko
First Reviewer and Adviser

Prof. Dr.-Ing. Samuel Kounev
Second Reviewer

Submission
19. October 2020

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.

Wuerzburg, 19. October 2020

. . . .

(Oliver Lutz)

Abstract

The abuse of programming flaws in cryptocurrency platforms can lead to big economic damage. Therefore, this work analyzes how deep-learning can be utilized to detect vulnerabilities in Ethereum smart contracts. As a result, a framework is developed to take care of data retrieval, pre-classification, and model building. With this framework, deep-learning models can be developed, trained, and evaluated to create an AI scanner. Even with simple neural-network architectures, an AI scanner, capable of detecting eight different vulnerability types, achieves an accuracy of 98%. This AI scanner can be deployed for further usage. However, it can also be extended by new vulnerability types to create an even more powerful scanner.

Zusammenfassung

Der Missbrauch von Programmierfehlern in Kryptowährungsplattformen kann zu großen wirtschaftlichen Schäden führen. Deshalb wird in dieser Arbeit analysiert, wie Deep-Learning genutzt werden kann, um Schwachstellen in Ethereum Smart Contracts aufzudecken. Als Ergebnis ist ein Framework entwickelt worden, das sich um die Datengewinnung, Vorklassifizierung und Modellbildung kümmert. Mit diesem Framework können Deep-Learning-Modelle entwickelt, trainiert und evaluiert werden, um einen KI-Scanner zu erstellen. Selbst bei einfachen neuronalen Netz-Architekturen erreicht der KI-Scanner, der acht verschiedene Arten von Schwachstellen erkennen kann, eine Genauigkeit von 98%. Dieser KI-Scanner kann zur weiteren Verwendung eingesetzt werden. Er kann jedoch auch um Schwachstellentypen erweitert werden, um ein noch mächtigeres Tool zu bilden.

Contents

1. Introduction

Since the Bitcoin [1] has been introduced in 2009, the interest and the attention to using electronic currencies are rising. By using an innovative underlying technology, called blockchain, Bitcoin is an entirely decentralized cryptocurrency and does not rely on trusted third parties, such as banks [1]. Furthermore, the technology around blockchain becomes more and more popular, which led to other variants of cryptocurrency platforms, such as Ethereum. In Ethereum, the focus also lies on creating fully automated contracts on top of the blockchain technology, the so-called smart contracts. With smart contracts, plenty of new behavior can be built on top of the blockchain. Since this technology is available to everyone, it is also easy to create and deploy own smart contract. In the process of developing a smart contract, it is crucial to pay attention to that the smart contracts do not open up any exploitability. Caused by the impossibility to revert certain actions, it is key that users can trust the reliability of smart contracts.

As indicated, this technology can lead to economic consequences caused by software vulnerabilities in smart contracts. Ethereum operates on open networks where everyone can join without trusted third parties. Therefore, the lack of code quality can lead to unforeseen security problems. The abuse of programming flaws, like the reentrancy bug, has been exploited to steal 60 million US dollars [2, 3]. This is just one of many attack possibilities, which have occurred due to several security vulnerabilities.

Consequently, mitigation strategies have to be researched, which will be the focus of this work. However, flaws of smart contracts are hard to mitigate if those are already deployed on the blockchain. In addition to that, the automation of finding vulnerabilities can be intricate, which is why experts often offer audits to increase security. Nevertheless, automated tools for scanning smart contracts are developed for the detection of programming flaws in smart contracts to support programmers during the development or in the pre-deployment phase. As a result, the flaws of programs can be detected and, hence, mitigated [4]. For that purpose, different approaches are already realized, such as symbolic execution, SMT solving, taint analysis, runtime monitoring, and fuzzing, which all have advantages and disadvantages.

A relatively new approach will be investigated, which is detecting vulnerabilities of smart contracts by utilizing the power of deep-learning. Since machine-learning gained popularity and turned out to be effective in vulnerability detection in other software, such as C and C++ [5], this approach is promising in solving the problem of vulnerability detection in smart contracts as well. With this work, an artificial intelligence (AI) scanner is developed capable of classifying smart contracts assigning the vulnerability type. It includes multiple vulnerability types combined with multiple tools. The only requirement for this approach is data. This data is retrieved from the publicly available blockchain. This way, a huge dataset is available, which can be used to learn the warnings for possible attacking points. Since only bytecode is publicly available, it will be used as a base for the deep-learning approach. This might also improve the accuracy since these bytes will not be processed to a low-level-language. This AI Scanner can be easily extended. This way, the AI scanner can develop into a powerful tool. In the end, a framework is developed which can be used for the retrieval of smart contracts' bytecode and extended by every imaginable neural-network architecture. This way, the existing models can be easily improved for future challenges.

This work is structured as follows. Firstly, background information about Ethereum, the smart contracts, will be introduced. It will be examined how simple code flaws can cause a financial crisis. Afterward, the basic concepts of deep-learning are introduced, which have been required for this work's realization. In the next section, existing tools and approaches are presented before we dive into the concepts and approaches of this work. Thereafter, the realization and implementation details are described. The models developed in this project are then evaluated against standard deep-learning metrics and compared with existing tools. In the end, this work will be concluded.

2. Background

Before we dive deeper into the details of this work, multiple background information is required. Therefore, this chapter will introduce the basic concepts of several domains. First of all, cryptocurrency platforms will be introduced, especially the Ethereum platform. In that context, smart contracts and the potential to exploit their vulnerabilities will be examined. Since this work is based only on Ethereum's smart contracts, the focus will lie on that cryptocurrency technology. Afterward, the base concepts and terminologies of machine-learning, especially deep-learning, will be presented. The standard approach of developing a deep-learning model, including its key factors, will be explored. Understanding these concepts is fundamental to move forward in this project.

2.1 Ethereum Platform

Before introducing smart contracts, we should take a brief look at cryptocurrency platforms using blockchain technology. These systems for cryptocurrencies, such as Bitcoin or Ethereum, can be described by the following key characteristics:

Decentralized Nature:

In contrast to conventional currencies, virtual money is not administered by a central authority but by a distributed peer-to-peer network. Therefore, a network of nodes, the so-called miners, takes care of performing money transactions, data storage, and updates. They are running the software to communicate with the network. All code, data, and transactions are shared and available for inspection on every single node. Everyone is allowed to submit actions on the blockchain since there is no single administrator. All actions performed inside of this network have to be confirmed by the majority of all participating nodes [6, 7, 8].

Mathematical Algorithm as a Basis of Cryptocurrency Value:

Money, in the Ethereum context called Ether, can be initially earned by solving a complex mathematical problem that can be accepted by the other nodes, the so-called mining. Once a transaction is triggered, a state value for the new block is calculated and stored for confirmation. By mathematical calculations, this transaction's validity is also verified by the nodes participating in the network. It is also ensured that there is a limited amount of money available inside the network. This way, the amount of money, including its owner, can be tracked at all times [6, 7].

Resilience to Data Manipulations from Outside:

The information of mined money is stored in a public data structure, the blockchain. Without going deeper into the blockchain technology, modifications and transactions are stored in blocks added to the chronologically ordered chain. This order represents the state transitions, such as an update of money ownership. Before a block is accepted in the blockchain, it is ensured the state is valid according to the existing chain. Therefore, the reference to the previous block is validated as well as the mathematically calculated results. Once the majority of the network accepts a new block, it is appended at the end of the blockchain. Afterward, it is impossible to rewrite or modify any information on the blockchain. The network will reject any attempts to modify blockchain entries. The data is, therefore, immutable and irreversible [6, 7, 8].

Pseudonymous Nature:

In general, there is no special registration necessary to use cryptocurrencies. Users performing actions inside the network are identified by a public key and a private key. All transactions are related to addresses instead of explicit users. Therefore, it can be hard to determine a user, although all transactions are stored publicly on the blockchain [6, 7].

Since the base concepts of the Ethereum platform are clear at this point, the smart contracts and their vulnerabilities are examined in the next section.

2.1.1 Smart Contracts

After introducing these essential concepts, we take a more in-depth look into the Ethereum network. The focus of Ethereum lies in providing a Turing-complete programming language. This language is used to write smart contracts. These smart contracts define custom rules for money ownership, transactions, or state updates on the blockchain. More detailed, a smart contract is a program that can be called by its address to run operations on the blockchain. Its code is defined by high-level programming languages such as Solidity [9]. Once compiled, a bytecode is generated. This bytecode will be executed inside of so-called Ethereum virtual machines. Since every byte of the smart contract represents a defined operation, the program flow can be reconstructed into an assembly-like code. This way, it is possible to analyze smart contracts even in the form of bytecode.

When triggered, the execution is autonomous and enforceable for all participating parties of smart contracts. Therefore, no person can influence the performance of an agreement. The execution of a transaction costs a price, the so-called gas. Smart contracts are stored on the blockchain and can not be removed or modified, as explained in the previous section. However, they are transparent to all users, and there is no party to trust but the algorithms of a computer. ”Code is law” applies in that context in the sense that if all conditions are fulfilled, the execution of the smart contracts is guaranteed [6, 10, 7, 8, 11, 12].

Before we move on to an illustration of potential vulnerabilities of smart contracts, we have a more in-depth technical look at smart contracts and their storage. Internally, state transitions are represented by a certain type of object, the so-called accounts. These accounts hold information about the nounce, the ether balance, and the storage. The nounce is a counter variable used to validate transactions. In parallel to that, smart contracts are accounts as well. In that case, the account stores the information of the bytecode. This account cannot be manipulated or controlled by a user. Therefore, instead of running a transaction calling a regular user account address, a transaction is triggered, calling the smart contract’s account address. This way, a smart contract is integrated into the blockchain [6, 10]. Information and results of a transaction are stored in so-called receipts and are bound to the transaction objects. Transactions, on the other hand, are stored in the blocks. This way, it is possible to collect information about smart contracts and find the bytecode. This approach is implemented by the tool ethereum-etl, which will be used in this project [13]. Detailed behavior and workflow are examined in Section 5.2.2. In the next section, we will look into vulnerabilities of smart contracts.

2.1.2 Vulnerabilities of Smart Contracts

As introduced in the previous section, smart contracts are stored on the blockchain. So if the code of smart contracts opens up any vulnerabilities, it is hard to mitigate these flaws. It can end up quite costly if an attacker successfully exploits any bugs when the network confirms the operations on the blockchain. That is why there are many tools and approaches to identify vulnerabilities before deploying smart contracts onto the blockchain.

In the following, one representative vulnerability type will be explained. Let us take a look at two contracts written in Solidity, shown in Figure 2.1. The Victim contract contains a withdraw() function that transfers Ether, Ethereum's currency, from the callee to the caller. When this function is executed, the caller's fallback function is invoked. Consequently, by recursively calling the withdraw() function again inside the fallback function, more money will be transferred [11, 14]. This way, 60 million US dollars have been stolen from the DAO contract [15, 16].

```
contract Victim {
  bool flag = false;
  function withdraw() {
      if (flag || !msg.sender.call.value(1 wei)()) throw;
      flag = true;
  }
}
contract Attacker {
  uint count = 0;
  function() payable {
      if(++count < 10) Victim(msg.sender).withdraw();
  }
}
```

Figure 2.1: A simplified version of an attack and victim contract which exploits the reentrancy bug written in Solidity [14].

This vulnerability type is just one of many existing program flaws, leading to critical damage to the whole system. In the further progress of this work, plenty of other vulnerability types will be investigated. In the end, an AI Scanner is developed capable of detecting a specific set of vulnerability types in smart contracts as well. Before we can start investigating this project's approach, we first take a look at the concepts of deep-learning in general.

2.2 Deep-Learning

In many areas, deep-learning has become a buzzword. In current days, data is collected everywhere to be utilized for complex tasks. Deep-learning can be found in many different areas, such as image or speech recognition, text classification, and many more. In this section, the main concepts of deep-learning will be introduced. The chapter starts with an introduction to machine-learning before moving to neural networks and natural language processing.

2.2.1 Introduction into Machine-Learning

Machine-learning is used in many ways based on the important key factor, the data. This approach automates problem solutions, such as classification, where objects are labeled in different categories, such as spam mail or regular mail. Nevertheless, it can also be set up as a prediction tool for stock prices using regression. Those are two scenarios where machine-learning turned out to be an effective solution. There are many more use cases where machine-learning can solve problems. However, since this work is about detecting smart contracts' vulnerability types, our focus lies on classification. In general, the most common machine-learning approaches can be divided into supervised and unsupervised learning.

Supervised Learning:

In the case of supervised learning, a model is trained by a set of labeled data. This data represents a mapping of an input and its expected classification output. The algorithm's task is to detect patterns, the so-called features, indicating which label should be selected. The features are characteristics that are used to distinguish data objects. Data objects are transformed into a feature-vector-based representation so that the algorithm can learn which feature correspond to the respective classification label. During this process, metrics are adjusted to the model. This way, it can be trained to adjust internal adjustable parameters to minimize the error and maximize the score. In the context of deep-learning, these parameters are called weights. Thus, the quality of a machine-learning performance relies on feature engineering, including feature generation, selection, analysis, evaluation, and many more, to provide as meaningful features as possible [17, 18, 19].

Unsupervised Learning:

In contrast to supervised learning, an unlabeled data set is provided to the algorithm in unsupervised learning. With this approach, the model learns to identify similarities and differences so that the input is grouped. Since there is no measurable accuracy, the resulting groups' quality depends on the data and the algorithm's structure. This strategy is often used to solve clustering problems [17, 18, 19].

Since it is more promising to use supervised learning, this work focuses on that approach. The machine-learning workflow can be divided into several phases, which are illustrated in Figure 2.2. At first, a dataset must be available, which might be labeled dependent on the strategy and the learning algorithm's purpose. It is useful to divide the data into training and test set. Afterward, the data must be cleaned to remove unnecessary parts that are not relevant for the classification to improve the algorithm's efficiency. After that, the data must be transformed into vectors, which can be processed for the equations used to learn the data patterns. Thereafter, a model is built, which is trained and adjusted using the training data set [17, 18, 19]. The test set is used to evaluate the model's accuracy, which often shows an overfitting problem. Overfitting means that the model is too specialized in predicting the training data set; however, it performs poorly on unknown data. Therefore,

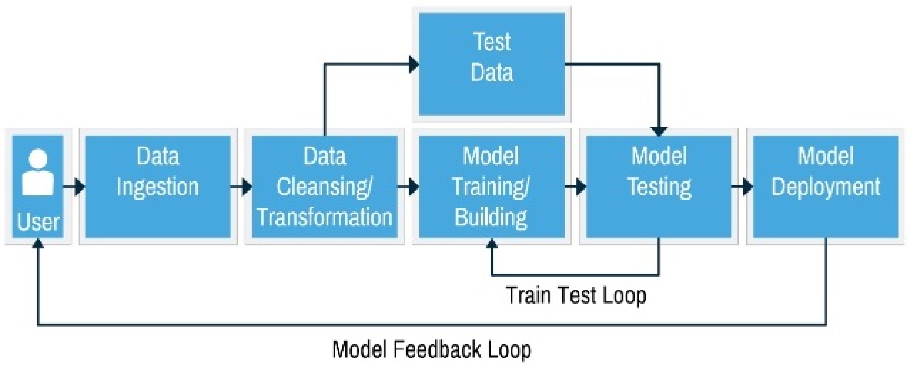


Figure 2.2: This is a basic overview of the workflow during the machine-learning development [17].

the model's parameters need to be adjusted, or the data input needs to be optimized [20]. When the accuracy meets its requirements, the model can be finally deployed into an application. In the context of classification, models can be distinguished into different classes. The first one is a multiclass classification, which means that the model's output is not binary. In detail, a model can predict more than two different classification classes. On the other hand, in multilabel classification use cases, multiple label classes can be assigned to a sample. In the case of multiclass-multilabel classifications, multiple non-binary labels can be assigned to a label [21]. The type of multilabel model applies to our project. Since this project will use neural networks for learning an AI Scanner, the classic machine-learning variants and implementations will not be introduced. However, neural networks will be examined in the next sections.

2.2.2 Neural Networks

One machine learning method is deep learning, which is based on neural networks. These networks are designed in a way so that they can learn and solve various problems. The architecture is a composition of multiple layers of different modules. A certain number of layers form a neural network. The first layer is called the input layer and is used for passing data to a neural network. In contrast to that, the last layer is called the output layer. The input layer and the output layer are connected with so-called hidden layers. Each layer contains a certain amount of nodes called neurons. One neuron calculates an output value, which will be passed to the next layer. Depending on the complexity of a problem, the number of layers, including the number of neurons, may be increased. A simple neural network is shown in Figure 2.3 [19].

The algorithm's task is to detect patterns, so-called features. The features are characteristics that are used to distinguish data objects. Data objects are transformed into a feature-vector-based representation so that the algorithm can learn which feature correspond to the respective output. With weighted connections between the neurons, the dataflow is influenced. A sample dataflow is shown in Figure 2.3, including an indication of its weights. By applying a function, the incoming value is transformed and passed to all subsequent connected layers. This function is called the activation function [22]. The learning process's challenge is modifying the weights so that a data input leads to the expected output. There are different strategies for modifying weights. An efficient way to adjust the weights inside a neural network is supervised training. In supervised learning, a model is trained by a set of labeled data. One sample represents a mapping of input and their expected label outputs. Therefore, error and success can be calculated directly since the expected result is available. During the training phase, functions are derived, increasing or decreasing the weights accordingly for each passed sample. By repeating

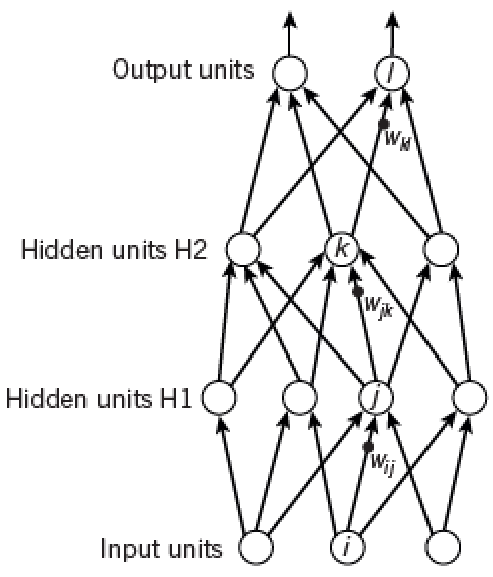


Figure 2.3: Simple neural network containing one input layer, two hidden layers and one output layer [19].

these adjustments with a large dataset of samples, the accuracy can be increased, and the error minimized [19]. The function which tracks the error of the model learning is called the loss function. The goal of the training is the minimization of that function to achieve a reasonable result [22].

2.2.3 Natural Language Processing

After investigating the basic machine-learning concepts focusing on deep-learning, we explore common strategies in natural language processing (NLP). By that, all kinds of use cases are included where languages should be processed by artificial intelligence. In our case, the model needs to learn the logic or language behind the smart contract's bytecode. It is required that the model can interpret the passed bytecode so that the classifications can be predicted accordingly. With that being said, we focus on the use case of text classifications since we expect that our model creates vulnerability classifications.

2.2.4 Text Representation

There are multiple ways of realizing a text classifier. Either way, the text needs to be first transformed into vectors with fixed-length. This transformation can be implemented in different ways. This section gives a short overview of some commonly-used techniques without going deeper into details. One technique is the so-called bag of words (BOW). In that case, a text's words are passed according to the number of occurrences [22]. Since the context is not represented in the vector, it might not be that suitable for all use cases. Another approach is using n-grams to represent text. A sequence of n words is stored in a vector. Therefore, the order of words inside a text is passed to the model learning algorithm. Since the dependency on the previous word is tracked, predictions of the next words in sentences can be created. Depending on the length of the n-gram, the context of words in a text is represented efficiently [22]. Especially in the context of deep learning, word embedding layers can be added to a neural network. These word embeddings train a mapping where words are placed into a high-dimensional space. By learning the semantic and syntactic information about words in the language, words with similar meanings are placed in similar vector regions. This layer can be pre-trained or trained on the fly [23, 22].

2.2.5 Recurrent Neural Network based Text Classification

In this section, we investigate Recurrent Neural Networks (RNN) suitable for classifying text based on a particular context. In most cases, these networks perform well for sequential inputs, such as languages. These models' special characteristic is storing information about previous runs and features, the so-called recurrent hidden state. This way, the textural structure, the word dependencies, and contexts can be interpreted by the model. As illustrated in Figure 2.4, the previous run's value influences the calculations made in the next run since it tracks previous data information. It shows that each neuron receives inputs from previous time steps. This way, sequences can be mapped together [19].

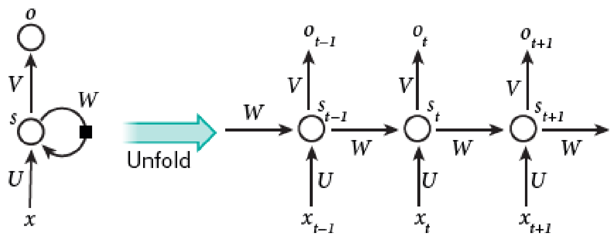


Figure 2.4: A recurrent neural network and the unfolding in time of the computation involved in its forward computation [19].

One of the most popular layers in that context is the Long Short-Term Memory (LSTM) layer, which does not simply store the previous state; however, it also captures long-term dependencies [19, 24, 25, 23, 26]. The lightweight version of an LSTM layer is the Gated Recurrent Unit (GRU) layer [26].

After introducing recurrent neural network layers used for deep-learning, we will now look at a whole model that can be used for text classification. Figure 2.5 shows a LSTM based model. In this case, even a bidirectional LSTM layer is added. This way, the structure is learned from both sides of a text. In this case, plenty of other hidden layers, such as Multilayer Perceptrons (MLP), are added as well. By increasing the number of layers, the performance of a model might be increased.

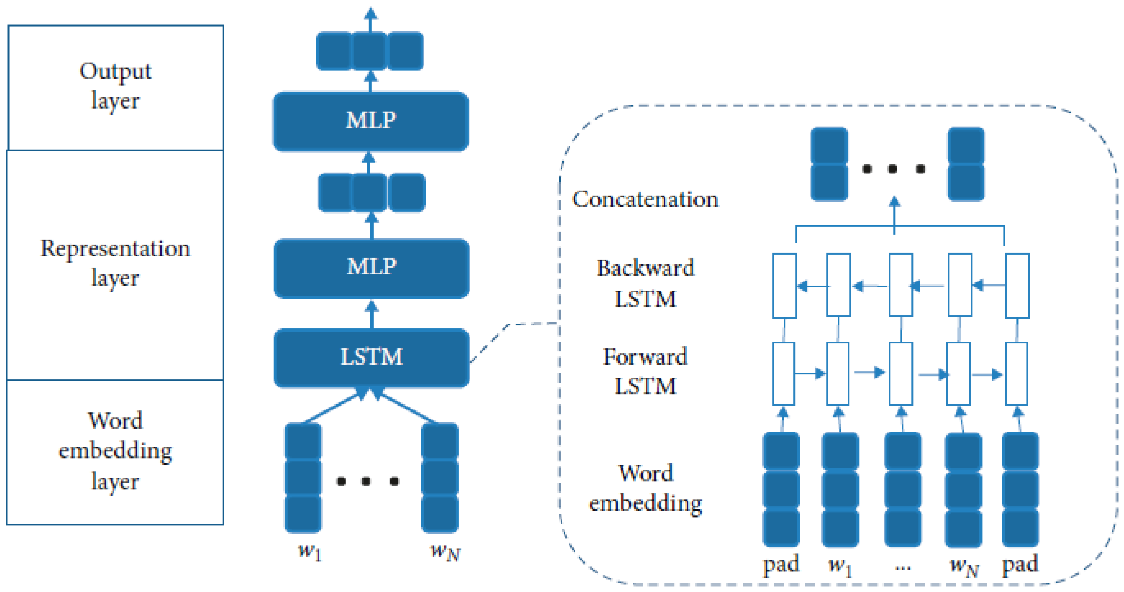


Figure 2.5: Model based on bidirectional LSTM [23].

3. Related Work

Given the tremendous economic impact of bugs in smart contracts and the impossibility to fix them after the deployment onto the blockchain, many smart contract pre-deployment scanning tools have been developed. In this section, strategies, how different tools approach the smart contract vulnerability detection problem are examined. Those can be divided into different categories. On the one hand, plenty of tools are static classifiers, other tools are dynamic classifiers, and on the other hand, machine-learning is used to classify smart contracts. Plenty of tools mix up certain approaches.

3.1 Static Vulnerability Detection Methods

Many tools perform analysis in a static, non-runtime environment. They examine the smart contract's source code or bytecode to detect vulnerabilities. Various tools create a custom model or structure to find weaknesses. In the following, static approaches are examined, which are used by several smart contract vulnerability detectors:

Slither:

Slither is a tool that uses taint analysis as a strategy to detect Solidity bugs dependent on user-controlled variables. In general, using taint analysis, the information flow can be tracked from the source to its sinks. For instance, all data derived by a taint source, such as user-controlled variables, may potentially change the program's behavior and is tainted. All the following values which are dependent on that data are tainted as well. Policies are applied to the program to introduce, propagate, and check the taint. When executing a statement, the taint analysis module checks if it is tainted or not and can raise an alert. An advantage of this approach is that the analysis can find nearly all vulnerabilities related to user input validation or other critical data flows. However, alerts may be too late. Overtainting, undertainting, and missing sanitization can also be a problem. Slither only accepts Solidity source-code, which will be used to create an internal representation [27, 28, 29, 30].

Oyente:

When using symbolic execution, the behavior of the program is represented by a built formula. Instead of actual input values, symbolic values determine if a program's particular path can be reached. Oyente works based on symbolic execution. In contrast to several other tools, it can detect vulnerabilities in both Solidity code and bytecode. It constructs a control flow graph of a contract, which is then used as a base to create input for the symbolic execution. With this approach, the environment is simulated in order to detect the vulnerabilities more efficiently. The main advantage of symbolic execution is that the program is examined for all execution paths instead of actual input values. This approach leads to good coverage in software testing. However, the performance depends on the number of explored paths and, thereby, the complexity of the contract [11, 31, 27, 28]. Since Oyente can work on the bytecode-level, it will be used for pre-labeling in this work. Further information will be given in Section 5.3.3.

Manticore:

Manticore is another symbolic execution tool which can work on bytecode. Symbolic values are passed to the bytecode tracking the discovered states. By verifying several invariants, a contract is analyzed. By running the symbolic execution in an emulated environment, the contract's state can be tracked as well as the Ethereum network state. With the analysis of the symbolic executed environment, certain vulnerabilities can be detected [32, 33].

Mythril:

Mythril combines multiple vulnerability detection approaches: symbolic execution, taint analysis, and Satisfiability Modulo Theories (SMT)solving. In SMT solving, the code of a smart contract is translated into SMT constraints for formal verification. These constraints are checked by performing queries to the SMT solver to ensure that the verification goals are satisfied. This way, bugs such as underflow or overflow, division by zero, unreachable code, and assertion fails can be detected. SMT solvers' advantage is that it can be used during the compilation time to give feedback about certain flaws before runtime. However, the generation of constraints can lead to complex formulas that need to be solved. Mythril can also work on both the bytecode-level as well as on Solidity source-code-level [34, 35].

Dedaub's Contract-Library:

The contract library by Dedaub [36, 37] provides multiple different features via online API. On the one hand, it collects the bytecode of smart contracts; however, on the other hand, it provides vulnerability classifications. These classifications are based on the bytecode of the blockchain since the source code is only available for 0.34% of the smart contracts. Bytecode analysis is performed on custom created decompilation techniques. Using that output, the tool, called MadMax, runs flow and loop analyses to detect gas-focused vulnerabilities. More detailed, the decompiler will return a specific schema. These schemas can be analyzed by applying certain rules which are defined to detect vulnerability types. All the resulting artifacts are published on the contract-library available for everyone [38, 39].

Securify:

Securify is a scalable, fully automated security analyzer for Ethereum smart contracts. It runs two detection methods. It firstly symbolically analyzes the dependency graph to receive semantic information. Afterward, several patterns are checked to identify whether a smart contract contains certain vulnerabilities. According to detected patterns, a smart contract is, therefore, classified as safe or can trigger violation patterns.This way, the smart contract is then classified [40, 41].

3.2 Dynamic Vulnerability Detection Methods

In contrast to previously presented strategies, we will have a look at smart contract vulnerability detection methods in the running state. For this purpose, the following approaches have been realized in a dynamic way to discover vulnerabilities:

MythX:

MythX [42] is built on top of Mythril and is a tool that combines the advantages of static code, symbolic, and taint analysis with code fuzzing to detect vulnerability. Fuzzing is used to execute the program providing invalid, unexpected, or random inputs. Afterward, states of a program can be produced, which might bring up some vulnerabilities. The efficiency might also be increased by considering inputs based on some information gathered from the smart contract. In general, fuzzing is simple and can often show up some unhandled exceptions, crashes, which may be overlooked by developers and tests. However, since this is a brute force approach, it comes with significant performance time and might have poor code coverage due to its dependency on the input [42, 28]. MythX, in general, is intended to support developers during their development. It provides a cloud-based online service that can be integrated into development lifecycles via API. Once a developer submits his source code, multiple analyzers start working in parallel, returning the classification result [43].

ReGuard:

Another fuzzing tool is ReGuard, which is specialized in the Reentrancy bug. It internally creates intermediate representations (IR), in detail, an abstract syntax tree (AST). Based on the generated IR, a C++ smart contract is constructed, keeping the original program execution behavior. Once the smart contract is fully transformed, a fuzzing engine creates a random byte input. By the analysis of the traces, reentrancy bugs are detected and reported [14].

ContractLarva:

Another approach, realized by ContractLarva, can be runtime verification where a violation of defined properties can lead to various handling strategies, such as a system stop. These properties can include undesired event traces of control- or data-flow. The main advantage of this strategy is that it is possible to prevent unexpected behavior at runtime. Therefore, bugs can survive various attacking attempts. A disadvantage is that the performance depends on the additional monitoring code's complexity, which also consumes gas. If properties are complex, it is more challenging to develop a proper monitoring [44, 45].

Maian:

The Maian tool is build based on the smart contract's bytecode. It executes the bytecode in custom EVMs with symbolic variables. By passing symbolic input to the contract, specific analyses detect the execution trace to identify related vulnerabilities, such as suicidal contracts. It combines the symbolic analysis and concrete validation. In concrete validation, the smart contract is executed on a fork of Ethereum. This way, the execution can be traced and validated. This way vulnerabilities are detected [46, 47].

3.3 Machine-Learning for Vulnerability Detection

A relatively new approach which is detecting vulnerabilities of smart contracts by utilizing the power of machine-learning is used by the following tools:

ContractWard:

The ContractWard tool utilizes the power of machine-learning to classify smart contracts into vulnerable and invulnerable. Therefore, the data set contains 49502 smart contracts, publicly available as source code. The model will be trained in a supervised way. For pre-labeling, the previously introduced Oyente tool will be used. It is assumed that those labels are reliable for their setup. Internally, the bytecode is transformed into bigrams, which will be used for feature extraction. A bigram is an n-gram where two words are stored together. Afterward, the bytecode will be substituted into logical groups, such as logical operations. This is performed to reduce the complexity and dimensions of input vectors. The oversampling technique is applied to overcome the imbalances of the dataset. In this step, samples for minority classes are created. Six binary classifiers are trained to realize the multi-label classification. In the end, the results of all classifiers are gathered to create the final output [48]. The following algorithms are employed to realize the machine-learning: eXtreme Gradient Boosting (XGBoost) [49], Adaptive Boosting (AdaBoost) [50], Random Forest (RF) [51], Support Vector Machine (SVM) [52], and k-Nearest Neighbor (KNN) [53].

LSTM approach:

Another machine-learning approach has been developed using LSTM neural networks to classify smart contracts. As data set, the previously introduced tool Maian is used for pre-labeling, detecting three different vulnerability classes. In contrast to the previous approach, a binary classifier is trained to only distinguish between vulnerable and invulnerable. In order to create a balanced dataset, the minority classes are oversampled, and the majority classes are undersampled. The data input is bytecode, which has been processed to receive the separate opcode. An embedding layer is used to create vector representations from the bytecode files. The length of the smart contracts is shrunk to 1600. The model is trained with two LSTM layers of 128 and 64 hidden layers [54].

Since we have seen all smart contract vulnerability scanner approaches, it is time to introduce this work's approach in the next section. We have decided to explore the deep-learning approach more deeply. Therefore, with this project, an AI Scanner is developed, which can distinguish eight vulnerability types of different three different tools.

4. Approach

In this section, the general approach of this work is examined, including several decision and assumptions which have been made in this process. As presented in Section 3, the work with machine-learning for smart contracts' vulnerability detection has already started. However, on the one hand, binary classifiers are used as expert models using classical machine-learning algorithms. And on the other hand, a deep-learning algorithm is developed with the scope of classifying smart contracts as vulnerable and invulnerable. With this work, it will be investigated if deep-learning can be utilized to classify smart contracts by its classification labels.

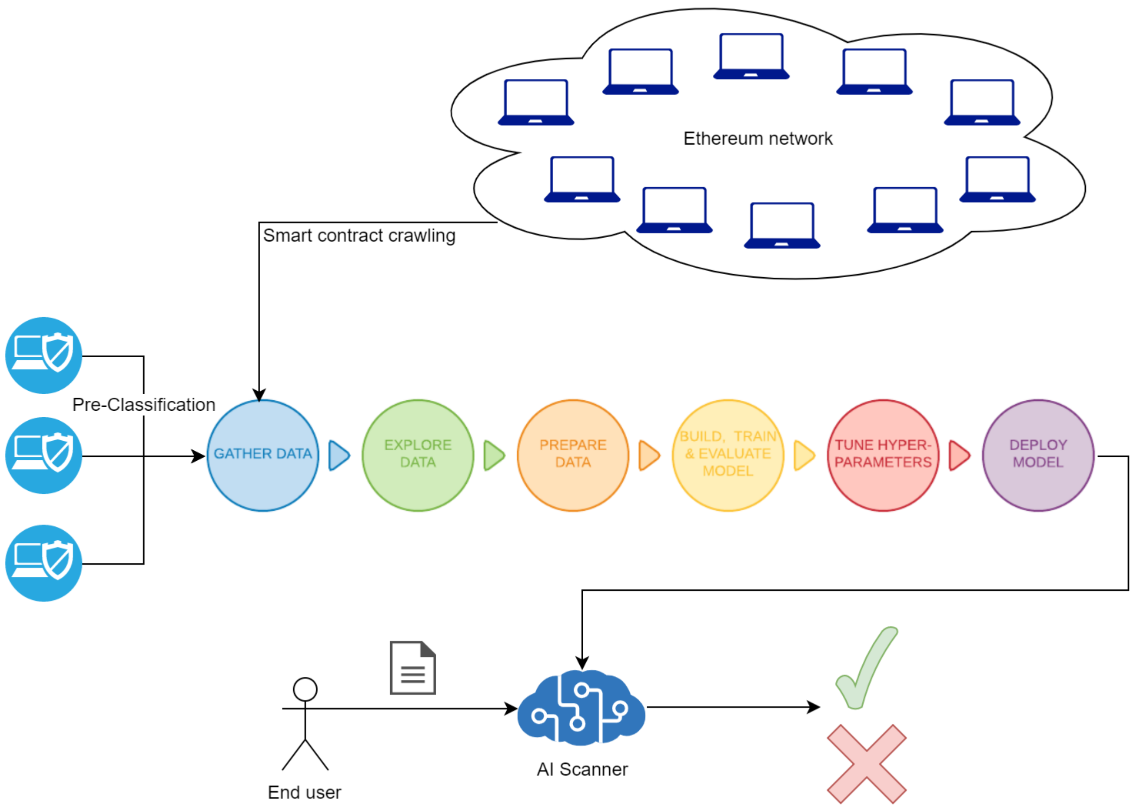


Figure 4.1: High-level overview of this work's approach [55].

The general overview can be obtained in Figure 4.1. In short terms, the smart contracts are downloaded from the Ethereum network. Using a pre-classification generated by several tools, the typical deep-learning approach will be followed to create the AI scanner. Further details are explained in the following sections.

4.1 Data collection

As already partly introduced in Section 2.2, various challenges need to be mastered to successfully create an efficient deep-learning model. Since we are at the beginning without having any data or pre-labeled classifications, this is the first challenge that needs to be tackled. Collecting data is crucial so that a deep-learning model can be trained. At this point, we decide to detect vulnerabilities of smart contracts based on bytecode. Since bytecode cannot be easily decompiled and only a limited amount of source code is available, the publicly available bytecode will be used. It might also make sense to detect vulnerabilities based on a low-level language. Since EVM bytecode is executed in a stack context, the control flow of a program might be more meaningful at this level. Furthermore, bytecode is available on all participating Ethereum nodes on the blockchain. Therefore, it is possible to crawl through all blocks searching for available bytecode. When crawling for the bytecode directly from the blockchain itself, it is also guaranteed that all samples will have the same format at the beginning. Given that the smart contract might be actively used over the years, common vulnerability types should be present on the blockchain. Since it is crucial to have enough data samples available for the learning process, it helps to have the full access to all deployed smart contracts. This way, a meaningful dataset can be constructed used as a base for the deep-learning. With this approach, it should also be simpler to create a balanced dataset for each class to further improve the expected performance. This smart contract crawling step can be either realized within this project or an existing tool that can be used and will be investigated in the implementation Section 5.

With the bytecode available several tools, introduced in Section 3, might be candidates to classify the retrieved smart contracts. Since all tools work differently, it is required that the tools can perform smart contract vulnerability scans based on the bytecode. The alternative can also be that classifier can process the smart contracts' address so that the required files are loaded internally. Based on the output, it needs to be decided whether enough representatives are available to perform deep-learning. Since enough smart contracts are available, it is initially not planned to artificially manipulate the dataset to create a classification for minority classes. In this step, multiple classification tools will be used to create a pre-labeling. This way, it is also tested if vulnerability types detected by different tools with different internal techniques can be mixed up. It also opens up the possibility that models based on this approach can be continuously extended. This might lead to various model changes; however, it is beneficial to mix up different vulnerability scans to create a more powerful and meaningful AI scanner.

When the pre-classification is finished, the retrieved data needs to be investigated whether additional adjustments need to be performed. First of all, the bytecode needs to be analyzed. Since the bytecode is a long hexadecimal number, it needs to be ensured that a deep-learning algorithm can interpret it. In typical text classification use cases, a sequence of words is passed to the algorithm. By training based on these sequences, the classifier can be trained. The sample length can also lead to a challenge. There are different strategies regarding this problem. All smart contracts that cross a certain threshold value will be ignored. This way, the deep-learning model would lose its attractiveness since these smart contracts are most likely not a candidate for training but also for predictions. Furthermore, with increased complexity, the probability of having program flaws increases. Alternatively, the smart contract can be chunked and then passed in parts to the algorithm.

Since the pre-labeling is based on the smart contract as a whole, it is not guaranteed that the vulnerability occurs in every chunk of that smart contract. In the end, we decided to chunk the dataset to relax memory usage during the deep-learning process. The detailed realization is described in the implementation Section 5.

In this step, during the data exploration phase, it needs to be decided on how many vulnerability types should be learned. This also includes the question of how many samples per class should be added into a dataset that will be passed to the deep-learning algorithm. When analyzing the number of representatives, a certain threshold should be defined, which is used to add or remove samples to the dataset. In the end, the target is to create balanced data to work on a straight-forward deep-learning model since this gives us the best chance to verify that the vulnerability classes can be learned by a deep-learning algorithm.

According to the described workflow, a lot of data needs to be processed. Figure 4.2 shows the expected dataflow, which will be realized for each smart contract. To sum it up, the bytecode is downloaded from the Ethereum network, pre-classified, and pre-processed. Thereafter, a dataset is constructed, which is directly passed to the deep-learning algorithm. Since a lot of data needs to be tracked, a database will be used for the realization of this project. In the end, pre-processed bytecode is passed as a data sample, including its labels. For each selected vulnerability type, a separate label is created. Therefore, we opted for a multilabel since one smart contract can be classified with more than one vulnerability. In the case of multiclass learning, each combination of the vulnerability types would need their own representation.

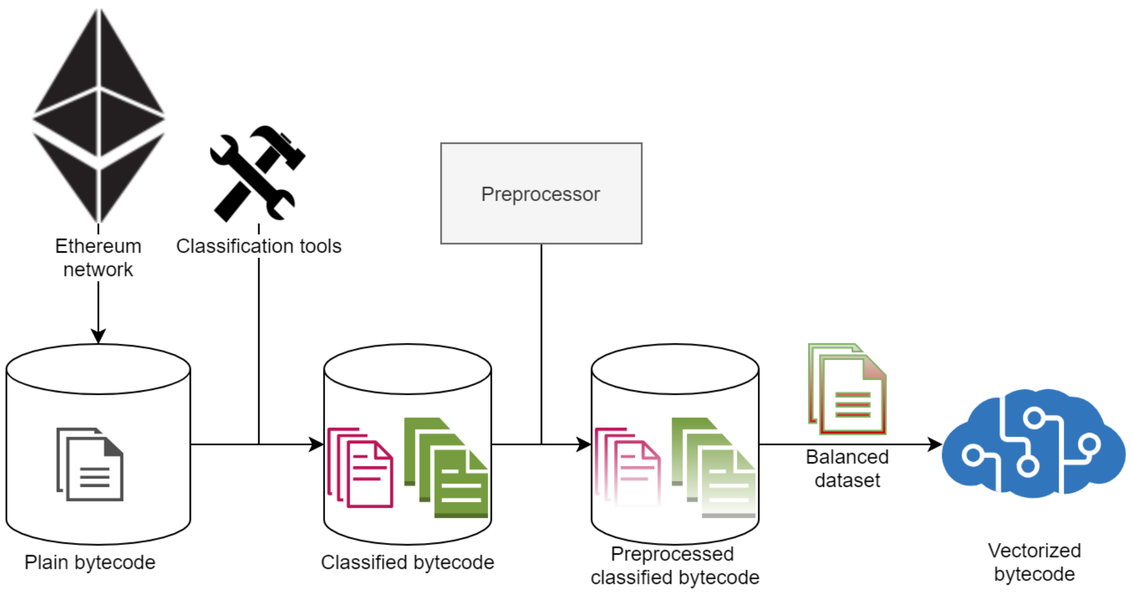


Figure 4.2: Expected dataflow of the smart contracts.

4.2 Deep-Learning

When the dataset is defined, the data needs to be prepared so that it can be directly passed to the deep-learning algorithm. An advantage of using the bytecode, the opcode vocabulary is very limited. When cutting off the opcode parameters, all invariants are removed from the bytecode. In this phase, we have consciously decided to work with embedding layers, given that the ContractWard tool uses n-grams. It is also the typical layer in text classification scenarios in the deep-learning context. Either way, the bytecode still needs to be parsed into a vector representation. Given that embeddings learn how to place words into the dimensions, numerical vectors are required. As indicated previously, the bytecode needs to be separated so that it can be interpreted as a text sequence of words. In this step, further simplification might also boost or limit the potential of efficient deep-learning classifiers. Either way, the resulting sequence of words needs to be tokenized so that the embedding layer can then process the smart contract. Since we chunk the dataset, the vector length can be relatively high in contrast to common test classification scenarios. In some cases, it even increases the efficiency of the input length is limited to a certain number.

The deep-learning process can only be triggered when a model architecture is defined. Since the performance of certain architectures needs to be tested, it makes sense to develop a small framework. The framework takes care of the correct training execution but provides a simple way of introducing a new kind of model architectures. This way, it is also preserved that the work on this approach can be continued easily. Besides the training execution, it also makes sense to manage all common parts of a deep-learning process, such as metric calculations or logging. It is also responsible for managing the incoming dataset. Therefore, the framework takes care that the dataset chunks are created and passed to the correct functions. Several chunks will act as the test set so that the performance of the trained model can be evaluated. Although working based on a framework taking care of plenty of stuff, a neural network still needs to be defined. We chose to go with typical recurrent neural network architectures commonly used in text classification. This also includes the selection of the activation and loss function.

After the model is built, trained, validated, and tested, the metrics will be analyzed so that further improvements can be implemented. Depending on the success, certain parameters can be adjusted to improve the overall results. One typical parameter is the number of layers in a model since it can lead to under- or overfitting. By increasing the architecture's complexity, it should be mentioned that the time for deep-learning increases heavily. Alternatively, the number of neurons for each layer can be modified. Since the bytecode has an arbitrary opcode length, the embedding dimensions may also be adjusted [55]. If all adjustments do not lead to success, the bytecode representation or the dataset in general needs to be changed.

In that phase, some different kinds of models will be trained and examined with different inputs. This way, the behavior on certain changes are analyzed. This information can be used for evaluation with other models created within this framework, however, also with the models developed by other tools.

Once an AI scanner is fully trained, it can be deployed for certain use cases. In the scope of this project, a small application is developed to showcase the intended workflow. An end-user passes a smart contract's bytecode file and therefore receives the probabilities if any vulnerability classes are detected in the smart contract. This way, a developer could scan his smart contract before deploying it onto the blockchain.

Figure 4.3 perfectly describes the workflow when developing a deep-learning model. The raw data is collected from certain providers. Afterward, the data is pre-processed until the data can be processed by the deep-learning algorithm. Several existing architectures and algorithms are tested until the best candidate has been found. This candidate can then be used to deploy it into a certain use case.

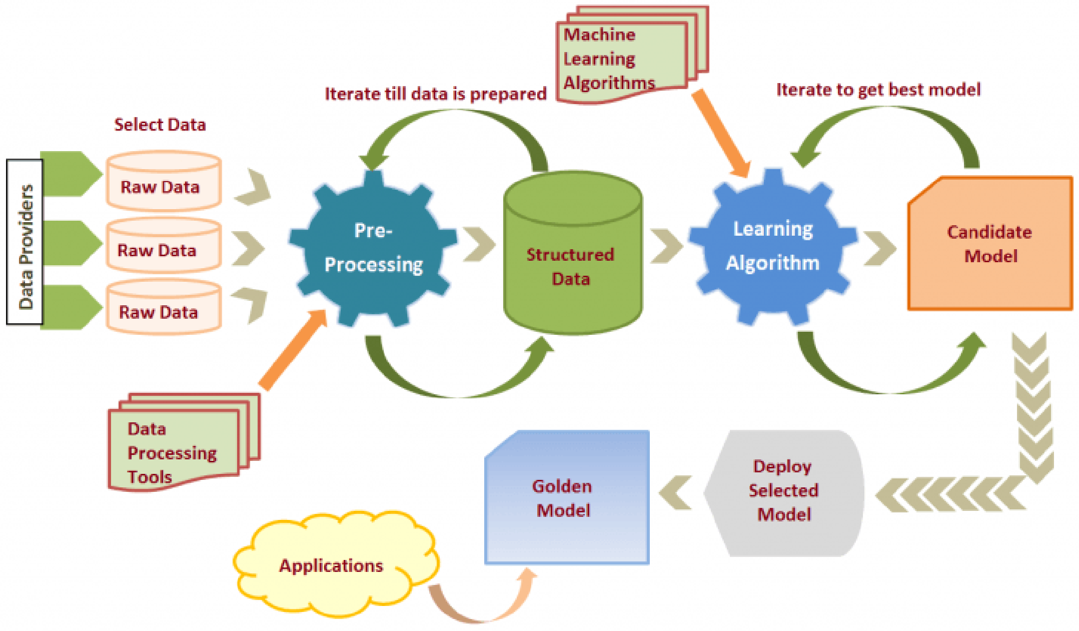


Figure 4.3: Workflow for the development of deep-learning models [56].

4.3 Evaluation

When the model developed with this approach is successfully created, we analyze if it can keep up with existing tools. In the worst-case scenario, a new model needs to be trained. Since the deep-learning approach is that reliant on meaningful data input, it can happen that several phases need to be repeated to achieve useful results.

Furthermore, this work opens up the door for new opportunities, which can be improved. Since this work focuses on the breakthrough of detecting vulnerability types based on neural networks, the model can be extended to support a greater variety of vulnerability types. In addition to that, the dataset can be maximized to find more representatives for minor label classes. Obviously, certain techniques, such as oversampling, can be applied to also train small vulnerability classes.

Furthermore, it can be investigated if smart contracts can not only be classified. Similar to image classifications, it might be possible to localize the critical parts of a smart contract so that it can be mitigated, even when detected on the bytecode level.

Since we now have a clear picture of this work's approach, the next section will dive deeper into the realization. It will explain the implementation, internal architecture, and components that are required to realize this project.

5. Implementation

Before going deeper into implementation details, the technologies applied to this solution are presented. This section will also show up the advantages of those technologies. Afterward, the realization of all phases defined in the previous section will be examined. These sections include applied methods, tools, architectural details, and implementation details.

5.1 Applied Technologies

In this section, some essential technologies will be introduced which have been applied in this project. It will be defined which programming language, data management tools, deep-learning framework, and packaging tools will be used in this work. The exact details will be presented in the corresponding component section. However, many components are developed with similar patterns sharing the same technologies.

5.1.1 Python

Since Python is commonly used for data processing, all modules developed in this project are written in Python. Tensorflow, one of the most prominent deep-learning Python frameworks developed by Google, will be used in this project. Therefore, we decided to keep all algorithms in Python. Supported by multiple available libraries, such as Pandas [57] and Numpy [58], data processing in Python can be easily realized. Python projects can also be simply placed into Docker containers to enable straightforward deployment and scalability.

Since the work packages to realize a deep-learning vulnerability scanner are often separated, we chose to develop the different functionalities in a modular fashion. Therefore, each module can be executed separately, sharing a defined amount of self-developed packages. Some of these packages are used in every algorithm, such as the logging package or the config validation package. Input is passed via CLI arguments. The package argparse [59] provides easy configuration options for CLI documentation. A sample architectural overview of newly created Python modules can be found in Figure 5.1. Modules, realizing a simple feature, contain a main file, a configuration validator, and a class implementing the core functionality. Except for the deep-learning module, nearly all modules are structured this way. For certain quality assurance, the pytest [60] library is utilized running tests on these modules. It is also possible to test the Docker image build and Docker container execution.

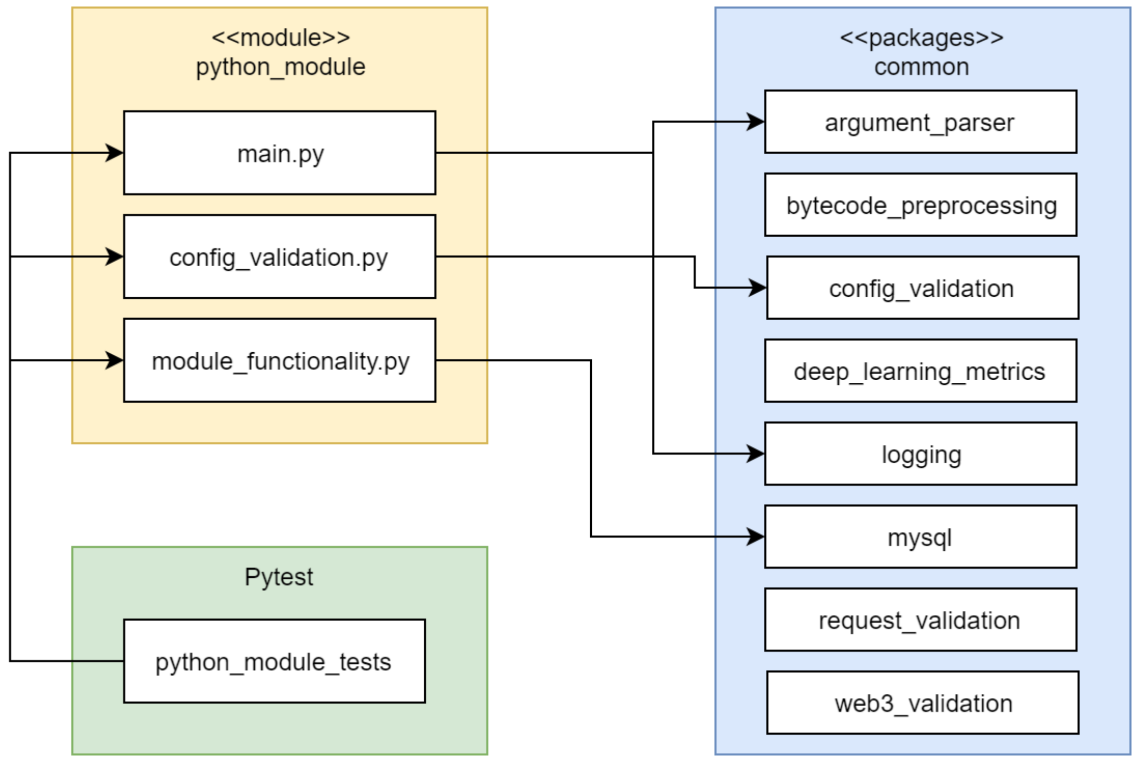


Figure 5.1: Architectural overview of developed modules in this project.

5.1.2 Docker

The open-source project Docker[61, 62, 63] delivers a solution to virtualize services in a simple fashion. Applications and its dependencies are wrapped up in isolated containers. Although containers remain isolated, they can be stacked and linked so that multiple containers work together. These containers are running instances of Docker images that can be built locally and uploaded to the cloud using a registry. Images are packages containing everything to run the software component, including the code, the runtime, libraries, environment variables, and configuration files. Since all application specifications are defined in a Docker image, it can be executed on all systems running the Docker engine. Virtualization is realized in a lightweight way, given that Docker containers share the host kernel so that no hardware needs to be virtualized. By scaling container replicas, it is possible to adapt to the demands of a service. This way, applications can be developed, deployed, and run independently of the system environment.

Since working with a massive amount of data, it is crucial creating ways to parallelize the execution of the developed applications. Because of its simplicity in software deployment and scalability, most developed components can be executed inside a Docker container. These containers can also be deployed on multiple machines to optimize and scale the data processing, which is necessary to realize the deep-learning approach.

5.1.3 MySQL Database

For the project scope, it is helpful to use a commonly known database that can be used to keep track of all data, which will be required for supervised deep-learning. It should be capable of storing large amounts of data with fast access times. In order to keep this approach scalable, it is crucial that millions of entries can be processed in a simple fashion. For this purpose, a MySQL [64] database will be used. Its querying options using the well-known SQL data language makes it a powerful tool for data processing. It is also possible to run the MySQL database inside a Docker container. On top of that, a phpMyAdmin [65] Docker container is deployed so that the database can optionally be managed via UI.

For our Python modules, a MySQL connector package is also available, which can be directly used to perform operations on the database. This connector will be utilized in this project.

5.1.4 Tensorflow

Tensorflow [66] is an open-source framework that can be utilized to create deep-learning models. Tensorflow utilizes the GPU to train deep-learning models. The name is derived by the main data representation in deep-learning, the tensor. These tensors can be interpreted as n-dimensional arrays. Calculations are realized with dataflow graphs where each node represents a certain mathematical operation.

Internally, Tensorflow is written in C++ [67]. It provides two APIs. Keras [68] can be used as a high-level front-end API, and for low-level adjustments, a C++ API is available as well. The Keras API comes with plenty of high-level features, which makes working with deep-learning algorithms straightforward. Besides that, TensorFlow provides a suite of properties and tools to make its usage very attractive [69].

Because of its straightforward approach with a low barrier of developing deep-learning models, Tensorflow will be used in this project. It makes it easy to define network architectures so that it can be perfectly utilized as a playground to create an AI scanner that should be capable of classifying smart contracts by its bytecode.

5.2 Smart Contract Bytecode Aquisition

Before we come to the challenge of creating deep-learning models to classify smart contracts, a data set of smart contracts' bytecode is required. In this section, all required components will be examined, necessary to receive information about the smart contracts from the blockchain, and to download them finally. In addition to that, some information about smart contracts will be transferred into a MySQL[64] database. The management of the smart contracts in a database will simplify tasks, which are explained in the next sections. For the realization of the described steps, a composition of multiple components is required. These components will be introduced in the next section of the architectural overview. After that, we will dive deep into the component details.

5.2.1 Architectural Overview

To receive the smart contracts' addresses, we use the open-source tool Ethereum ETL [13, 70]. The tool connects to the Ethereum network and exports blockchain content into CSV files. The content of these CSV files is used to extract relevant information, especially the contract addresses, into a MySQL database. There are multiple Web APIs available that can be used to download the bytecode by its addresses. The addresses on the blockchain have been received in the previous steps. For this project, we use a combination of the APIs, provided by Infura[71] and Dedaub's Contract-Libray[72], to download and store the bytecode when available. The relationship between each component is illustrated in Figure 5.2.

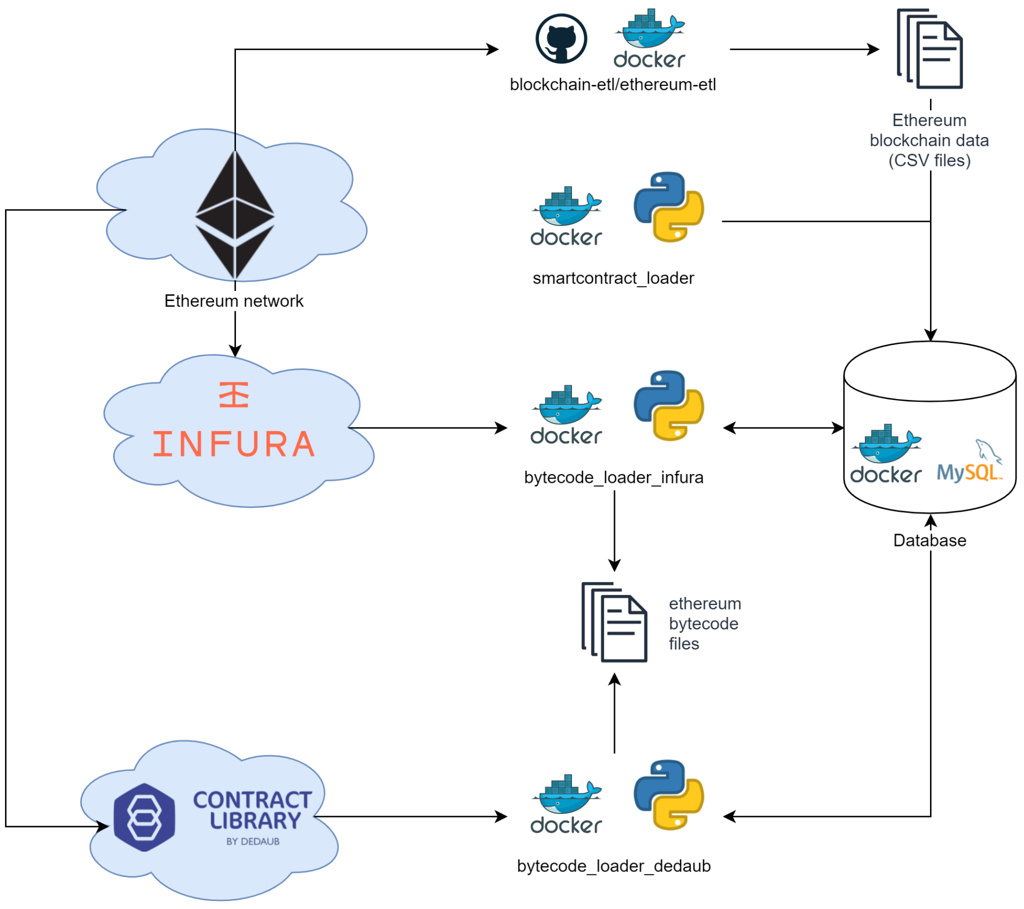


Figure 5.2: Architectural overview of components which deliver the smart contract bytecodes in the end.

As introduced in the previous chapter, all components are executed inside Docker containers to parallelize the data-intensive tasks. The additional self-developed components are all written in Python. The component *smartcontract_loader* is developed to extract the information about smart contracts into the MySQL database. As explained before, this information is available in CSV files generated by the Ethereum ETL tools. With the addresses available in the database, the Python modules *bytecode_loader_infura* and *bytecode_loader_dedaub* can request the bytecodes from the APIs. Therefore, all tasks regarding communication with the APIs, file management, and database synchronization are implemented in this part. The exact results and processes are described in the upcoming sections.

5.2.2 Ethereum ETL

Before vulnerability analysis can get started, the smart contract bytecode files must be retrieved from the Ethereum blockchain. The tool Ethereum ETL provides this exact possibility to extract blocks and transactions from the blockchain into CSV files. These CSV files contain all the necessary information actually to download the bytecode files. The tool can be executed inside a Docker container so that it can be started right away.

```
1  version: "3.7"
2
3  services:
4    ethereum-etl:
5      build:
6        context: .
7        dockerfile: Dockerfile
8      image: ethereum-etl:latest
9      container_name: ethereum-etl
10     command: export_all -s 0 -e 4999999 -b 100000 -p https://
          ↪ mainnet.infura.io
11     volumes:
12       - $HOME/ethereum-data:/ethereum-etl/output
```

Figure 5.3: Docker-Compose File to run Ethereum ETL algorithm.

As displayed in Figure 5.3 in line 10, the command is configured for each container. The options used in this command are:

- **-s** to set the start block number
- **-e** to configure the end block number
- **-b** to setup the batch size (the number of blocks to export at a time)
- **-p** to choose the URI of the provider which will be used

By setting different number ranges, the boundaries of blockchain blocks are defined, which will be included in the discovery. The CSV output files need to be mounted to the host system for further usage. For the scope of this project, the first 5 million blocks of the blockchain will be discovered and used to find smart contracts. Internally, the Ethereum ETL uses the Ethereum JSON RPC API to communicate with the Ethereum network and transfers the data into the CSV files. For this project, Ethereum's mainnet is used. For the smart contract retrieval, it utilizes the following endpoints of the API in this process:

- **eth_getBlockByNumber**: This request returns information about a block on the blockchain by its number. The response then includes information about the block, such as hash, parent hash, nonce, and, most importantly, an array of transaction hashes. These hashes will then be used as input for the next API call.
- **eth_getTransactionReceipt**: As the name indicates, the transaction receipt will be loaded in this step. These receipts provide information about the result of a transaction, such as status, used gas, logs, and the contract address. This contract address is the most important information retrieved by this tool for this project's further progress.

To sum things up, the Ethereum ETL communicates with an Ethereum node via the JSON RPC interface to get the transactions of a specified block range. The hashes of the transaction will be used to find out the contract address. All information gathered in this process is then stored in CSV files, which act as the first base for the next step. In the end, the following resources are stored in CSV files:

- blocks
- transactions
- token-transfers
- receipts
- logs
- contracts
- tokens
- traces

Since we are only interested in the contract address, we focus on those files. Figure 5.1 demonstrates the information gathered by Ethereum ETL. In the first column, the address for the smart contract is stored. Afterward, the bytecode of some contracts is available. As we can see, the first entries seem to be incomplete. Therefore, we decide to download the bytecode files on our own, only based on all addresses available in the retrieved CSV files. The other columns contain all function signature hashes and information about some interface conformity, such as ERC20 [73].

address	**bytecode**	**function_sighashes**	**is_erc20**	**is_erc721**	**block_number**
0xfef1f3...	0x6060...	0x0dbe671f	False	False	
0x25df0f...	0x6060...	0x28d3ad3f...	False	False	
0x23bf9a...	0x		False	False	
0xf9530b...	0x		False	False	
0xa1efc5...	0x		False	False	
0x034faa...	0x		False	False	
0xcf3487...	0x		False	False	
0x313fe4...	0x		False	False	
0xf013ad...	0x6060...	0x04289bec...	False	False	

Table 5.1: Part of the Ethereum-ETL contract.csv file.

Before we continue with the actual download process, all addresses are loaded from the gathered CSV files into a MySQL database, shortly described in the next section.

5.2.3 Smart Contract Loader

This section introduces the first of many Python modules required to create a deep-learning model. The task of this module is to load all addresses of smart contracts into a MySQL database. By this operation, the process of managing the data is initiated. The Ethereum ETL tool's output, presented in the previous chapter, serves as input for this module. When running this script, it is assumed that a database is set up and running. The process is triggered by executing the module with arguments about the MySQL database, such as endpoint and credentials, and the local path to the CSV files retrieved by the Ethereum ETL tool. Internally, the program can be separated into three main phases:

- **Input validation**:: First of all, all arguments passed to the algorithm are verified before the actual code execution. This verification includes checks if the specified directory containing the CSV files is present and if a connection to the MySQL database can be established. If the program is not appropriately configured, the execution stops immediately informing the user.

- **Initial database setup**: Before loading the files into the database, the initial database is created, containing the following columns: id, address, and bytecode_path. For each contract, an identifier is generated as well as a bytecode_path file name. By this, contract tracking is simplified and managed. The id will be automatically generated when data is inserted. It will also generate leading zeros, which will simplify the file name generation. The bytecode attribute is set to UNIQUE to prevent the duplication of entries. The following columns are also added for future modules: op_length, self-destruct, download_status, dedaub_classified, oyente_classified, mythril_classified. The meaning of these attributes will be examined in upcoming chapters. By default, all these entries are initiated with NULL. The schema of this database is illustrated in Table 5.2.

Column name	**Column type**	**Additional constraints**
id (PRIMARY)	int(7)	NOT NULL AUTO_INCREMENT
address	varchar(100)	UNIQUE
bytecode_path	varchar(120)	DEFAULT NULL
self_destruct	tinyint(1)	DEFAULT NULL
download_status	tinyint(1)	DEFAULT NULL
dedaub_classified	tinyint(1)	DEFAULT NULL
oyente_classified	tinyint(1)	DEFAULT NULL
mythril_classified	tinyint(1)	DEFAULT NULL

Table 5.2: Schema of the database created by module *smartcontract_loader*.

All the above steps are realized with a shared python package, which serves as an interface between a python module and the actual database. It acts as a connector so that any Python module can perform Create, Read, Update, and Delete (CRUD) operations on the database.

- **Data transfer of the addresses into the database**: When all preparation has been finished, the actual import process is started. The program goes through all CSV files available in the specified directory filtering out all addresses. It internally reads in addresses with a batch size of 1000 and inserts them into the database.

id	address	bytecode_path	op_length	download_status	...
0000001	0x94c81a...	NULL	NULL	NULL	...
0000002	0x61c5e2...	NULL	NULL	NULL	...
0000003	0x432a20...	NULL	NULL	NULL	...
0000004	0xa50156...	NULL	NULL	NULL	...
0000005	0xe28e72...	NULL	NULL	NULL	...
0000006	0x412c79...	NULL	NULL	NULL	...
0000007	0xf66759...	NULL	NULL	NULL	...
0000008	0x137706...	NULL	NULL	NULL	...
0000009	0xf4a44a...	NULL	NULL	NULL	...
0000010	0xcdd1f5...	NULL	NULL	NULL	...

Table 5.3: State of the database after executing the *smartcontract_loader* module.

After importing the smart contracts' addresses of the first 5 million blocks of the blockchain into the database, the database contains 1.208.173 entries. Each entry represents a smart contract detected by the Ethereum ETL tool. Table 5.3 shows the state of the database after executing this module, as explained before. It is also possible to run this module inside a Docker container, which simplifies the deployment process and enables parallelism.

5.2.4 Bytecode Loader Infura

After retrieving the smart contracts' addresses from the blockchain, the bytecode needs to be downloaded. The bytecode serves as a base for the pre-classification and the input for the deep-learning algorithm. As described in the previous section, we have gathered 1.208.173 addresses. The module presented in this module utilizes the API provided by Infura to download the bytecode. Infura provides a simple way to connect to the Ethereum network without setting up an own Ethereum node. Infura provides an API on top of the JSON RPC API, which is used to communicate with their Ethereum clients. For this purpose, a free account needs to be created to generate a project-id used for all download requests.

The Infura URL containing the project id needs to be passed to the algorithm, downloading the bytecode. Besides that URL, the target path where the files should be stored and the MySQL access information needs to be configured. It is recommended to parallelize the download processes to increase the speed of this process. By specifying an id-range, the module looks up the ids from the database and downloads the corresponding bytecodes from the Ethereum network. Since this module can be executed inside a Docker container, it can even be executed on multiple machines. After the initial input validation, including the database, Infura endpoint, and target directory check, the bytecode paths will be defined. These bytecode paths are generated by concatenating the id and the address of the smart contract entry in the database.

If all input prerequisites are fulfilled, the main process is executed. Inside of the specified id range, the algorithm processes batches of size 1000. Therefore, it loads the gathered addresses from the database, which have not been downloaded yet. The received addresses are passed to a web3 client [74]. This client can be used for connecting to Http and Https based JSON-RPC servers, which is, in our case, the Infura servers. It provides functionality specialized for Ethereum interaction. For the bytecode downloading the *web3.eth.getCode* functionality will be used. When the address of a smart contract is passed to this function, the bytecode will be downloaded. This bytecode will then be stored in the files defined in the database. For keeping track of downloaded files, the *download_status* in the database is updated accordingly.

id	**address**	**bytecode_path**	**download_status**	**self_destruct**	**...**
0000001	0x94c...	0000001_0x94c....bin	1	0	...
0000002	0x61c...	0000002_0x61c....bin	1	0	...
0000003	0x432...	0000003_0x432....bin	1	0	...
0000004	0xa50...	0000004_0xa50....bin	1	0	...
0000005	0xe28...	0000005_0xe28....bin	0	1	...
0000006	0x412...	0000006_0x412....bin	1	0	...
0000007	0xf66...	0000007_0xf66....bin	1	0	...
0000008	0x137...	0000008_0x137....bin	1	0	...
0000009	0xf4a...	0000009_0xf4a....bin	1	0	...
0000010	0xcdd...	0000010_0xcdd....bin	0	1	...
0000011	0x2bb...	0000011_0x2bb....bin	1	0	...

Table 5.4: State of the database after executing the *bytecode_loader_infura* module.

After running the *bytecode_loader_infura* module, 1.155.085 bytecodes have been downloaded. This means that for almost 96% of the smart contracts, the bytecode is successfully loaded. As indicated in Section 5.2.2, in many cases, an empty bytecode 0x is downloaded. There are some scenarios why this might happen. It can be the case that the Ethereum node is not fully synced with the network so that the bytecode is not available. Alternatively, an empty contract might be deployed. Another reason might also be that the smart contract is self-destructed. These smart contracts are flagged by updating the *self_destruct* and the *download_status* column in the database. Table 5.4, containing a snippet of the database, illustrates the result according to the described behavior.

Another bytecode acquisition method will be introduced in the next section, which will partly solve the unavailable bytecode situation.

5.2.5 Bytecode Loader Dedaub

As discussed in the previous section, plenty of smart contract bytecode could not be downloaded. With the module introduced in this chapter, the gaps shall be filled. Therefore, we utilize the API provided by Dedaub, the Ethereum Contract Library. In addition to bytecode, this API provides multiple other smart contracts' information. For instance, disassembled or even decompiled smart contracts are available. Furthermore, as we will see in Section 5.3.2, vulnerability warnings are added to the response body. An example request can look as follows in Figure 5.4. The only relevant data is bytecode, which will be loaded by the module *bytecode_loader_dedaub*.

Similar to the previous module, a so-called package-range can be passed to the algorithm. This package-range represents the number of smart contracts where the bytecode could be downloaded with the *bytecode_loader_infura*. By configuring a specific range, this process can be parallelized as well to increase the download speed. In this case, the parallelization is a little limited since the API limits the number of requests. Since this process's outcome should be synced with the database, the connection information needs to be provided to this Python module. Moreover, the target storage location and the API endpoint can be configured.

After the initial input validation, including the database and target directory check, the main process will be triggered. With a batch size of 1000, the leftover addresses from the *smart_contract_loader_infura* module will be queried. Therefore, the flags *self_destruct* and the *download_status* of the database, as shown in Figure 5.4, will be used. All entries with *download_status* value *0* and the *self_destruct* value *1* are candidates to be processed

```
{
  "network": "Ethereum",
  "address": "0x94c81a1dbc5c41a5e9962a2d6da5aa5ff684259f",
  "block_number": 86800,
  "date_scanned": "2019-03-20T00:00:00+0000",
  "ether": 0,
  "has_source": false,
  "has_bytecode": true,
  "has_disassembled": true,
  "has_decompiled": true,
  "has_warning": false,
  "source": null,
  "bytecode": "60606040526000357c0100000000000...",
  "disassembled": "...",
  "decompiled": "...",
  "json_abi": "[...]",
  "warnings": {},
  "bytecode_md5": "3a46ee2fec930058d63f5ea3bf7b1392"
}
```

Figure 5.4: Sample response body of a Dedaub Contract-library API request.

by this module. These queried addresses will be used as input for the API provided by Dedaub to retrieve the bytecode. This bytecode will be stored at the specified target location according to the name convention defined in the database. If an error occurs or the bytecode is not available, the addresses are skipped. Otherwise, the *download_status* flag of the database is updated to the value *1*. After running this module, the process terminates with a state shown in Figure 5.5.

id	**address**	**bytecode_path**	**download_status**	**self_destruct**	**...**
0000001	0x94c...	0000001_0x94c....bin	1	0	...
0000002	0x61c...	0000002_0x61c....bin	1	0	...
0000010	0xcdd...	0000010_0xcdd....bin	1	1	...
0000018	0x128...	0000018_0x128....bin	1	1	...
0000029	0xa1a...	0000029_0xa1a....bin	0	1	...
0000033	0x581...	0000033_0x581....bin	1	1	...
0000034	0x08f...	0000034_0x08f....bin	1	0	...
0000036	0x86c...	0000036_0x86c....bin	1	1	...
0000041	0x7ae...	0000041_0x7ae....bin	1	0	...

Table 5.5: State of the database after executing the *bytecode_loader_dedaub* module.

In the end, only an additional 1.526 bytecode file could be retrieved. Since the number of smart contracts is sufficient for this project, the leftover 51.562 addresses of the first 5 million blocks will be ignored. In total, 1.156.611 smart contract bytecode files will be available for the upcoming vulnerability analysis. They will be firstly required as an input for existing smart contract classifiers for pre-labeling. With this pre-labeling, patterns in bytecode files can be learned by a deep-learning model. These steps will be examined in the following chapters.

5.3 Pre-Classification of Smart Contracts

Before we start working on this project's deep-learning part, a pre-classification of the samples is required for supervised-learning. Since we have the smart contracts available in the form of bytecode, only tools that work based on bytecode will be used for this project. In this case, three tools will be used for the pre-labeling. To achieve the first decent results, a reasonable size of representatives needs to be gathered, which can be used for the deep-learning. The developed deep-learning model will unite the results of multiple tools and provide a more meaningful output for end-users. This section will present how smart contracts have been pre-classified and loaded into the database for further analysis.

5.3.1 Architectural Overview

For this work's scope, we have decided to gather the results created by the tools Oyente, Mythril, and Dedaub's Contract Library. These tools have been selected since they provide analysis based on bytecode or the smart contract addresses. Since decompiled bytecode may lose plenty of information, tools that scan smart contracts based on the source code level are not that feasible. In general, we need to differentiate between the API of Dedaub and the other two tools. In contrast to the API where vulnerability scans have been already executed, Oyente and Mythril classifications need to be executed by ourselves.

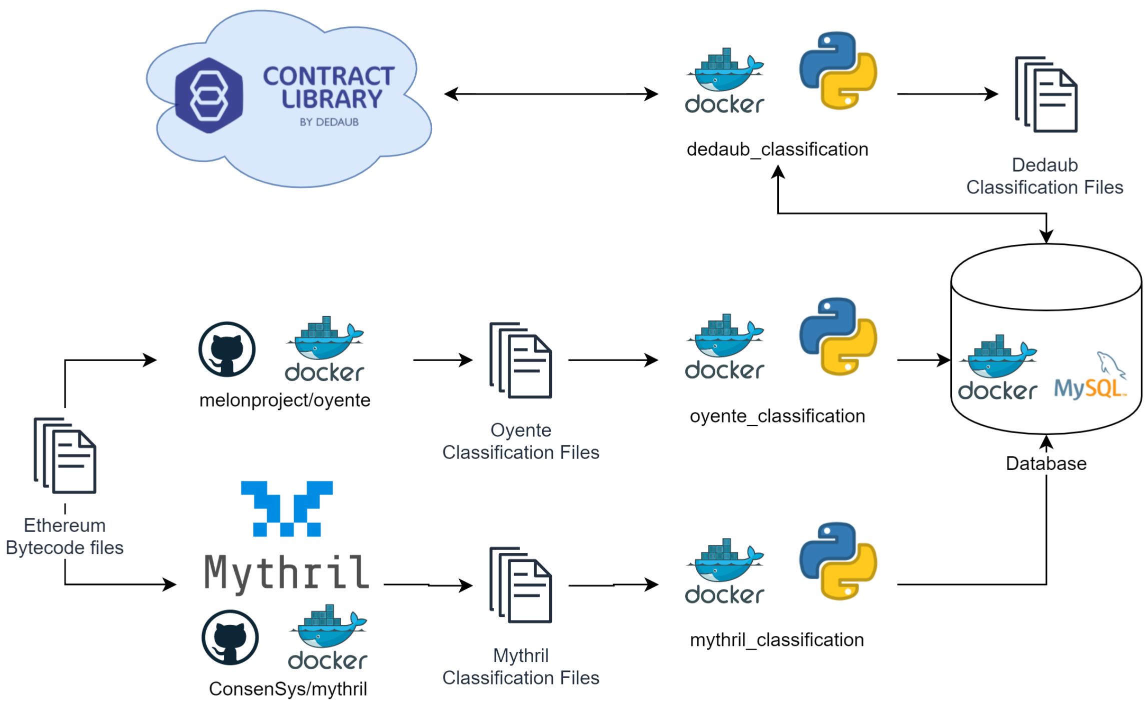


Figure 5.5: Architectural overview of components which pre-classify smart contracts.

As illustrated in Figure 5.5, a workflow is defined to achieve the overall goal of gathering vulnerability classifications into the database. As indicated in Section 5.2.5, the vulnerability types can be directly retrieved from the API. Therefore, a new Python module takes care of loading the classification types into the database. For the other two tools, a Docker container is provided so that the gathered bytecode only needs to be mounted into the container to be processed. In both cases, classification files will be generated. The content of these files will then be loaded into the database using new Python modules.

In the end, we will have an overview of the number of representatives of a certain number of vulnerability classes. Since there are no dependencies between the classification tools, the set of vulnerability types can be easily extended with other available tools.

In the following sections, the detailed process of how the pre-labeling of smart contracts has been realized is examined. Thereby, the focus lies on the usage of these tools and not on its internal implementation. Further details have been examined in Section 3.

5.3.2 Dedaub Smart Contract Classification

Since the API usage has been presented for module *bytecode_loader_dedaub* of Section 5.2.5, there is no need for further introduction. As illustrated in the sample response of Figure 5.6, the vulnerability types are available. For persistency purposes, these classifications will also be stored in files as well as in the database. Similar to the previous modules, an id-range can be configured for the classification loading process. The ids are defined inside of the database for the smart contract management. For each database entry, the address will be used to generate the API call. This way, the classifications can be loaded. By configuring an id-range, this process can be parallelized as well to increase the download speed. Although, in this case, the parallelization is a little limited since the API limits the number of requests.

```
{
  "network": "Ethereum",
  "address": "0xa50156cf80fa9ec2e16899e4fb7e072300787417",
  "block_number": 88069,
  "date_scanned": "2019-03-20T00:00:00+0000",
  "..."
  "has_bytecode": true,
  "..."
  "has_warning": true,
  "source": null,
  "bytecode": "60606040523615610061576 0e060020a6000...",
  "..."
  "warnings": {
    "Accessible selfdestruct": "{\"functions\": [\"0
       ↪ x960edffb(uint256 varg0, uint256 varg1, uint256
       ↪ varg2)\"], \"description\": [\"Selfdestruct at 0
       ↪ x960edffb(uint256 varg0, uint256 varg1, uint256
       ↪ varg2) potentially accessible\\n\"]}",
    "DoS (Unbounded Operation)": "{\"functions\": [\"
       ↪ joinGame()\"], \"description\": [\"Array iterator
       ↪  at joinGame() may be susceptible to DoS by
       ↪ increasing storage requirements at joinGame()\\n
       ↪ \"]}",
    "Tainted selfdestruct": "{\"functions\": [\"0x960edffb
       ↪ (uint256 varg0, uint256 varg1, uint256 varg2)\"],
       ↪  \"description\": [\"Smart contract user could
       ↪ potentially override safedestruct address at 0
       ↪ x960edffb(uint256 varg0, uint256 varg1, uint256
       ↪ varg2)\\n\"]}"
  },
  "bytecode_md5": "c0bab5bc1994e3aa4035584976c1dc37"
}
```

Figure 5.6: Sample response body of a Dedaub Contract-library API request.

The newly created Python module requires information about the database endpoint as well as the target storage path. After the input validation, a new table is created according to the schema illustrated in Table 5.6. It contains the id and address of the smart contract. It also stores the name of the created classification files. For each vulnerability class provided by the API, a column is created. The vulnerability class's value is set to value *1* when the API detects a flaw of that vulnerability type. Otherwise, the value is set to value *0*.

Column name	**Column type**	**Additional constraints**
id (PRIMARY)	int(7)	NOT NULL UNIQUE
address	varchar(100)	UNIQUE
classification_path	varchar(120)	DEFAULT NULL
Accessible selfdestruct	tinyint(1)	DEFAULT NULL
Bad Randomness	tinyint(1)	DEFAULT NULL
DoS (Induction Variable Overflow)	tinyint(1)	DEFAULT NULL
DoS (Unbounded Operation)	tinyint(1)	DEFAULT NULL
DoS (Wallet Griefing)	tinyint(1)	DEFAULT NULL
ERC20 Underflow	tinyint(1)	DEFAULT NULL
FALLBACK_MAY_FAIL	tinyint(1)	DEFAULT NULL
FALLBACK_WILL_FAIL	tinyint(1)	DEFAULT NULL
FALLBACK_WILL_FAIL (cheap LOG)	tinyint(1)	DEFAULT NULL
Reentrancy	tinyint(1)	DEFAULT NULL
Reentrancy (Constantinople)	tinyint(1)	DEFAULT NULL
Reentrancy (low confidence)	tinyint(1)	DEFAULT NULL
Reentrancy (Low Severity)	tinyint(1)	DEFAULT NULL
Tainted delegatecall	tinyint(1)	DEFAULT NULL
Tainted Ether Value	tinyint(1)	DEFAULT NULL
Tainted Owner Variable	tinyint(1)	DEFAULT NULL
Tainted selfdestruct	tinyint(1)	DEFAULT NULL
Tainted Storage Index	tinyint(1)	DEFAULT NULL
TwinCalls	tinyint(1)	DEFAULT NULL
Unchecked Tainted staticcall	tinyint(1)	DEFAULT NULL

Table 5.6: Schema of the database created by module *dedaub_classification*.

When the database is set up, the algorithm starts working on the defined id-range. The addresses, which are in the defined id-range and the bytecode is available, will be queried in batches of size 1000. The bytecode must be available since it should be the base for the deep-learning model. An API request will be triggered using the address as a parameter. When the classification has been successfully loaded, a file will be created to persist the acquired information. It creates a JSON file storing the warnings retained by the API. The file which persists the warnings object of the above example request is illustrated in Figure 5.6.

In addition to that, the classifications are gathered in the database according to the previously described behavior. Finally, the flag of the primary smart contract table with Schema 5.2 introduced in Section 5.2.3 is updated. When an error occurred, the *dedaub_classified* flag is set to value *0*. On success, the value of the flag will be updated to the value *1*.

In the end, 1.156.610 smart contracts have been pre-classified. 48.578 of these smart contracts have been labeled with at least one vulnerability type. In Figure 5.7, the distribution of the vulnerability classes can be examined. As we can see, *tainted_selfdestruct*, *dos_unbounded_operation*, and *accessible_selfdestruct* are the dominating vulnerability classes. These are reasonable candidates for the deep-learning algorithm.

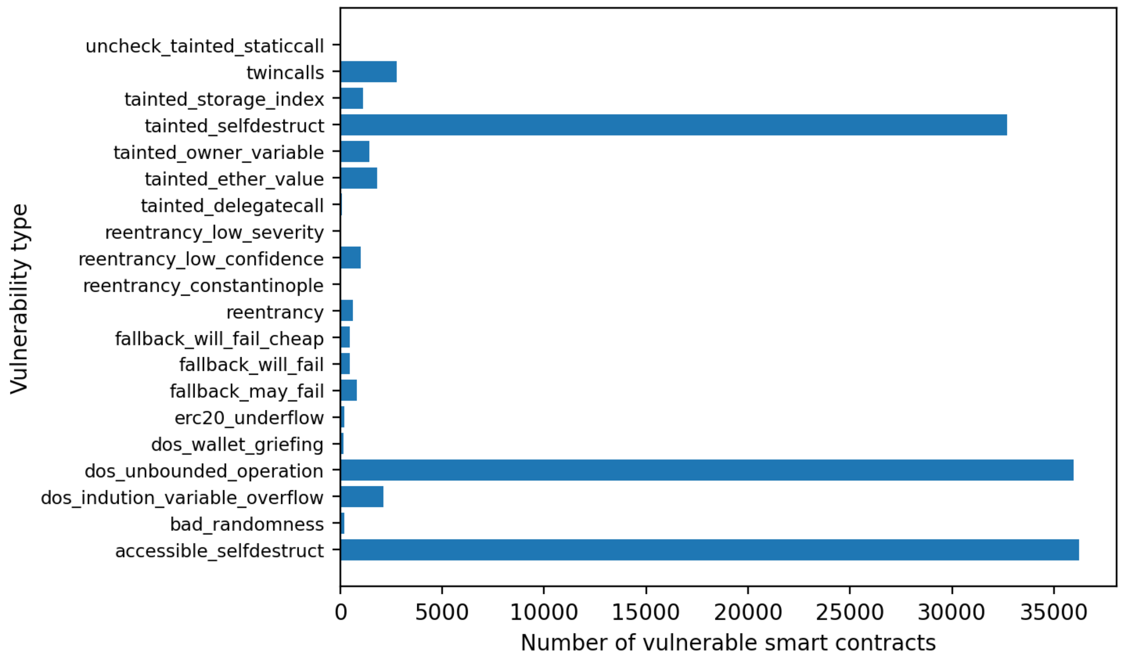


Figure 5.7: Distribution of the Dedaub contract-library's classification labels.

5.3.3 Oyente Smart Contract Classification

In contrast to the Dedaub API approach, the Oyente vulnerability scan has to be executed by ourselves. Therefore, the pre-labeling process is divided into two separate steps. Firstly, Oyente scanning is triggered, which produces classification output files. Secondly, a newly created Python module loads the information stored in the files into the database. This way, the Oyente classification for each available smart contract can be gathered to be used for the deep-learning process.

```
#!/bin/bash

# For each bytecode file mounted into the container
for bytecode in /oyente/oyente/bytecodes/*/*.bin;
do
  # Skip already classified smart contracts
  if [ -f "$bytecode.json" ]; then
    echo "$bytecode␣already␣classified."
  # Run the Oyente classification
  else
    python oyente.py --source "$bytecode" --bytecode --json
  fi
done;
```

Figure 5.8: Script file to trigger Oyente vulnerability scan of mounted bytecode files.

Since Oyente can be deployed via a Docker container, this process can be parallelized to increase the runtime execution speed. Inside of the Docker container, the Python project is integrated. Since it is designed to run the vulnerability scan for one smart contract, a short bash script must be developed to perform a mass-scan-operation. It iterates over all bytecode files and triggers the Oyente scan. As parameters, Oyente will be configured to process the bytecode file and create an output JSON file. This JSON file contains the classification made by Oyente. The script, illustrated in Figure 5.8, is mounted inside the Docker container and triggered at the beginning of the execution. It is also required to configure the mount points properly for the input bytecode files and the JSON output files. This way, Oyente vulnerability scanning can be easily started and scaled.

When the scanning process has been finished, the JSON files are ready to be loaded into the database. One JSON file contains the classification of one smart contract, which is shown in Figure 5.9. The *integer_overflow* and *integer_underflow* classification are not available on the bytecode level. Therefore, they will not be loaded into the database.

```json
{
 "vulnerabilities": {
  "callstack": true,
  "reentrancy": false,
  "time_dependency": true,
  "integer_overflow": [],
  "integer_underflow": [],
  "money_concurrency": false
 },
 "evm_code_coverage": "81.9"
}
```

Figure 5.9: Sample classification of the Oyente tool.

Since the classification is now available on the filesystem, it needs to be loaded into the database. This step is realized with the Python module *oyente_classification*. Similar to the module's procedure presented in Section 5.3.2, a new table is created according to a schema, as illustrated in Table 5.7. The database table contains the id, address of the smart contract, and the Oyente classification files' name. For each vulnerability class provided by the Oyente tool, a column is created. The vulnerability class's value is set to value *1* when a flaw of a vulnerability type has been detected. Otherwise, the value is set to value *0*.

Column name	**Column type**	**Additional constraints**
id (PRIMARY)	int(7)	NOT NULL UNIQUE
address	varchar(100)	UNIQUE
classification_path	varchar(120)	DEFAULT NULL
callstack	tinyint(1)	DEFAULT NULL
reentrancy	tinyint(1)	DEFAULT NULL
time_dependency	tinyint(1)	DEFAULT NULL
money_concurrency	tinyint(1)	DEFAULT NULL

Table 5.7: Schema of the database created by module *oyente_classification*.

After finishing the input validation and database table creation, the classifications will be read from the file and loaded into the database. Eventually, the *oyente_classified* flag of the primary smart contract table with Schema 5.2 introduced in Section 5.2.3 is updated to the value *1* on success. When an error occurred, the flag is set to value *0*.

As we can see in Figure 5.10, approximately 27,5% of all smart contracts are classified with the flag *money_concurrency*. In general, plenty of representatives have been found for each classification class, except the *time_dependency* flag. Therefore, some new candidates have been found which can be passed to a deep-learning algorithm.

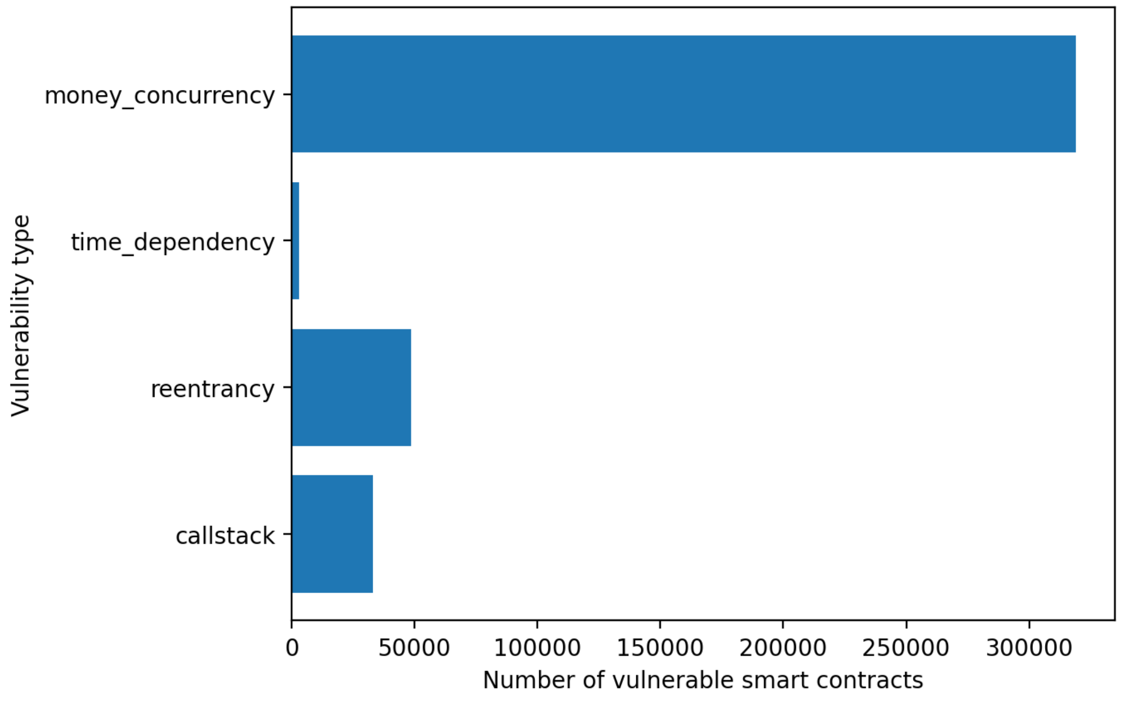


Figure 5.10: Distribution of the Oyente's classification labels.

5.3.4 Mythril Smart Contract Classification

Similar to the Oyente tool, examined in the previous section, the Mythril vulnerability scan has to be executed by ourselves. Therefore, it is also divided into the Mythril scanning process and the database update. In detail, we firstly produce classification output files via the Mythril tool inside of a Docker container. Secondly, a newly created Python module loads the classifications stored in the files into the database. This way, the Mythril classifications are stored centrally and can be managed for deep-learning dataset creation.

Mythril can also be deployed via a Docker container to increase the runtime execution speed. Inside of the Docker container, the tool is ready to be used right away. Like the Oyente vulnerability scan, a short bash script must be developed to perform the mass-scan-operation. It iterates over all bytecode files and triggers the Mythril scan. Since Mythril combines symbolic execution, SMT solving, and taint analysis; some parameters need to be configured. For performance issues, it is required to set various timeouts so that all smart contract files' processing does not go beyond the limits. The tool also gets the bytecode path as input and is told to produce a JSON file containing the scan results. The script, shown in Figure 5.11, and bytecode files are mounted inside the Docker container. The expected environment is partly configured in the script, so the mount points must be appropriately configured. This way, it is possible to scale and run the Mythril vulnerability scanning easily.

```
#!/bin/bash

# For each bytecode file mounted into the container
for bytecode in /home/mythril/bytecodes/*/*.bin;
do
    # Skip already classified smart contracts
    if [ -f "$bytecode.json" ] && [ -s "$bytecode.json" ];
      ↪ then
        echo "$bytecode␣already␣classified."
    # Run the Mythril classification
    else
        echo "Start␣with␣$bytecode."
        myth analyze -o json --execution-timeout 60 --create-
          ↪ timeout 30 --solver-timeout 1000 -f "$bytecode"
          ↪  > "$bytecode.json"
        echo "Finished␣with␣$bytecode."
    fi
done;
```

Figure 5.11: Script file to trigger Mythril vulnerability scan of mounted bytecode files.

When all available smart contracts have been scanned by the Mythril tool, the JSON files are ready to be loaded into the database. For each bytecode file, an output file has been generated. A sample classification file is illustrated in Figure 5.12.

```
{
  "error": null,
  "issues": [{
      "description": "The contract executes an external
        ↪ message call...",
      "function": "constructor",
      "swc-id": "107",
      "title": "External Call To Fixed Address",
      ...
    },
    {
      "description": "The return value of a message call
        ↪ is not checked...",
      "function": "constructor",
      "swc-id": "104",
      "title": "Unchecked Call Return Value",
      ...
    }
  ],
  "success": true
}
```

Figure 5.12: Sample classification of the Mythril tool.

All classifications can now be loaded from the files into the database. This procedure is realized with the Python module *mythril_classification*. For this third classification tool, a database table is created according to the schema of Table 5.8. The database table contains the id, address of the smart contract, and the Mythril classification files' name. For each vulnerability class provided by the Mythril tool, a column is created. The entry will be set to value *1* when the vulnerability has been detected in a smart contract. Otherwise, the value is set to value *0*.

Column name	**Column type**	**Additional constraints**
id (PRIMARY)	int(7)	NOT NULL UNIQUE
address	varchar(100)	UNIQUE
classification_path	varchar(120)	DEFAULT NULL
success	tinyint(1)	DEFAULT NULL
DEFAULT_FUNCTION_VISIBILITY	tinyint(1)	DEFAULT NULL
INTEGER_OVERFLOW_AND_UNDERFLOW	tinyint(1)	DEFAULT NULL
OUTDATED_COMPILER_VERSION	tinyint(1)	DEFAULT NULL
FLOATING_PRAGMA	tinyint(1)	DEFAULT NULL
UNCHECKED_RET_VAL	tinyint(1)	DEFAULT NULL
UNPROTECTED_ETHER_WITHDRAWAL	tinyint(1)	DEFAULT NULL
UNPROTECTED_SELFDESTRUCT	tinyint(1)	DEFAULT NULL
REENTRANCY	tinyint(1)	DEFAULT NULL
DEFAULT_STATE_VARIABLE_VISIBILITY	tinyint(1)	DEFAULT NULL
UNINITIALIZED_STORAGE_POINTER	tinyint(1)	DEFAULT NULL
ASSERT_VIOLATION	tinyint(1)	DEFAULT NULL
DEPRECATED_FUNCTIONS_USAGE	tinyint(1)	DEFAULT NULL
DELEGATECALL_TO_UNTRUSTED_CONTRACT	tinyint(1)	DEFAULT NULL
MULTIPLE_SENDS	tinyint(1)	DEFAULT NULL
TX_ORDER_DEPENDENCE	tinyint(1)	DEFAULT NULL
TX_ORIGIN_USAGE	tinyint(1)	DEFAULT NULL
TIMESTAMP_DEPENDENCE	tinyint(1)	DEFAULT NULL
SIGNATURE_MALLEABILITY	tinyint(1)	DEFAULT NULL
INCORRECT_CONSTRUCTOR_NAME	tinyint(1)	DEFAULT NULL
SHADOWING_STATE_VARIABLES	tinyint(1)	DEFAULT NULL
WEAK_RANDOMNESS	tinyint(1)	DEFAULT NULL
SIGNATURE_REPLAY	tinyint(1)	DEFAULT NULL
IMPROPER_VERIFICATION_BASED_ON_MSG_SENDER	tinyint(1)	DEFAULT NULL
REQUIREMENT_VIOLATION	tinyint(1)	DEFAULT NULL
WRITE_TO_ARBITRARY_STORAGE	tinyint(1)	DEFAULT NULL
INCORRECT_INHERITANCE_ORDER	tinyint(1)	DEFAULT NULL
ARBITRARY_JUMP	tinyint(1)	DEFAULT NULL
DOS_WITH_BLOCK_GAS_LIMIT	tinyint(1)	DEFAULT NULL
TYPOGRAPHICAL_ERROR	tinyint(1)	DEFAULT NULL

Table 5.8: Schema of the database created by module *mythril_classification*.

After finishing the input validation and database table creation, the Mythril scanning results will be read from the file and loaded into the database. Finally, the *mythril_classified* flag of the primary smart contract table with Schema 5.2 introduced in Section 5.2.3 is updated to the value *1* on success. When an error occurred, the flag is set to value *0*.

As Figure 5.13 shows up, many assert violations have been detected. Besides that, only a small number of vulnerability types have been detected on our set of bytecodes in a way that a deep-learning algorithm can quickly learn patterns. How the gathered labels can create a reasonable dataset for deep-learning will be covert in Section 5.4.3.

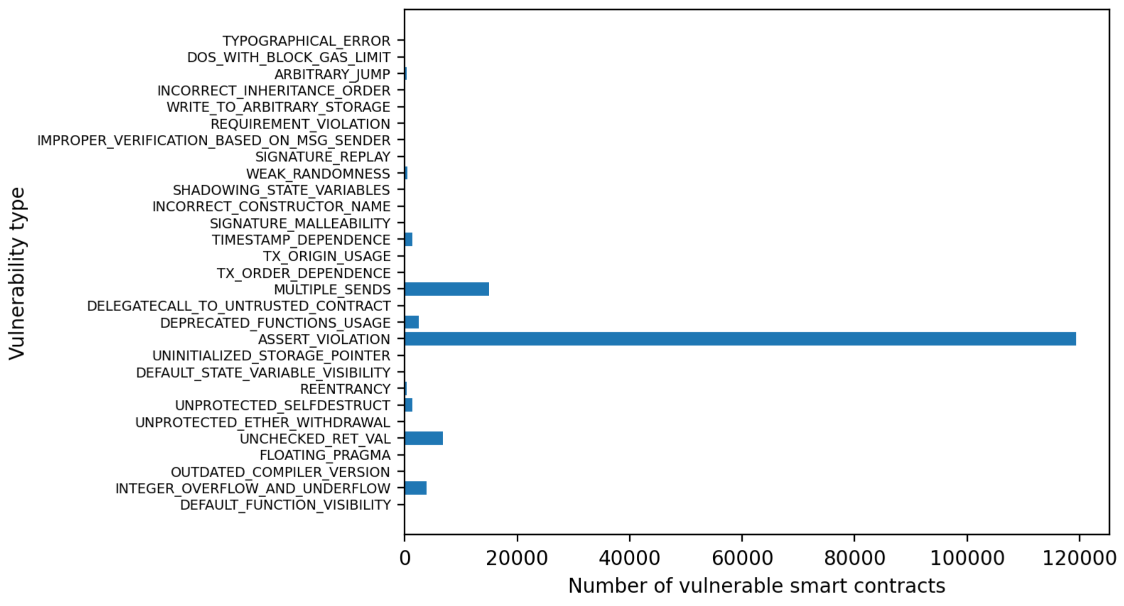


Figure 5.13: Distribution of the Mythril's classification labels.

5.4 Deep-Learning Classification

Since a bytecode file collection and its pre-labeling has been built up, the base for the deep-learning vulnerability scanning approach has been made. This data needs to be processed in a way that can be directly passed for the model training. In this section, all related algorithms will be introduced, leading to a model serving API, which can classify smart contract files by their plain bytecode files. First of all, the overall relationships between the components will be explained before each module is examined.

5.4.1 Architectural Overview

As illustrated in the architectural overview Figure 5.14, the plain bytecode files need to be pre-processed.

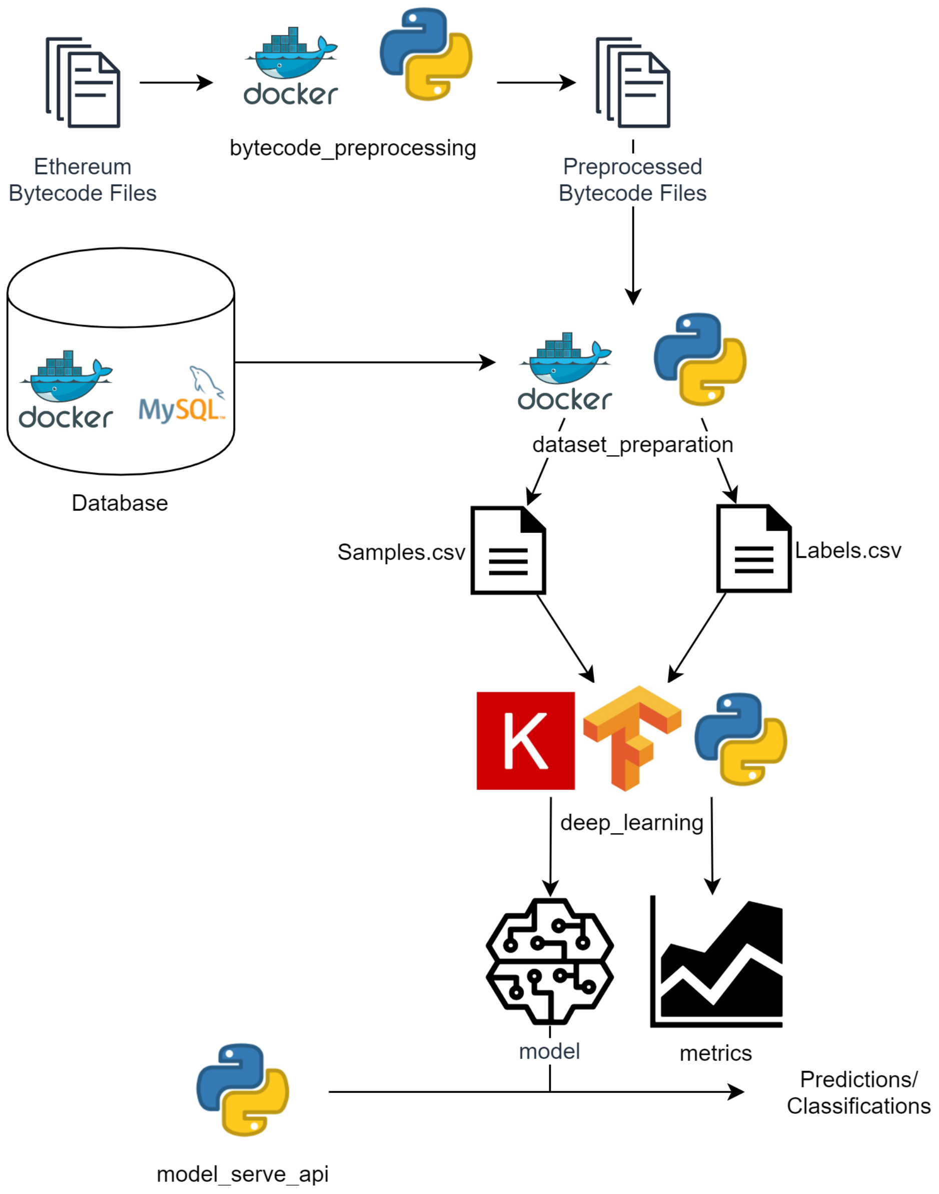


Figure 5.14: Architectural overview of components which lead to model-based classification.

The pre-processing is helpful to simplify the input for the deep-learning training. Thereby, it is essential to reduce complexity without losing meaningful data. Since common text classification methods should be applied, it is also necessary to separate the bytecode operations to increase the learning's information value. Therefore, the *bytecode_preprocessing* module will take care of it, producing output files containing the pre-processed bytecode. This module can also be executed inside a Docker container, as well as the next module. The next step, before actually training deep-learning models, is the preparation of a dataset. For this step, the pre-classification of previous chapters' tools stored in the database will be utilized. As a result, CSV files will be created, which can then be passed as input for the deep-learning algorithm. One file will be created containing the bytecode samples and another one storing the corresponding labels.

This leads us to the most challenging part of this project, the *deep_learning* module. For deep-learning, the *tf.keras* package will be used to realize a model training. Since Tensorflow should utilize the GPU's full power, this module will be executed on bare metal. In this step, a model capable of detecting vulnerability types will be trained. Based on the calculated metrics, it can be determined how performant the deep-learning model works. Finally, a model serving API is developed to see the trained model in action. It provides an endpoint where a plain bytecode file can be passed, resulting in a model prediction. This way, end-users are able to run vulnerability scanning with the newly created model.

5.4.2 Bytecode Preprocessing

The milestone has been reached with the collection of the pre-labeling of all available smart contracts. Similar to existing vulnerability scanning methods using artificial intelligence, the usual text classification methods will be applied to our context. Therefore, the bytecode needs to get some text-like properties to be applied with common text-classification deep-learning models. Currently, the bytecode is one long hexadecimal number that represents a particular operation sequence. This sequence needs to be pre-processed into a sequence of operations divided by a unique separator. It also makes sense to remove unnecessary bytecode in this step so that the alphabet used in the deep-learning process is as compact as possible. By this, the dimensions of the tensors created in the deep-learning process will be simplified.

In order to transform these bytecodes into meaningful data, all the bytecode files need to be pre-processed. However, it is crucial to keep them meaningful data to detect reliable patterns. Therefore, we have to take a more in-depth look into the smart contract bytecode file to find reasonable simplification rules.

```
6060604052361561006157 60e060020a60003504631460 08e38114610063
    ↪ 578063667d5d22146100a25780638b299903146100ab578063960e
    ↪ dffb146100b7578063d4f77b1c146101c5578063f71d96cb146102
    ↪ 20578063f7c1f50a146102535 75b005b610329600435602435 6002
    ↪ 826003811015610002575082018160038110156100025750506020
    ↪ 808204909201600201549190066101000a900460ff1681565b6103
    ↪ 366005548156 5b61032960015460ff16 ...
```

Figure 5.15: Untouched sample bytecode file which needs to be pre-processed.

In Figure 5.15, an untouched bytecode file of a smart contract is shown. A look in the Ethereum whitepaper is required to understand how the bytes represent the EVM operations. In this paper, the rules of how to create disassembler code from the bytecode are listed. The following properties apply to the bytecode of a smart contract:

- First of all, one byte represents one operation. Therefore, two hexadecimal digits can be decoded into one disassembler operation. This fact means that there can be only 256 different operations at most. In fact, 135 disassembler operations are defined for the Ethereum bytecode.

- Secondly, only one operation type expects a dynamic parameter with dynamic input length, the *PUSH* operation. The input length can be at most 32 bytes large.

- Since all operations work based on a stack, multiple operations have the same functionality. However, the number of dimensions, such as the number of stack items, can vary.

Since we know how to interpret the bytecode, we can think of different pre-processing rules. The focus should always be the performance of the deep-learning. Therefore, it is essential to simplify the text without losing meaningful data:

- First, the parameter needs to be removed since the varying input would drastically increase the alphabet complexity.

- All bytes where a disassembler operation is not defined should be explicitly substituted with a common representation. This way, the deep-learning algorithm should understand it the easiest way.

- All operations with the same functionality working in different dimensions can be merged into one common operation representation.

- A standard separator should be added to the sequence of disassembler operations. This separator can be whitespace so that the bytecode can be read kind of like a human-readable text.

- The bytecode representation of the operation can be kept. In terms of text classification, the two hexadecimal digits are interpreted as words. Therefore, all entries of the alphabet remain unique.

Since these basic rules are defined, they need to be applied to the bytecode files. Therefore, a new Python module has been developed, which creates pre-processed bytecode files. These bytecode files will then directly be passed into the deep-learning algorithm to learn the patterns and finally find vulnerabilities. The module will be structured so that different rules can be easily attached and applied to bytecode files. In this case, two sets of rules are defined. The first one removes all parameters from the *PUSH* command, keeping the operation representation untouched. Since every *PUSH* command defines the input length, these parameters can be removed systematically. The separator will be added as well as a common representation of unknown commands. In the module, this ruleset is defined as default normalization. The second rule set works almost the same. It additionally packages common functionalities to one unified operation representation. For instance, the *PUSH1 - PUSH32* commands are represented by the bytes *0x60 - 0x7f*. After processing the bytecode files, all *PUSH* operations will be represented by *0x60*. This way, the alphabet dimensions are reduced by 31. This simple substitution will be applied to all functions which work in different dimensions. Since this ruleset reduces the complexity of the bytecode, we call it reduction normalization.

```
INVALID_INSTRUCTION = "XX"

INSTRUCTION_RULES = {
    "00": {
        "mnemonic": "STOP",
        "parametersize": 0,
        "replace": "00"
    },
    "01": {
        "mnemonic": "ADD",
        "parametersize": 0,
        "replace": "01"
    },
    #...
    "60": {
        "mnemonic": "PUSH1",
        "parametersize": 1,
        "replace": "60"
    },
    "61": {
        "mnemonic": "PUSH2",
        "parametersize": 2,
        "replace": "60"
    },
    "62": {
        "mnemonic": "PUSH3",
        "parametersize": 3,
        "replace": "60"
    },
    #...
}
```

Figure 5.16: Sample rule definition for bytecode subsitution.

Figure 5.16 shows how the rules are defined in the Python module. For each operation, the parameter size, the replacement, and the mnemonic are defined. This way, the incoming bytecode can be processed byte for byte and substituted accordingly.

Since this module's concept should be clear at this point, we take a brief look at the implementation details. The algorithm is executed by passing the normalization method, the path to the location where the source bytecode files are located, and the pre-processed bytecode files should be stored. Additionally, the *op_length* will be tracked in the main smart contract database, defined in Table 5.2, to keep track of each smart contract's sequence length. After validating the configuration, all bytecode files available at the specified location will be processed by the algorithm. Byte for byte, the rules defined the way as illustrated in Figure 5.16, are applied to the bytecode and stored into a new file. Parameters will be skipped. In the end, the resulting output of the default normalization is shown in Figure 5.17, and the pre-processed bytecode of the reduction method is shown in Figure 5.18.

```
60 60 52 36 15 61 57 60 60 0a 60 35 04 63 81 14 61 57 80 63
  ↪ 14 61 57 80 63 14 61 57 80 63 14 61 57 80 63 14 61 57
  ↪ 80 63 14 61 57 80 63 14 61 57 5b 00 5b 61
...
90 91 16 31 04 90 82 81 81 81 85 88 83 f1 50 50 50 50 50 56
  ↪ 00 XX XX XX XX 54 8b 62 45 XX 88 38 6f 63
```

Figure 5.17: Pre-processed bytecode according to the *default* normalization rules.

As we can see in both pre-processed bytecode files, the bytecode is transformed into 2-digit hexadecimal numbers with dividing whitespaces. Invalid operations have been substituted with the value *XX*. If we have a closer look, we can see that Figure 5.18 does not contain any numbers between *0x61* and *0x7f*. These values have received the common value *0x60*. This behavior can be observed for the other operations as well, such as the values between *0x80* and *0x8f* in line 3 of both figures.

```
60 60 52 36 15 60 57 60 60 0a 60 35 04 60 80 14 60 57 80 60
  ↪ 14 60 57 80 60 14 60 57 80 60 14 60 57 80 60 14 60 57
  ↪ 80 60 14 60 57 80 60 14 60 57 5b 00 5b 60
...
90 90 16 31 04 90 80 80 80 80 80 80 80 f1 50 50 50 50 50 56
  ↪ 00 XX XX XX XX 54 80 60 45 XX 80 38 60 60
```

Figure 5.18: Pre-processed bytecode according to the *reduction* normalization rules.

As a result of this module, we retrieved pre-processed bytecode files, which can then be passed to the deep-learning algorithm without further adjustments. The rulesets developed in this module are also packaged in a separate Python package so that other modules can also pre-process bytecode. These rulesets will be required by the model serving API where the plain bytecode is passed to the algorithm returning the model prediction to the end-user. The detailed behavior is presented in Section 5.4.5.

5.4.3 Dataset Preparation

For supervised deep-learning, it is required to prepare a designated dataset. This dataset contains the raw samples, pre-processed bytecode files in our case, and the associated labels. After executing the module presented in the previous section, the pre-processed bytecode is available for all smart contracts. Therefore, we need to define a meaningful dataset. For this work, a dataset will be selected with a certain amount of representatives and a nearly equal distribution. This way, we reduce the chance of overfitting. As we have seen in Section 5.3, vulnerability classes' distribution varies between 0 to approximately 320.000. The most represented vulnerability classes are:

- Callstack detected by Oyente: Due to the fact that the call-stack's depth limit is 1024, all transactions fail, and an attacker can take advantage of that [11].
- Reentrancy detected by Oyente: As explained in Section 2.1.2, the reentrancy bug leads to recursive calls that can be caused by the manipulation of the fallback function.
- Money concurrency detected by Oyente: A smart contract that can be executed concurrently requires shared resources. This way, this resource might be accessed by concurrent processes [11].
- Accessible self-destruct detected by Dedaub's Contract-Library: A smart contract provides a selfdestruct function publicly callable [36].
- DoS (Unbounded Operation) detected by Dedaub's Contract-Library: The smart contract contains a loop that operates on a scalable data structure. By continuously increasing the size, it can result in a public call or the denial-of-service [36].
- Tainted selfdestruct detected by Dedaub's Contract-Library: A smart contract forwards a balance to a recipient when calling its selfdestruct function. However, the recipient can be externally changed [36].
- Assert violation detected by Mythril: A code segment in a smart contract is always incorrect. With external manipulation, unexpected behavior can be triggered [75].
- Multiple Sends detected by Mythril: The smart contract does not handle exceptional conditions, which can lead to unexpected behavior [75].

These classes will be used for the deep learning algorithm. A maximum of 15.000 representatives of each vulnerability class will be added to create an equally sized distribution. The dataset will also be completed with 15.000 completely clean smart contracts where no vulnerabilities have been detected by the tools used in this project. Since all data is managed in a MySQL database, a dataset can be easily queried. The Python module *dataset_preparation* takes care of creating the dataset and loading it into CSV files, which can be passed directly to the deep-learning algorithm. This module requires the path to the pre-processed bytecode files, the database connection information, and the target file location. After the input validation, a new database table is created with a script collecting smart contracts by a query. This query is shown in Figure 5.19.

```sql
CREATE TABLE dataset AS
    SELECT *
    FROM (
        /* Unflagged smart contracts
        */
        (SELECT
            sc.id, sc.address, sc.bytecode_path, oyente.callstack, oyente.reentrancy, oyente.money_concurrency, ...
        FROM smart_contracts sc
        JOIN oyente_classification oyente ON sc.id = oyente.id
        JOIN dedaub_classification dedaub ON sc.id = dedaub.id
        JOIN mythril_classification mythril ON sc.id = mythril.id
        WHERE oyente.callstack = 0 AND oyente.reentrancy = 0 AND oyente.money_concurrency = 0 AND ...
        ORDER BY sc.id ASC
        LIMIT 15000)

        UNION

        /* Smart contracts with oyente callback flag
        */
        (SELECT
            sc.id, sc.address, sc.bytecode_path, oyente.callstack, oyente.reentrancy, oyente.money_concurrency, ...
        FROM smart_contracts sc
        JOIN oyente_classification oyente ON sc.id = oyente.id
        JOIN dedaub_classification dedaub ON sc.id = dedaub.id
        JOIN mythril_classification mythril ON sc.id = mythril.id
        WHERE oyente.callstack = 1
        ORDER BY sc.id ASC
        LIMIT 15000)

        UNION ...
    ) AS tmp
    ORDER BY tmp.id ASC;
```

Figure 5.19: SQL script to create a dataset for the deep-learning process.

It realizes the demands defined at the beginning of this section. Once the table has been created, the only job left for this module is storing the database entries into CSV files, which can be directly passed to the deep-learning algorithm. A snapshot of the CSV files is illustrated in Figures 5.20 and 5.21.

```
"id","bytecode"
"1","60 60 52 60 35 60 90 04 80 60 14 60 ..."
"2","60 60 52 60 35 60 90 04 80 60 14 60 ..."
"3","60 60 52 60 35 60 90 04 80 60 14 60 ..."
```

Figure 5.20: Output *samples.csv* which will be directly passed to the deep-learning model.

```
"id","callstack","reentrancy","money_concurrency","Accessible
   ↪ selfdestruct","DoS (Unbounded Operation)","Tainted
   ↪ selfdestruct","ASSERT_VIOLATION","MULTIPLE_SENDS"
1,1,0,0,0,0,0,0,0
2,0,0,0,0,0,0,0,0
3,0,0,0,0,0,0,0,0
```

Figure 5.21: Output *labels.csv* which will be directly passed to the deep-learning model.

In the end, a dataset is defined with the following distribution, which is shown in Figure 5.22. It contains 93.497 smart contracts. Since a smart contract can also be classified multiple times, there are some small outliers. However, the distribution is relatively balanced throughout the dataset.

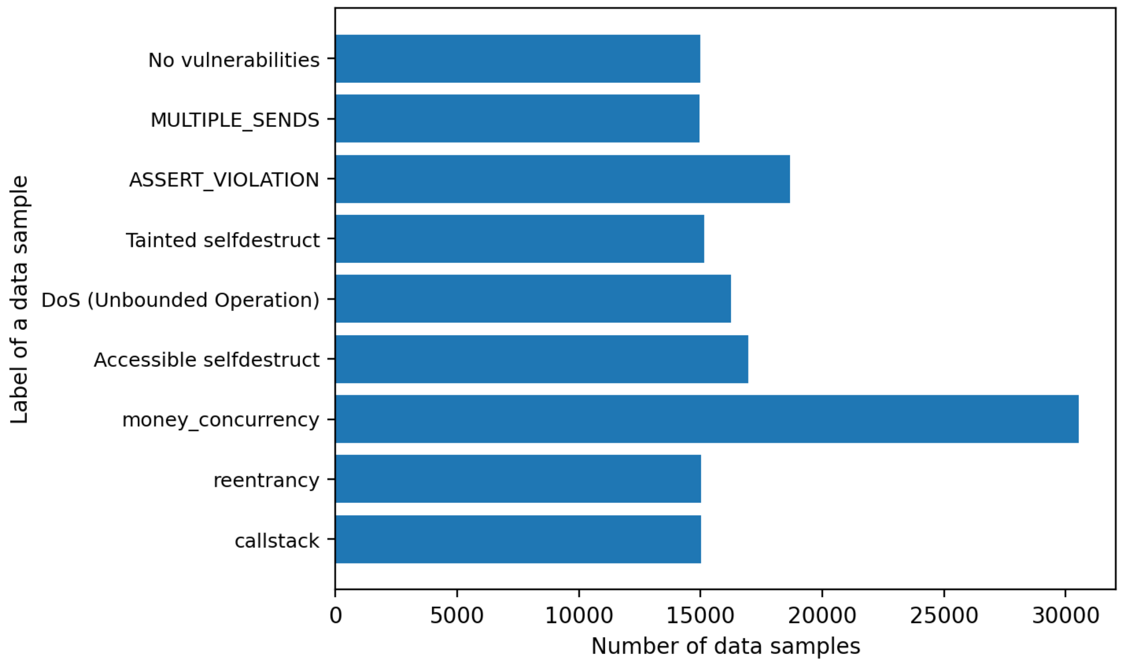


Figure 5.22: Distribution of all labels in the dataset.

5.4.4 Deep-Learning

After retrieving, pre-classifying, and pre-processing the bytecode files, we can start working on the deep-learning process. This section presents the heart of this work, which is developing a deep-learning model capable of detecting vulnerabilities by its bytecode. Therefore, we first take a look at certain challenges and requirements for this framework. Afterward, the module's architecture is introduced before looking at the workflow inside this module.

5.4.4.1 Requirements to the Deep-Learning framework

In general, the approach is similar to common text classification scenarios. Therefore, common model architectures can be used for our approach as well. In typical NLP problems, it is typical to use supervised word embeddings. These embeddings are often used on the sentence level for typical NLP problems. Since we do not have sentences in our smart contracts, the whole bytecode is passed to the Embedding layer. Although the bytecode is pre-processed, the length of smart contracts is varying a lot. This problem can be observed in Figure 5.23.

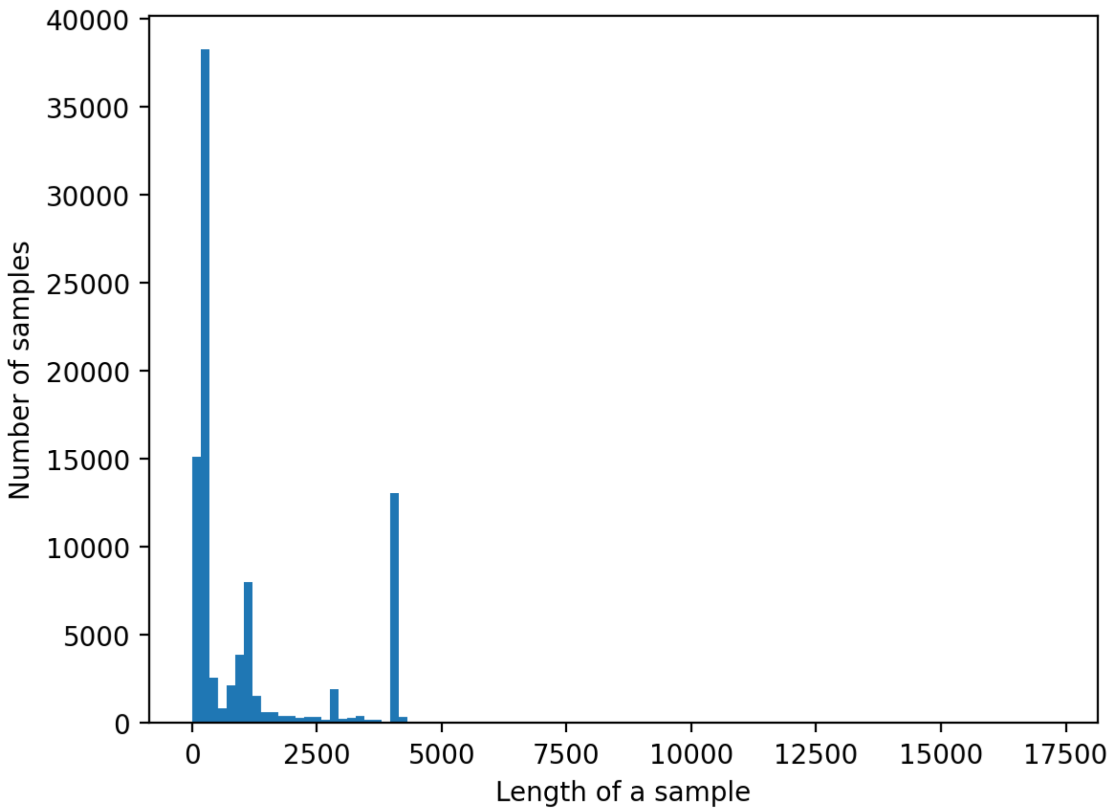


Figure 5.23: Distribution of all the sequence length in the dataset.

By defining a high dimension for the first layer, the memory usage and the learning time is inefficient. On the one hand, we could think of chunking the bytecode files and chunking the dataset to resolve this problem on the other hand. Since our labeling is based on the smart contract as a whole, the first option is not feasible. This approach would require a pre-labeling on the bytecode chunk level. Therefore, the dataset will be chunked to overcome this problem. The detailed process of running the deep-learning on dataset chunks is explained later in this chapter.

Another challenge, while creating a decent deep-learning framework, is the difference in model types. Different output dimensions require different ways of measuring deep-learning metrics. As indicated in Figure 5.24, a multi-output-layer (MOL) model requires a multi-dimensional input vector. It produces metrics for each output layer at runtime. These produced statistics can not be merged at runtime. In contrast, metrics on single labels can not be calculated at runtime for single-output-layer (SOL) models. However, after finishing learning one chunk, the metrics provided by both model types will be calculated to get comparable results. Therefore, the output vectors are split in the case of SOL models and merged in the case of MOL models to calculate the metrics defined for the other model type.

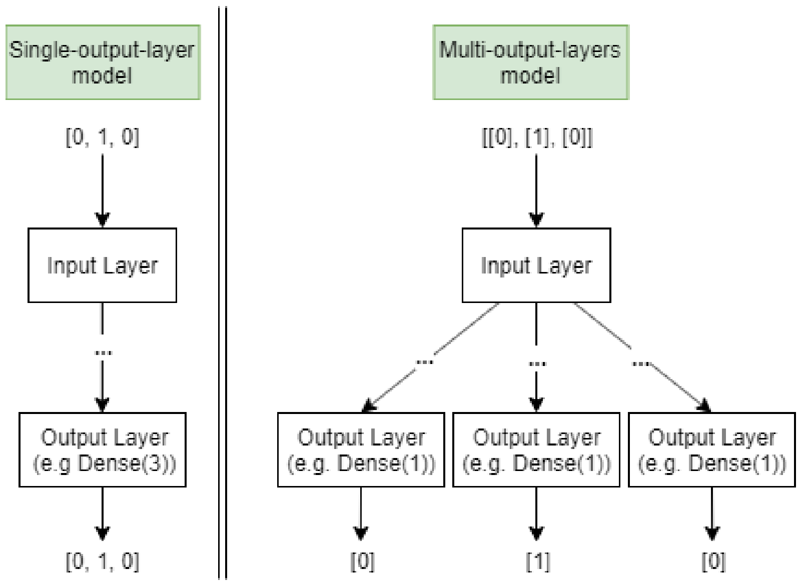

Single-output-layer model
[0, 1, 0]
Input Layer
...
Output Layer (e.g Dense(3))
[0, 1, 0]
Multi-output-layers model
[[0], [1], [0]]
Input Layer
...
...
...
Output Layer (e.g. Dense(1))
Output Layer (e.g. Dense(1))
Output Layer (e.g. Dense(1))
[0]
[1]
[0]

Figure 5.24: Comparison of input and output vectors in single-output-layer and multi-output-layers models.

Therefore, the framework should work in a generic way so that all models, regardless of single-output-layer or multiple-output-layers model, can be defined and used for the learning process. We should be able to easily adjust and add model architectures without causing additional development effort. Since we introduced the deep-learning framework's demands, we can look at the newly created Python module *deep_learning*.

5.4.4.2 Module architecture

Before we start covering this module's workflow, we first look into the module's architecture shown in Figure 5.25. This image shows the relationship between all components. First of all, components can be separated into the following four different component types:

- Abstract classes: These classes define shared functionality that is required in multiple deep-learning processes. The exact implementation can differ in different scopes depending on the inheritance structure.
- Shared components: These components are shared for each model type and are executed independently of the algorithm configuration or the model architecture.
- Single-Output-Layer (SOL) and Multi-Output-Layer (MOL) model related components: Since the model architecture might require different implementations, various components are specialized for that model type. The main difference is the output vector dimension. Since all metric calculation is based on the output vector, some functionality is separated in SOL and MOL components.

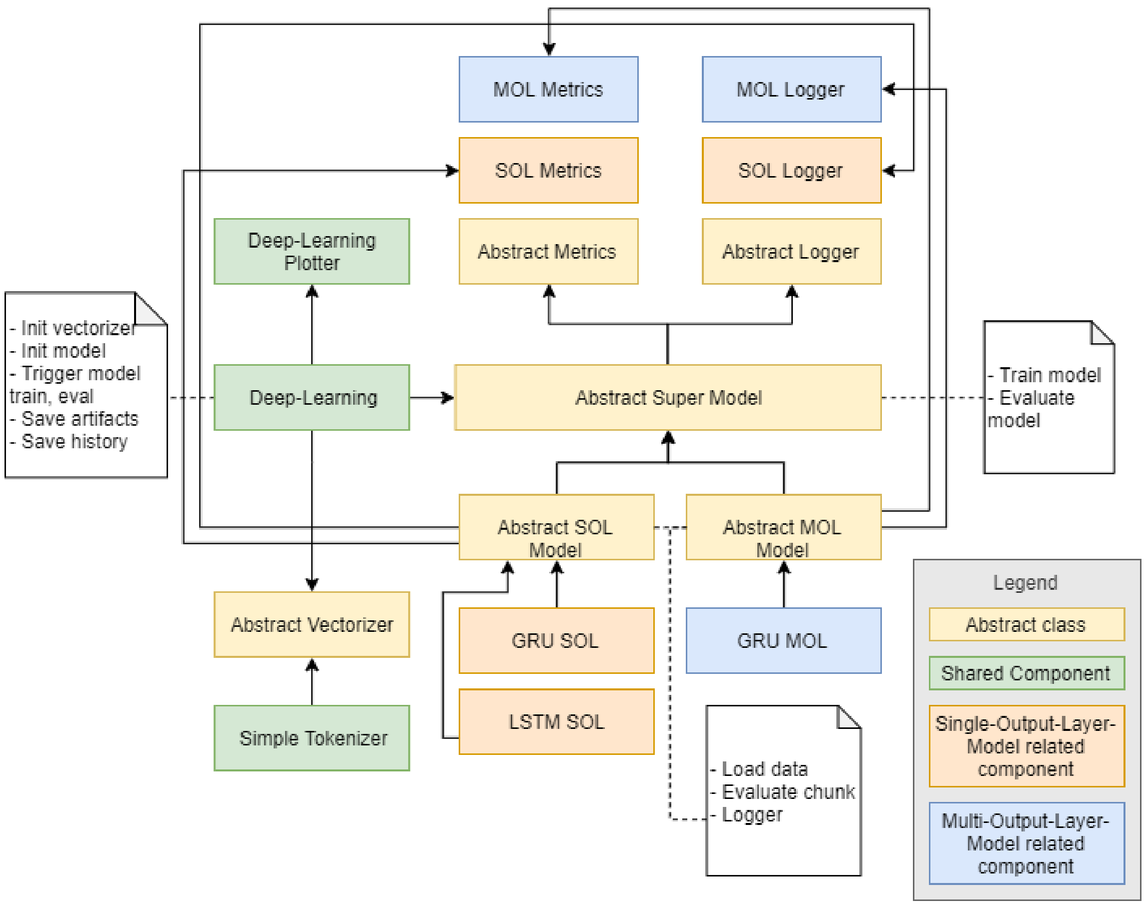


Figure 5.25: Component view of deep-learning related modules.

The central *Deep-Learning* class is the starting point of the algorithm. It can also be seen as the overall manager of the model learning execution. After the input is validated, the vectorizer is initialized. In this case, a simple text tokenizer is used, which is used to transform text sequences into integer vectors. In this step, the text in the form of a vector is converted to a unified sequence length of 4000. Therefore, the vector is either truncated or padded. The deep-learning algorithm can then process these vectors. Afterward, a specified model, such as a *GRU SOL*, will be initialized. Each model inherits the functionality of the *Abstract Super Model*. Since the training and evaluation process runs the same way for both models, the routine is defined here. It internally calls abstract methods implemented in either the *Abstract SOL model* or the *Abstract MOL model*. Since there is a difference in the input dimension, the sample loading requires different implementations. The same thing needs to be realized for the chunk evaluation. A *Abstract Logger* will be attached to the deep-learning process so that the metrics can be traced at runtime. This Abstract Logger traces the learning time for each chunk and provides functionality to log all metrics after one chunk learning has been finished. The *SOL Logger* and *MOL Logger* simply trace the metrics calculated at runtime.

Table 5.9 shows the metrics which can be calculated at runtime by its model type. As we can see, only the subset accuracy, the subset true and false prediction cannot be calculated by the MOL models at runtime. However, the internal calculations run in a completely different way. In a scenario with five label types and a batch size of 64, the SOL metrics are calculated on a 64x5 tensor. In contrast, the MOL metrics are calculated once for each of the five output layers on a 64x1 tensor. The metrical result is, in either case, the averaged value of the whole batch. The opposite metric method is calculated manually

Metrics function	**SOL model**	**MOL model**
accuracy	X	X
precision	X	X
recall	X	X
f1	X	X
auc	X	X
subset_accuracy	X	
hamming_loss	X	X
jaccard_similarity	X	X
true_positive	X	X
true_negative	X	X
false_positive	X	X
false_negative	X	X
subset_true_prediction	X	
subset_false_prediction	X	

Table 5.9: Usage of metric functions by model type.

after a chunk has been trained for the specified number of epochs. This way, SOL models create label-class-specific metrics, and MOL models calculate subset-specific metrics. It increases transparency if a model performs efficiently for each label class. This presented chunk evaluation is triggered inside the *Abstract SOL Model* and the *Abstract MOL Model*.

As seen in Table 5.9, many metric functions can be defined for both model types. These calculations are defined in the *Abstract Metrics* component. The subset-specific metrics are defined in *SOL Metrics*. This component also takes care of manipulating the matrix to create vectors for label-specific metric calculations. Similar to that, the *MOL Metrics* provides the functionality to merge all output vectors to create subset-specific metrics. With that knowledge about the *Abstract Super Model*, we come back to the functionality of the main *Deep-Learning* component. When the model is initialized, this component is responsible for triggering the deep-learning process's training and evaluation phase. In the end, the history and plots which will be saved for further analysis. The artifacts are stored as well to enable the possibility to create a model serving program. These artifacts include an information file as well as the actual model and the used tokenizer.

5.4.4.3 Deep-Learning workflow

With the knowledge of the module's architecture presented in the previous section, we will now have a more in-depth look into the workflow. It receives specific arguments to realize a proper configuration:

- the location of the dataset files (samples.csv, labels.csv)
- the desired target location
- the type of model which should be used for the deep-learning
- the number of dataset iterations
- the number of chunk epochs

As always, the input is validated before the main process is started. After configuring the number of iterations and epochs, multiple resources get initialized. At first, the Tensorflow session is initialized. Afterward, the vectorizer is initialized and passed to the model creation in the next step. As mentioned in Section 5.4.4.1, the dataset needs to be chunked since the bytecode files contain too large sequences. The dataset chunking works as follows: The samples and corresponding labels are read from the CSV files. Since the id column is not required for smart contract tracking anymore, it will be removed. The samples and labels will be merged into one *DataFrame* to keep the correlation between samples and labels. Afterward, the dataset is shuffled and temporarily stored in a new sample and label file. Finally, the chunks will be created from the temporary file and stored into chunk files according to a defined chunk size. The default chunk size is set to 1.024. Since the chunk datasets are small enough, they can be passed to the deep learning algorithm without running into memory issues. In the next step, training and test datasets will be defined. Therefore, 80% of the chunks will be used for training, and 20% for the final model testing. With these preparation steps, we are ready to start training the model.

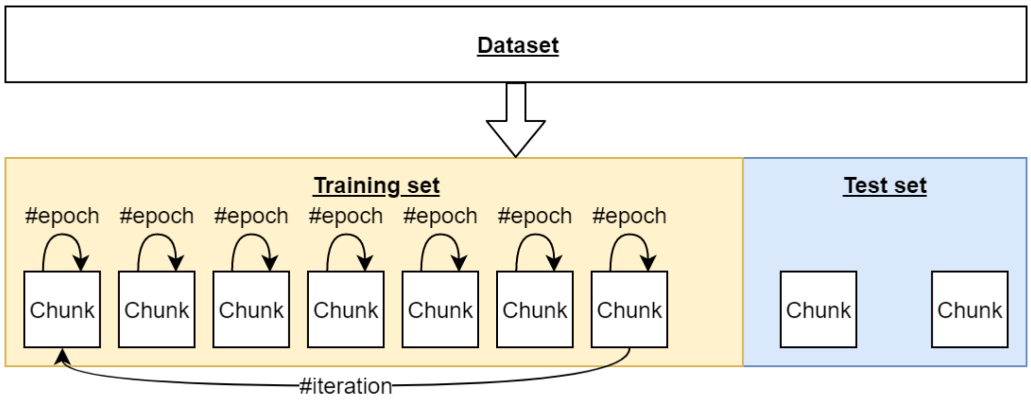


Figure 5.26: Illustration of the chunked-based deep-learning approach.

As illustrated in Figure 5.26, for each created dataset chunk, the model will be trained according to the configured number of epochs and the number of iterations. The number of epochs defines how often the chunk will be used for training before switching to the next chunk. In contrast, the number of iteration defines how often all the chunked training sets should be passed for training. Before the chunked data samples can be passed to the model's input layer, the bytecode sequence needs to be vectorized. In this case, a simple tokenizer is used, which creates numeric vectors. In addition to that, a fixed *MAX_SEQUENCE_LENGTH* is applied to the vectors. Sequences shorter than the length are padded in the beginning, and sequences longer are truncated.

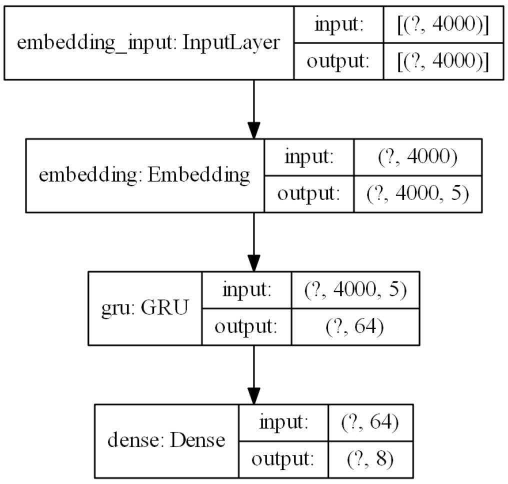
When the tokenized samples are passed to the model fitting function, 10% of a chunk will be used as a validation set to prevent overfitting. When one chunk training step has been finished, the same chunk will be used for evaluation purposes. This way, all defined metric functions for SOL and MOL models will be calculated here. The pseudo-code of explained behavior is shown in Figure 5.27. The results will be stored in a metric history object to keep track of the progress. When the training has been finished, the train scores are calculated and tracked as well. In the testing phase, the remaining chunks will be passed to the model to calculate the overall performance. All metrics are tracked and stored in the form of plots and JSON files. Inside the JSON file, every measurement is stored in arrays. For potential applications, all required artifacts to run the model for any predictions are stored as well.

```
for i in range(ITERATION):

    for id in train_IDs:
        #...
        # Import data
        data_samples, labels = load_data(sample_file_path, label_file_path)

        # Vectorize samples
        samples = vectorizer.vectorize_text(data_samples)

        # Create validation sets
        samples_train, samples_val, labels_train, labels_val = _split_chunk(
            samples_converted, labels_converted)

        # Train the model
        history = model.fit(samples_train,
                            labels_train,
                            batch_size=BATCH_SIZE,
                            epochs=EPOCHS,
                            verbose=1,
                            validation_data=(samples_val, labels_val),
                            callbacks=[model_logger])

        # Get score of train chunk
        evaluate_chunk(model, samples_train, labels_train,
                                train_scores)
```

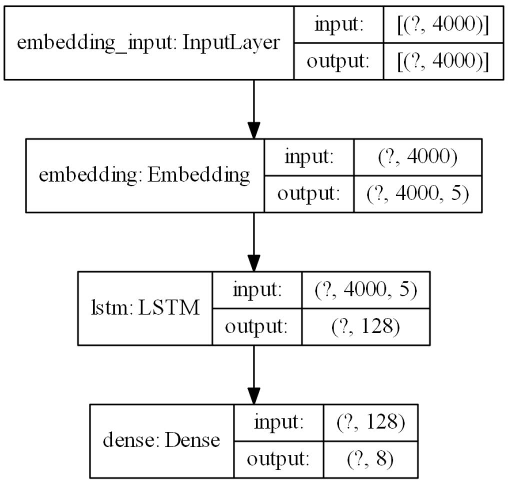
Figure 5.27: Pseudo code of running deep-learning based on dataset chunks.

5.4.4.4 Deep-Learning models

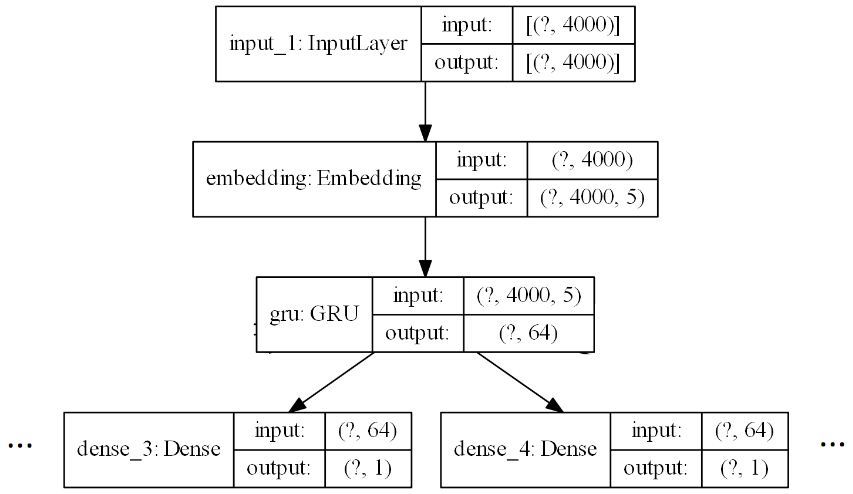
Since the architecture and the workflow of the deep-learning process should be clear at this stage, we will look at the different model architectures used in this work. As mentioned in the previous sections, different models have been defined to realize the deep-learning vulnerability scanning approach. Therefore, simple Recurrent Neural Networks (RNN) have been designed, commonly used for text classification scenarios. The three models, illustrated in Figure 5.28, will be trained and compared.



(a) Architecture of the single-output-layer GRU model.



(b) Architecture of the single-output-layer LSTM model.



(c) Architecture of the multi-output-layers GRU model.

Figure 5.28: Architecture of neural networks used in this project.

Figure 5.29 shows a code snippet of the multi-output-layer GRU model creation. For this model, an embedding is defined with fixed dimension size. The input length determines the other dimension. A dropout is configured to prevent overfitting, which means that certain neurons will not be used in the learning process. Afterward, all layers are connected. For each vulnerability class, an output layer is created, also defining the activation function. For the whole model, the loss function is set to binary_crossentropy.

```python
def create_model(self, num_features: int, input_length: int)
    -> Model:
"""Creates an instance of a GRU model without pretrained
    Embedding."""

    input_layer = Input(shape=(input_length, ))
    embedding_layer = Embedding(input_dim=num_features,
                                output_dim=self.
                                    EMBEDDING_DIMENSION,
                                input_length=input_length)(
                                    input_layer)
    gru_layer = GRU(64, dropout=0.2,
                    recurrent_dropout=0.2)(embedding_layer)

    outputs = []
    for i in range(self.num_label_classes):
        output = Dense(1, activation='sigmoid')(gru_layer)
        outputs.append(output)

    model = Model(inputs=input_layer, outputs=outputs)

    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=MOL_METRICS)
```

Figure 5.29: Code definition of the multi-output-layer GRU model in Python.

The performance of these models will be evaluated in Section 6. However, since the LSTM and GRU are both RNN, the difference is pretty small. With the GRU layer, a very lightweight layer has been used, which reduces the training time. If the vulnerability types' complexity is increased, it might be helpful to switch onto layers with more trainable parameters.

5.4.5 Model serving API

The *deep_learning* module of the previous section creates models that can classify smart contracts. To see a model in action, a new Python module providing a REST API endpoint that can be used to produce model predictions on bytecode files. Since an end-user does not want to take care of the bytecode transformation, the API takes care by itself. The produced artifacts during the deep-learning process will be passed to run the API. These artifacts include the stored model and the vectorizer used for this model. Some additional configuration is passed so that the API can internally map the labels to actual vulnerability class names. A sample configuration file is shown in Figure 5.30.

```
{
  "label_classes": [
    "callstack",
    "reentrancy",
    "money_concurrency",
    "Accessible selfdestruct",
    "DoS (Unbounded Operation)",
    "Tainted selfdestruct",
    "ASSERT_VIOLATION",
    "MULTIPLE_SENDS"
  ],
  "model_path": "MOL_GRU_model.h5",
  "model_type": "MOL_GRU_model",
  "vectorizer_max_seq_length": 4000,
  "vectorizer_path": "simple_tokenizer.json",
  "vectorizer_type": "simple_tokenizer"
}
```

Figure 5.30: Sample configuration file which is passed to the model serving API and can be retrieved via endpoint.

After the input validation, a Tensorflow session is created for the API to run predictions on the model. It is ensured that all required resources defined by the configuration are loaded correctly. Therefore, the vectorizer used for the model is initialized as well as the model. It is also required to load the metrics used during the deep-learning process since it will be used for the prediction. For the API itself, a Flask API [76] will be used. This Python package provides a simple way to create API endpoints. In our case, we will open up two different endpoints. The first one shows the config passed to this Python module, as illustrated in Figure 5.30. The second endpoint will trigger the model prediction. The endpoint is available at */model-predict*, where plain bytecode of smart contracts can be passed in the request body. The expected body should look, as shown in Figure 5.31.

```
{
  "smart_contract": "606060405236156100
      ↪ f8576000357c010000000000..."
}
```

Figure 5.31: Sample request body when calling the */prediction* endpoint.

Internally, the smart contract's bytecode needs to be pre-processed before it can be passed to the model. Therefore, it is firstly transformed according to the substitution rules defined in Section 5.4.2. Afterward, the resulting pre-processed bytecode will be vectorized using the same tokenizer used for the learning process with its initialized values. It is crucial to store and load the previously used vectorizer since the bytecode operations will otherwise be mapped to other values than expected by the model. When the bytecode is vectorized, it can be passed to the model, predicting if certain smart contracts' vulnerability flaws can be detected. In addition to that, the prediction time is tracked and shown to the user. The response from this endpoint is shown in Figure 5.32.

```
{
  "model_prediction_time": "0:00:00.743833",
  "prediction": {
    "ASSERT_VIOLATION": 0.00010630795441102237,
    "Accessible selfdestruct": 0.9998799562454224,
    "DoS (Unbounded Operation)": 0.9996991157531738,
    "MULTIPLE_SENDS": 0.0012760674580931664,
    "Tainted selfdestruct": 0.9998599290847778,
    "callstack": 0.9995280504226685,
    "money_concurrency": 0.0013179528759792447,
    "reentrancy": 0.0009357615490444005
  },
  "prediction_time": "0:00:00.784721"
}
```

Figure 5.32: Sample request response containing the model prediction when calling the */prediction* endpoint.

As we can see, the probabilities for each vulnerability class are returned to the end-user. For the smart contract passed to the API in this example, the model predicts that the smart contract contains the vulnerability types *Accesible selfdestruct*, *DoS (Unbound Operation)*, and *callstack*. We can also see the measured prediction time until the model finished the classification. The total prediction time, including the contract transformation, is returned as well. This way, a smart contract can now be classified by the deep-learning model from the end-user perspective.

6. Evaluation

In this section, the metrics calculated during training in Section 5.4.4 are examined and compared. We executed the deep-learning process for three different models, as shown in Section 5.4.4.4. These three models are, on the one hand, a single-output-layer GRU and LSTM model and, on the other hand, a multi-output-layer GRU model. In addition to that, the number of iterations and the number of epochs have been mixed up to investigate how these changes influence the overall performance.

Before diving deeper into the results, we have a look at the metric functions calculated during the whole process.

6.1 Definition of the Metrics

As already listed in Table 5.9, plenty of statistics have been registered. For single-output-layer models, metrics are calculated on the whole output subset. However, in contrast to that, the metrics are measured by each class in multi-output-layer cases. In the case of our AI Scanner, we only receive binary output for multiple classes. Therefore, the predicted value can only be true or false. In the following, these metrics are defined.

Loss function

The loss function is a crucial part of the training. The goal of the whole training process is to optimize the loss function and therefore minimize the error. The loss gives an indication of the model's prediction quality. The calculated values are higher when the prediction is far away from the expected result. In our case, the binary cross-entropy function is used for all three models. This function is used since we have multiple binary output labels. It is defined as follows with the expected value y and the predicted value $\hat{y}$:

$$L_{BCE}(y, \hat{y}) = -(ylog(\hat{y}) + (1 - y)log(1 - \hat{y})) \tag{6.1}$$

Since we have a relatively equal data distribution among the class types, this function should work well in our case [77].

True Positives, True Negatives, False Negatives, False Positives

The results of true positives (TP), true negatives (TN), false negatives (FN), and false positives (FP) are the base value for various other metric functions. The true values represent the number of correct predicted values, which can be either true positive or true negative. In contrast to that, the false values indicate that the model calculated the wrong value [22]. Since these values indicate the error of a deep-learning model, plenty of assumptions can be derived by these values. During the runtime, Tensorflow uses the predicted tensor values and the pre-classified tensor values to calculate the metrics. The calculation can be realized, as shown in Figure 6.1. With tensor multiplication, the number of true positives can be measured. Similar to that code snippet, the other values can be calculated.

```python
def true_positive(self, y_true: tf.Tensor, y_pred: tf.Tensor) -> tf.Tensor:
    """Calculates the number of true positives.

    Args:
        y_true (tf.Tensor): Correct labels which are passed for training/testing.
        y_pred (tf.Tensor): Predicted labels which are returned by a classifier.

    Returns:
        tf.Tensor: The number of true positives between element of y_true
              and y_pred.
    """

    y_pred_rounded = tf.math.round(y_pred)

    return tf.math.count_nonzero(y_pred_rounded * y_true,
                                 axis=None,
                                 dtype=tf.float64)
```

Figure 6.1: Calculation of true positives in Tensorflow.

In the upcoming metric functions, these base values will be processed.

Accuracy

The accuracy represents the ratio of correctly predicted values to all predicted samples. This can be either calculated for a certain class or for a subset set, which will be defined below in this section [22]. This value can be calculated as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{6.2}$$

Precision

The precision shows the ratio of true positive predicted values to all positive predicted values. This incicates the reliability of the positive predicted values [78]. The precision formula is defined as:

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

Recall or Sensitivity

The recall or sensitivity shows the ratio of true positive predicted values to all values that should be predicted as positive. This indicates the reliability of all positive predicted values [78]. The recall can be measured by this calculation:

$$Recall = \frac{TP}{TP + FN} \tag{6.4}$$

F1 Score

The F1 score calculates the harmonic mean of precision and recall [66].

$$F1\_score = \frac{2 * precission * recall}{precission + recall} \tag{6.5}$$

Specificity

The specificity shows the ratio of true negative predicted values to all values that should be predicted as negative. This indicates the reliability of all predicted negative values [78]. The equation for specificity is defined as follows:

$$Specificity = \frac{TN}{TN + FP} \tag{6.6}$$

Area under the Curve (AUC)

The area under the curve (AUC) calculates the probability that the model predicts the correct value for a random sample. If this is the case, the AUC value is close to 100% [78]. In detail, this value can be calculated using this formula:

$$AUC = \frac{sensitivity + specificity}{2} \tag{6.7}$$

Hamming Loss

The hamming loss is the probability of false predicted values in ratio to all predicted samples [66].

$$Hamming\_Loss = \frac{FP + FN}{TP + TN + FP + FN} \tag{6.8}$$

Jaccard Similarity

The jaccard similarity ignores the correctly predicted false positives focussing on the reliability of positive predicted values [78]. The jaccard similarity can be measured as follows:

$$Jaccard_similarity = \frac{TP}{TP + FP + FN} \tag{6.9}$$

Subset Accuracy

The subset accuracy calculates the accuracy of the whole output in the case of multi-label models [66]. Therefore, it is defined as:

$$Subset\_Accuracy = \frac{subset\_true\_prediction}{subset\_true\_prediction + subset\_false\_prediction} \tag{6.10}$$

6.2 Evaluation Results

Since the metrics have been defined in the previous section, we can now check how the developed AI scanner performs. Before we look at actual statistics, the system specifications are listed in Table 6.1.

System component	**Component version**
Operating system	Windows 10
CPU	Intel Core i7-9700K
Memory	16GB
GPU	NVIDIA GeForce RTX 2070 SUPER
GPU memory	8192 MB GDDR6
CUDA	v10.1.105
CUDA cores	2560
Python	v3.7.7
Tensorflow	v2.1.0

Table 6.1: System environment specification used to run the deep learning process.

All learning processes are triggered the same way, passing the dataset in the exact same order. The only parameters changed for each learning process are, on the one hand, the model architecture and, on the other hand, the configuration of iterations and epochs.

If we take a look at Table 6.2, we see that all three approaches end quite promising. During the test, each model is capable of detecting the smart contracts' vulnerability types with at least 97.2% accuracy. However, it is interesting to see that the LSTM model is outperformed by the simpler GRU model. This might be the case since the calculatable parameters could not be adjusted in time. Besides that, the difference between the single-output-layer model and the multi-output-layer model is almost non-existent. This might change if the number of vulnerability classes is increased.

After comparing the different models, we investigate if the performance can be boosted even more by increasing the number of iterations and epochs. For this purpose, the same model will always be used; in this case, the MOL GRU model. The resulting metrics can be investigated in Table 6.3. As we can see, the metrics can slightly be improved by simply iterating over the dataset again. It makes sense that the train averages improve since the model learns more details of the dataset with repeated training. However, the test metrics on the unknown data are improved as well. For instance, the accuracy is improved by 0.7% and the subset accuracy even by 3% when comparing the first and the last column. It can also be investigated that the experiment with one iteration and repeated epochs is slightly more effective when compared with the run of repeated iterations and one epoch.

Metric	SOL GRU	SOL LSTM	MOL GRU
Average train loss	0.189	0.230	0.189
Average train accuracy	0.933	0.913	0.934
Average train precision	0.819	0.753	0.819
Average train recall	0.700	0.635	0.702
Average train F1	0.745	0.676	0.746
Average train AUC	0.844	0.806	0.845
Average train subset-accuracy	0.672	0.582	0.676
Average train hamming-loss	0.066	0.086	0.065
Average train jaccard-similarity	0.663	0.582	0.666
Average test loss	0.090	0.106	0.090
Average test accuracy	0.974	0.972	0.974
Average test precision	0.956	0.941	0.964
Average test recall	0.905	0.907	0.900
Average test F1	0.930	0.924	0.931
Average test AUC	0.948	0.947	0.946
Average test subset-accuracy	0.878	0.875	0.875
Average test hamming-loss	0.025	0.027	0.025
Average test jaccard-similarity	0.870	0.859	0.871
Training time in hours	1:38:09	1:34:34	1:38:13
Total learning in hours	2:04:37	2:04:05	2:02:11
Average prediction time in minutes	00:00.682	00:00.752	00:00.800

Table 6.2: Comparison of different model architecture after learning one iteration and one epoch.

Metric	1 it. 1 ep.	1 it. 3 ep.	3 it. 1 ep.	3 it. 3 ep.
Average train loss	0.189	0.103	0.107	0.067
Average train accuracy	0.934	0.965	0.964	0.976
Average train precision	0.819	0.929	0.927	0.963
Average train recall	0.702	0.845	0.846	0.898
Average train F1	0.746	0.881	0.881	0.926
Average train AUC	0.845	0.919	0.919	0.946
Average train subset-accuracy	0.676	0.826	0.825	0.879
Average train hamming-loss	0.065	0.0344	0.035	0.023
Average train jaccard-similarity	0.666	0.823	0.821	0.877
Average test loss	0.090	0.058	0.067	0.050
Average test accuracy	0.974	0.980	0.978	0.981
Average test precision	0.964	0.975	0.972	0.970
Average test recall	0.900	0.919	0.912	0.928
Average test F1	0.931	0.946	0.941	0.949
Average test AUC	0.946	0.957	0.953	0.961
Average test subset-accuracy	0.875	0.899	0.892	0.905
Average test hamming-loss	0.025	0.019	0.021	0.018
Average test jaccard-similarity	0.871	0.898	0.889	0.903
Training time in hours	1:38:13	4:55:44	5:04:09	14:46:18
Total learning in hours	2:02:11	5:21:03	6:10:06	15:53:01
Average prediction time in minutes	00:00.682	00:00.762	00:00.771	00:00.863

Table 6.3: Comparison of the MOL-GRU model performance with different iteration and epoch configuration.

So far, we only investigated the model metrics according to the overall result. It would be interesting if all vulnerability classes perform the same way. Table 6.4 shows the class-specific metrics retrieved by the MOL-GRU model. As a reminder, these eight classification types are as follows:

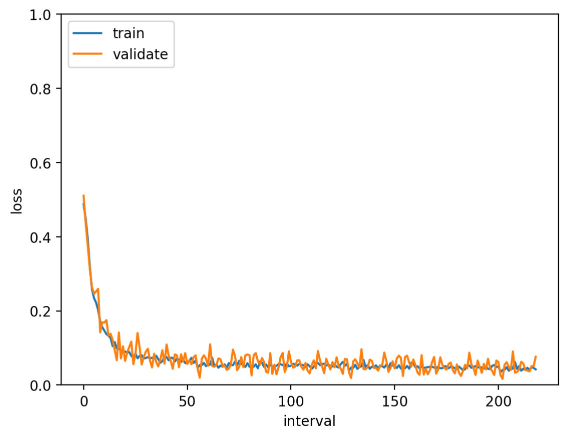
1. Callstack
2. Reentrancy
3. Money concurrency
4. Accessible self-destruct
5. DoS (Unbounded Operation)
6. Tainted self-destruct
7. Assert violation
8. Multiple sends

The model predicts the correct values in at least 97% of the test samples independent of the vulnerability class. The reentrancy bug and Tainted self-destruct are even detected with the probability of 99%. In the case of the Callstack and the Multiple sends bug, the recall is relatively low. This indicates that this vulnerability is more often wrongly detected. It is also remarkable that the DoS (Unbounded Operation) might be often predicted wrongly since the Jaccard-similarity value with 84% is noticeably low. All other values are in a relatively similar range.

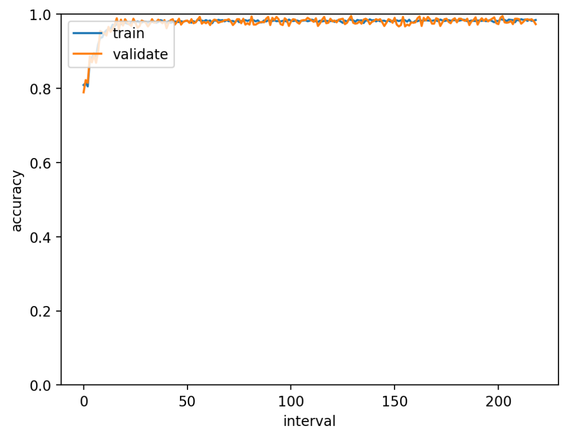
Metric	**cl. 1**	**cl. 2**	**cl. 3**	**cl. 4**	**cl. 5**	**cl. 6**	**cl. 7**	**cl. 8**
Average test loss	0.053	0.034	0.076	0.052	0.063	0.010	0.047	0.065
Average test accuracy	0.979	0.991	0.970	0.981	0.970	0.997	0.977	0.980
Average test precision	0.983	0.990	0.967	0.973	0.925	0.994	0.949	0.995
Average test recall	0.886	0.953	0.940	0.925	0.902	0.991	0.934	0.886
Average test F1	0.932	0.971	0.953	0.948	0.913	0.993	0.941	0.937
Average test AUC	0.941	0.975	0.962	0.959	0.943	0.995	0.961	0.942
Average test hamming-loss	0.020	0.008	0.029	0.018	0.029	0.002	0.022	0.019
Average test jaccard-similarity	0.873	0.944	0.912	0.902	0.841	0.986	0.890	0.882

Table 6.4: Class-specific metrics retrieved by the MOL-GRU model training.

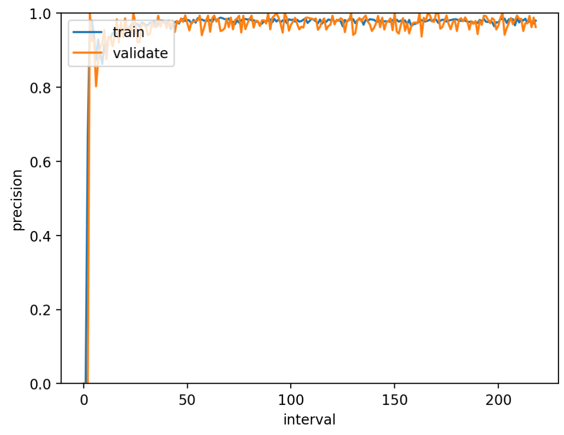
After having a look at the averaged numbers of the model metrics, it is also useful to check the progress of each statistic. In Figures 6.2 and 6.3 all the metrics' progress can be observed. These metrics show the progress of the MOL-GRU model, which was trained for three iterations and three epochs per chunk. It can be found out that the validation metrics vary a lot, which indicates that it performs a good job of preventing overfitting. Besides that, it can be observed that different metrics converge earlier than others. Some values, such as the loss, converge approximately in interval 50, whereas most other values converge in interval 20. This explains why all the models work very efficiently, even with much shorter training time.



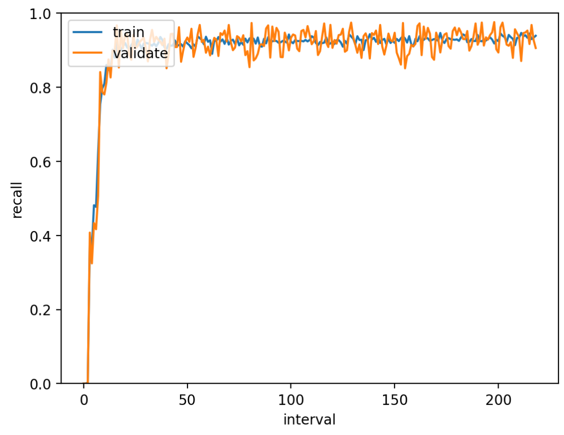
(a) Training and validation loss



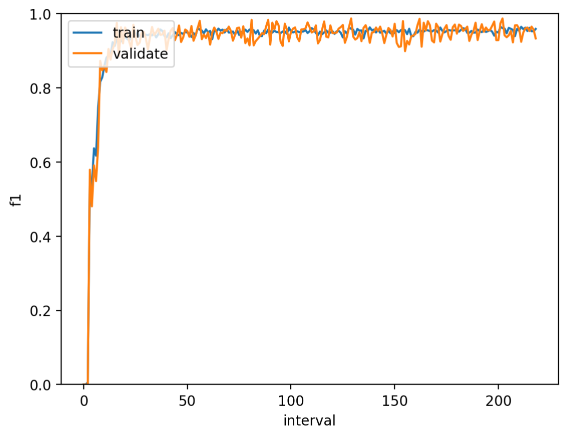
(b) Training and validation accuracy



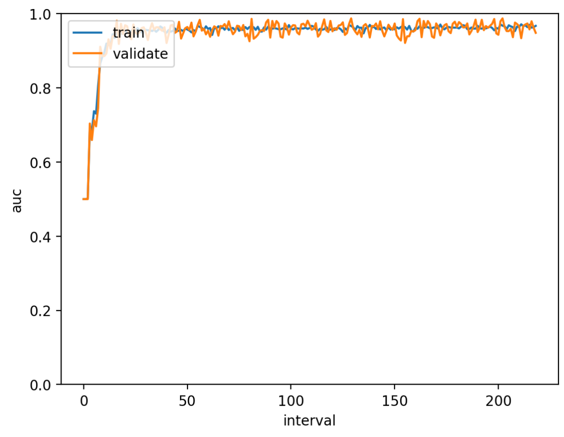
(c) Training and validation precision



(d) Training and validation recall

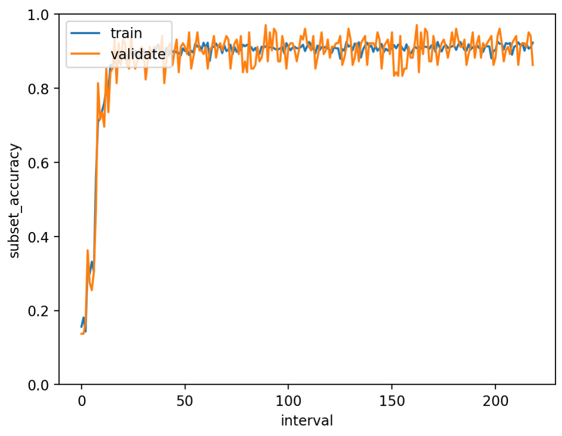


(e) Training and validation F1 score

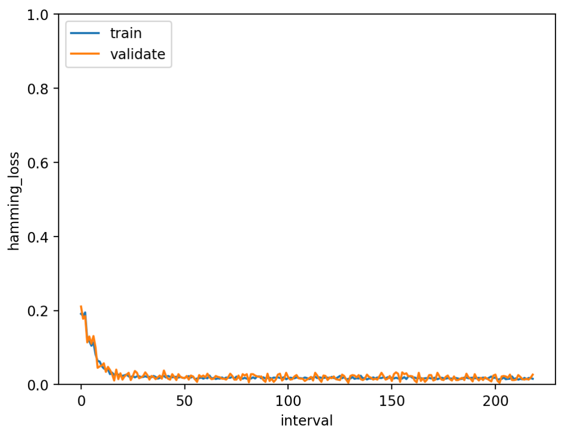


(f) Training and validation AUC

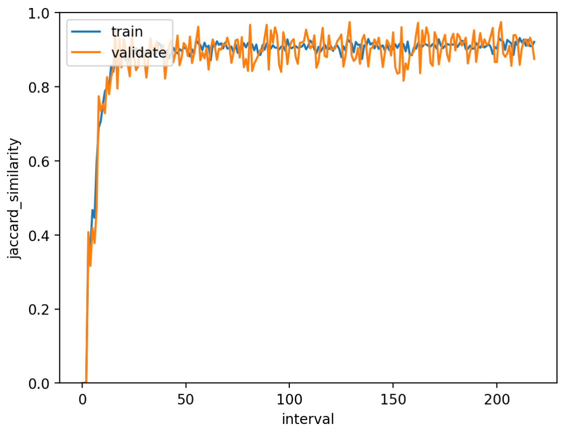
Figure 6.2: Metric progress during model training - part 1.



(a) Training and validation subset-accuracy



(b) Training and validation hamming-loss



(c) Training and validation jaccard similarity

Figure 6.3: Metric progress during model training - part 2.

6.3 Comparison with other machine-learning approaches

After developing our own AI scanner, it is time to compare the achieved result with the machine-learning approaches examined in Section 3. First of all, it needs first to be noted that the approaches cannot be compared exactly since each classifier work with other techniques and other complexities.

The ContractWard tool receives source-code as input, which needs to be compiled and translated to be processed by the classifier. With this classifier, six vulnerability types can be detected. They utilize the power of parallel binary classifiers in order to create expert models. The prediction time, including the preprocessing, takes approximately four seconds. As indicated in Tables 6.2 and 6.3, the average prediction time stays under one second. With the expert classifiers, the ContractWard could achieve slightly better performance results [48]. However, the approach with neural networks might be a lot easier to scale so that it is possible to detect even more smart contracts' vulnerability types.

The LSTM approach can be compared a lot easier since both approaches use kind of the same technique. In contrast to the LSTM binary approach, only classifying contracts into vulnerable and invulnerable, our AI Scanner is capable of detecting exact vulnerability types. In addition to that, eight instead of three vulnerability classes are learned by our model. With this work, it is also shown that it is possible to mix up the result with different tools. Although the accuracy of the LSTM approach is nearly perfect, the other metrics are topped by our approach. This includes the recall, precision, F1, and AUC score [54].

7. Conclusion

The threat of smart contract exploitation requires mitigation strategies because this technology comes with some flaws and bugs. It is a significant task to detect vulnerabilities of smart contracts before deploying them to the public blockchain because of the difficulty of removing buggy code. With this work, a framework is developed, which can be used to develop an AI scanner capable of detecting eight different vulnerability classes. Furthermore, it takes care of data retrieval, pre-classification, and model building. On the basis of publicly available bytecode of smart contracts in combination with publicly available tools, this project has been realized. This way, the accuracy of 98% could be achieved with common text classification technologies. In a deployed state, developers can now classify bytecode of smart contracts. Furthermore, this approach looks promising that it can be extended by more classification types. This way, this tool could turn out to be very powerful.

Since multiple classification tools have already been mixed, this work can be extended by the addition of new vulnerability types. These new types can either be the less represented ones from this project or others provided by new classifiers that can work based on bytecode. The increased number of classification types may also lead to deeper neural network architectures, improving the overall result. Data augmentation might also be a promising technique to support less represented classification classes. Since this project is based on the 1.2 million smart contracts derived from the first five million blocks, the dataset can also be increased to improve and fine-tune the model performance. Another interesting topic in that area is the localization of vulnerabilities in smart contracts. Since the deep-learning algorithm learns specific patterns that are the base for decision making, it could be possible to create a mechanism to highlight suspicious areas.

ACKNOWLEDGMENTS

I would like to say thank you to my supervisor, Prof. Dr.-Ing. Alexandra Dmitrienko, for her support and guidance throughout this project. Furthermore, I would also like to thank Huili Chen and Hossein Fereidooni for supporting me, especially in the area of deep-learning.

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence

AST Abstract Syntax Tree

AUC Area under the Curve

API Application Programming Interface

BOW Bag of Words

CPU Central Processing Unit

CRUD Create, Read, Update and Delete

CSV Comma-Separated Values (file format)

EVM Ethereum Virtual Machine

FN False Negative

FP False Positive

GPU Graphics Processing Unit

GRU Gated Recurrent Units

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IR Intermediate Representations

JSON JavaScript Object Notation (file format)

KNN k-Nearest Neighbor

LSTM Long Short-Term Memory

MLP Multilayer Perceptrons

MOL Multiple-Output-Layer

NLP Natural Language Processing

RF Random Forest

RNN Recurrent Neural Networks

RPC Remote procedure call

SMT Satisfiability Modulo Theories

SOL Single-Output-Layer

SVM Support Vector Machine

TN	True Negative
TP	True Positive
UI	User Interface
URI	Uniform Resource Identifier

Bibliography

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." [Online]. Available: http://bitcoin.org/bitcoin.pdf

[2] D. Siegel, "Understanding the dao attack," Jun 2016. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists

[3] M. d. Castillo, "The dao attacked: Code issue leads to $60 million ether theft," Jun 2016. [Online]. Available: https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft

[4] A. Dika and M. Nowostawski, "Security vulnerabilities in ethereum smart contracts," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, July 2018, pp. 955–962.

[5] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: foundations, design landscape and research directions," *CoRR*, vol. abs/1608.00771, 2016. [Online]. Available: http://arxiv.org/abs/1608.00771

[6] "Ethereum Whitepaper." [Online]. Available: https://ethereum.org

[7] A. Savelyev, "Contract law 2.0: 'smart' contracts as the beginning of the end of classic contract law," *Information & Communications Technology Law*, vol. 26, no. 2, pp. 116–134, 2017. [Online]. Available: https://doi.org/10.1080/13600834.2017.1301036

[8] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *Financial Cryptography and Data Security*, J. Clark, S. Meiklejohn, P. Y. Ryan, D. Wallach, M. Brenner, and K. Rohloff, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 79–94.

[9] "ethereum/solidity," Oct. 2020, original-date: 2015-08-17T12:27:26Z. [Online]. Available: https://github.com/ethereum/solidity

[10] "Introduction to smart contracts." [Online]. Available: https://ethereum.org

[11] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 254–269. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978309

[12] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: foundations, design landscape and research directions," *CoRR*, vol. abs/1608.00771, 2016. [Online]. Available: http://arxiv.org/abs/1608.00771

[13] "Ethereum in BigQuery: How we built this dataset." [Online]. Available: https://cloud.google.com/blog/

[14] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: Finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 65–68. [Online]. Available: http://doi.acm.org/10.1145/3183440.3183495

[15] D. Siegel, "Understanding the dao attack," Jun 2016. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists

[16] M. d. Castillo, "The dao attacked: Code issue leads to $60 million ether theft," Jun 2016. [Online]. Available: https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft

[17] P. Sodhi, N. Awasthi, and V. Sharma, "Introduction to machine learning and its basic application in python," in *Proceedings of 10th International Conference on Digital Strategies for Organizational Success*, January 2019. [Online]. Available: http://dx.doi.org/10.2139/ssrn.3323796

[18] G. Dong and H. Liu, *Feature engineering for machine learning and data analytics*. CRC Press, 2018.

[19] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. [Online]. Available: https://doi.org/10.1038/nature14539

[20] T. Dietterich, "Overfitting and undercomputing in machine learning," *ACM Comput. Surv.*, vol. 27, no. 3, p. 326–327, Sep. 1995. [Online]. Available: https://doi.org/10.1145/212094.212114

[21] O. Dekel and O. Shamir, "Multiclass-multilabel classification with more classes than examples," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010, pp. 137–144.

[22] "Machine Learning Glossary." [Online]. Available: https://developers.google.com/machine-learning/glossary

[23] O. Dekel and O. Shamir, "Multiclass-multilabel classification with more classes than examples," *Journal of Machine Learning Research - Proceedings Track*, vol. 9, pp. 137–144, 01 2010.

[24] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification," in *Twenty-ninth AAAI conference on artificial intelligence*, 2015.

[25] S. Minaee, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlu, and J. Gao, "Deep learning based text classification: A comprehensive review," 2020.

[26] R. Dey and F. M. Salem, "Gate-variants of gated recurrent unit (gru) neural networks," in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2017, pp. 1597–1600.

[27] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 317–331.

[28] J. Cai, S. Yang, J. Men, and J. He, "Automatic software vulnerability detection based on guided deep fuzzing," in *2014 IEEE 5th International Conference on Software Engineering and Service Science*, June 2014, pp. 231–234.

[29] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework For Smart Contracts," *arXiv e-prints*, p. arXiv:1908.09878, Aug 2019.

[30] "crytic/slither," Oct. 2020, original-date: 2018-09-05T21:56:35Z. [Online]. Available: https://github.com/crytic/slither

[31] "melonproject/oyente," Oct. 2020, original-date: 2017-03-17T14:42:38Z. [Online]. Available: https://github.com/melonproject/oyente

[32] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," 2019.

[33] "trailofbits/manticore," Oct. 2020, original-date: 2017-02-10T22:28:34Z. [Online]. Available: https://github.com/trailofbits/manticore

[34] "ConsenSys/mythril," Oct. 2020, original-date: 2017-09-18T04:14:20Z. [Online]. Available: https://github.com/ConsenSys/mythril

[35] L. Alt and C. Reitwiessner, "Smt-based verification of solidity smart contracts," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2018, pp. 376–388.

[36] "Ethereum Contract Library by Dedaub." [Online]. Available: https://contract-library.com/

[37] "Smart Contract Audits." [Online]. Available: https://www.dedaub.com/

[38] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276486

[39] N. Grech, "nevillegrech/MadMax," Oct. 2020, original-date: 2018-09-01T13:25:32Z. [Online]. Available: https://github.com/nevillegrech/MadMax

[40] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[41] "eth-sri/securify," Sep. 2020, original-date: 2018-09-13T10:33:15Z. [Online]. Available: https://github.com/eth-sri/securify

[42] "MythX: Smart contract security service for Ethereum." [Online]. Available: https://mythx.io/

[43] B. M. M. i. t. co founder and C. H. O. o. MythX, "MythX Tech: Behind the Scenes of Smart Contract Security Analysis," Dec. 2019. [Online]. Available: https://blog.mythx.io/features/mythx-tech-behind-the-scenes-of-smart-contract-analysis/

[44] G. Pace and J. Ellul, "Runtime verification of ethereum smart contracts," 09 2018.

[45] gordonpace, "gordonpace/contractLarva," May 2020, original-date: 2017-12-14T19:27:41Z. [Online]. Available: https://github.com/gordonpace/contractLarva

[46] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," 2018.

[47] I. Nikolic, "ivicanikolicsg/MAIAN," Oct. 2020, original-date: 2018-03-12T07:58:25Z. [Online]. Available: https://github.com/ivicanikolicsg/MAIAN

[48] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, pp. 1–1, 2020.

[49] T. Chen and C. Guestrin, "Xgboost," *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug 2016. [Online]. Available: http://dx.doi.org/10.1145/2939672.2939785

[50] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119 – 139, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002200009791504X

[51] L. Breiman, "Machine learning, volume 45, number 1 - springerlink," *Machine Learning*, vol. 45, pp. 5–32, 10 2001.

[52] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, vol. 9, no. 3, pp. 293–300, 1999.

[53] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inf. Theory*, vol. 13, pp. 21–27, 1967.

[54] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, "Towards safer smart contracts: A sequence learning approach to detecting security threats," 2019.

[55] "Introduction | ML Universal Guides." [Online]. Available: https://developers.google.com/machine-learning/guides/text-classification

[56] "Machine Learning Process And Scenarios," May 2017. [Online]. Available: https://elearningindustry.com/machine-learning-process-and-scenarios

[57] "pandas - Python Data Analysis Library." [Online]. Available: https://pandas.pydata.org/

[58] "NumPy." [Online]. Available: https://numpy.org/

[59] "argparse — Parser for command-line options, arguments and sub-commands — Python 3.9.0 documentation." [Online]. Available: https://docs.python.org/3/library/argparse.html

[60] "pytest: helps you write better programs — pytest documentation." [Online]. Available: https://docs.pytest.org/en/stable/

[61] Docker, "Empowering app development for developers." [Online]. Available: https://www.docker.com/

[62] C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2723872.2723882

[63] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, 03 2014.

[64] "MySQL." [Online]. Available: https://www.mysql.com/

[65] p. contributors, "phpMyAdmin." [Online]. Available: https://www.phpmyadmin.net/

[66] "TensorFlow." [Online]. Available: https://www.tensorflow.org/

[67] "tensorflow/tensorflow," Oct. 2020, original-date: 2015-11-07T01:19:20Z. [Online]. Available: https://github.com/tensorflow/tensorflow

[68] “Keras: the Python deep learning API.” [Online]. Available: https://keras.io/

[69] I. L. Tom Hope, Yehezkel S. Resheff, in *Learning TensorFlow.* O’Reilly Media, Inc., August 2017.

[70] “blockchain-etl/ethereum-etl,” Aug. 2020. [Online]. Available: https://github.com/blockchain-etl/ethereum-etl

[71] “Ethereum API | IPFS API Gateway | ETH Nodes as a Service.” [Online]. Available: https://infura.io/

[72] “Ethereum Contract Library by Dedaub.” [Online]. Available: https://contract-library.com/

[73] Ethereum, “ethereum/eips,” Mar 2019. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

[74] “Introduction — Web3.py 5.12.1 documentation.” [Online]. Available: https://web3py.readthedocs.io/en/stable/

[75] “Overview · Smart Contract Weakness Classification and Test Cases.” [Online]. Available: http://swcregistry.io/

[76] “Flask API.” [Online]. Available: https://www.flaskapi.org/

[77] S. Jadon, “A survey of loss functions for semantic segmentation,” 06 2020.

[78] D. Powers, “Evaluation: From precision, recall and f-factor to roc, informedness, markedness & correlation,” 2008.