cryptography

Article

Detecting Smart Contract Vulnerabilities with Combined Binary and Multiclass Classification

Anzhelika Mezina [1] and Aleksandr Ometov [2,*]

[1] Department of Telecommunications, Faculty of Electrical Engineering and Communications, Brno University of Technology, Technicka 12, 616 00 Brno, Czech Republic; xmezin00@vut.cz
[2] Electrical Engineering Unit, Faculty of Information Technology and Communication Sciences, Tampere University, 33720 Tampere, Finland
* Correspondence: aleksandr.ometov@tuni.fi

Abstract: The development of Distributed Ledger Technology (DLT) is pushing toward automating decentralized data exchange processes. One of the key components of this evolutionary step is facilitating smart contracts that, in turn, come with several additional vulnerabilities. Despite the existing tools for analyzing smart contracts, keeping these systems running and preserving performance while maintaining a decent level of security in a constantly increasing number of contracts becomes challenging. Machine Learning (ML) methods could be utilized for analyzing and detecting vulnerabilities in DLTs. This work proposes a new ML-based two-phase approach for the detection and classification of vulnerabilities in smart contracts. Firstly, the system's operation is set up to filter the valid contracts. Secondly, it focuses on detecting a vulnerability type, if any. In contrast to existing approaches in this field of research, our algorithm is more focused on vulnerable contracts, which allows to save time and computing resources in the production environment. According to the results, it is possible to detect vulnerability types with an accuracy of 0.9921, F1 score of 0.9902, precision of 0.9883, and recall of 0.9921 within reasonable execution time, which could be suitable for integrating existing DLTs.

Keywords: modeling; classification; vulnerability detection; distributed systems

Citation: Mezina, A.; Ometov, A. Detecting Smart Contract Vulnerabilities with Combined Binary and Multiclass Classification. *Cryptography* **2023**, *7*, 34. https://doi.org/10.3390/cryptography7030034

Academic Editor: Josef Pieprzyk

Received: 9 May 2023
Revised: 24 June 2023
Accepted: 5 July 2023
Published: 7 July 2023

1. Introduction

With the rapid development of distributed systems, smart contracts have become one of the targets of vulnerability searches from both sides of the information security barricade [1]. Essentially, a smart contract is an automatically executed transaction protocol intended to manage, control, or document events according to the terms of a contract [2]. Because of the autonomy of the smart contract operation, it is almost impossible to reverse the transaction in case of a successful attack on a contract in public systems [3]. Simultaneously, attacks on smart contracts exist and cause serious damage to existing systems. As a few historical examples, USD 70 M was stolen in a DAO attack [4] followed by USD 300 M being blocked because of the MultiSig wallet of the company Parity [5]. Both attacks succeeded because of vulnerabilities in the execution of smart contracts.

Naturally, some tools for checking smart contracts' safety already exist, and many researchers have put their careful attention into developing others [6–8]. Some use symbolic execution technology, and other instruments are based on predefined patterns. Also, there are tools based on data processing tools for finding and classifying vulnerable smart contracts.

Nonetheless, DLT systems are aimed at data immutability. Additionally, such systems can produce a lot of side transaction data during operation. Notably, one of the most promising tools to proactively react to the changes in and misbehavior of information exchange is related to ML algorithms. They could be used to operate over with this information [9]. Considering that it is much more difficult to control distributed systems' processes than centralized systems, ML can assist in managing them in a (semi-)automatic manner. With ML, it is possible to effectively predict the state of the distributed system, find vulnerabilities, and explore attacks on the fly.

There are several reasons and perspectives for using ML for the smart contracts analysis:

1. The research on and necessity of solutions, which can be applied in the real world, is an actual problem.
2. The possible solution demonstrates a general way to apply ML to increase distributed systems' qualitative metrics.
3. Ethereum is one of the most popular platforms for creating decentralized applications, which is why the proposed solution will have practical value and can be compared with existing approaches for smart contract analysis in Ethereum. The quality dataset Ethereum [10] for designing and testing ML solutions is publicly available.

Based on the above motivations, the main goal of this work is to develop a novel ML-based framework, which predicts if the smart contract is vulnerable or not, and, if vulnerable, classifies the vulnerabilities as one of the following types: suicidal, prodigal, greedy, and suicidal prodigal contracts.

This paper's contributions are as follows:

- We designed a two-phase ML-based framework that can detect vulnerable contracts and determine the type of vulnerability.
- Our system focuses on the vulnerable contracts by filtering the non-vulnerable ones, which can potentially save time and reduce computational load.
- After comparison with a single-phase classification, the proposed system has fewer false-negative detection samples.
- We have proved that instead of the traditional method of application, either multiclass or binary classification, it is possible to achieve better results using the two phases consequently.

The rest of the paper is organized as follows: First, Section 2 describes the existing solutions and recent approaches based on ML. Next, Section 3 provides a description of the research methodology and scenarios. Then, the results are presented and discussed in Section 4, while Section 5 highlights future aspects, challenges, and the integration of aspects of the ML-based detection systems. The last Section 6 concludes the paper.

2. Related Work

This section outlines the state of the art of present tools used for the analysis of smart contracts with a focus on ML applications, as well as highlights the research gap in this domain.

2.1. Tools for Analysis of Smart Contracts

Several tools use symbolic analysis, representing the values of program variables as symbolic expressions of the symbolic input values [11]. One of the earliest tools for detecting vulnerabilities in smart contracts, which utilizes a symbolic execution technology, is OYENTE [11]. It is based on the Control Flow Graph (CFG) [12]. The serious drawback of this tool is that confirmations of flagged contracts are only performed manually. Similarly, it can produce false-positive results due to imprecise modeling of domain-specific elements [13]. Finally, the instrument can only achieve sufficient code coverage on realistic contracts [9].

The Mythril tool [14] can distinguish between the following vulnerabilities: integer overflow/underflow, re-entrancy vulnerability, delegate call to untrusted callee, unprotected self-destruct instruction, authorization through tx.origin, and assert violation. One of the advantages of this tool is the ability to work without access to the source code of smart contracts. On the other hand, there is a limitation regarding invocation depth, as for all symbolic execution-based tools, i.e., vulnerability cannot be found because of the balance between analysis speed and depth.

The Securify solution is proposed in [15]. It is based on compliance pattern analysis. Its pipeline contains two steps. Firstly, it analyzes the contract's dependency graph and extracts semantic information from the source code. Secondly, it verifies compliance and violation patterns [15]. Compared to OYENTE and Mythril, Securify performs better in terms of various metrics and, overall, checks more vulnerabilities. The drawbacks of the presented tool are not many. It operates under the assumption that all instructions in the contract are reachable. Also, it cannot reason about numerical properties. Finally, it requires expert knowledge to generate predefined patterns of vulnerability.

Nikolic et al. [16] proposed a tool called MAIAN, which labels vulnerable smart contracts as suicidal, prodigal, or greedy. The proposed solution specifies trace properties, which employ interprocedural symbolic analysis and concrete validation for exhibiting real exploits. In general, the tool achieved an 89% true-positive rate. Despite the good results (around 90% plus detection probability), the system has other advantages. The main one is that it is possible to check contracts for bugs with MAIAN by running the contract. Also, the tool does not use the contract's source code for the analysis. Another problem is that the validation of contracts can only be performed by MAIAN either on flagged contracts that are alive within the forked Ethereum chain or on contracts with an existing source code available.

2.2. *ML for Smart Contract Analysis*

The authors of [3] propose a so-called sequence learning approach for detecting vulnerable smart contracts. They use deep learning algorithms to classify contracts as weak/not vulnerable and develop Long-Short Term Memory (LSTM) neural networks. The developed model shows a detection test accuracy of 99.57% and an F1 score of 86.04%. Notably, it detected up to 92.86% of false-positive contracts classified by MAIAN. Similarly to MAIAN, the proposed model does not need access to source codes. In addition, like any ML model, it can constantly train on new contracts, thereby increasing its quality. An important disadvantage of this model is that it classifies models as vulnerable and non-vulnerable. Still, it does not give any information about the class of vulnerability. Moreover, the model analyses the sequence of opcodes; however, by learning opcode sequences it cannot consider the control flow of a smart contract, so the model can not find control flow vulnerabilities.

Another work [17] proposes the slice matrix as a new feature for detecting vulnerable contracts. Additionally, the authors experimented with Feedforward Neural Network (FNN), Convolutional Neural Network (CNN), and Random Forest (RF). The training process was independently conducted over three different vulnerabilities: "has short address", "is greedy", and "has flows". The results show that RF performs better than FNN and CNN and that using NN with a slice matrix gives better results than using NN and opcode features. However, applying the slice matrix feature with RF was not possible, and its advantages still need to be fully explored.

Since CNN is usually used for image processing, transforming 1D data into 2D with the following application of NNs is also a popular method of data processing. This method was applied in [18]. The authors compiled the smart contract source code, transformed bytecodes into code with a fixed size, mapped it to a contract-based image, and the proposed CodeNet was trained on those prepared images. The model was trained on four vulnerabilities. According to the results, this model performed better than well-known tools, such as Mythril, Oyente, etc., with an accuracy of 98.79%.

The Graph Neural Network is proposed to be used for vulnerability detection in [19]. The approach consists of three phases: extracting security patterns from the source code, constructing a contract graph and performing normalization, and vulnerability detection. The proposed method was compared with state-of-the-art techniques, and achieved the following accuracies: reentrancy—89.15%, timestamp dependence—89.02%, and infinite loop vulnerabilities—83.21%.

Another approach uses LSTM, an Artificial Neural Network (ANN), and a Gated Recurrent Unit model (GRU) to detect vulnerable smart contacts in the Internet of Things (IoT) environment [2]. The proposed framework consists of three parts: the deployment layer, the data preparation layer, and the prediction layer. The LSTM model achieved the best results: accuracy—0.99, precision—0.92, and recall—0.88.

The authors of [20] utilize the traditional ML methods. The paper proposes the use of extracted bigram features from simplified operation codes for training five ML algorithms and two sampling algorithms. The best model is XGBoost, which achieved Micro-F1 and Macro-F1 over 96%.

2.3. Summary

According to the literature review, the existing approaches have several drawbacks. Firstly, many works are aimed at determining whether the contract is vulnerable. However, just a few are focused on detecting vulnerability types and have a limited number of vulnerabilities. Secondly, ML-based models have space for improvement: not all have achieved high accuracy. Another interesting point is that many approaches utilize different types of NN, but the traditional ML methods need more attention.

In this work, we addressed the named limitations and proposed a two-phase system, which would determine not only if the contract was vulnerable, but also the type of vulnerability.

3. Methodology

The main goal of this work is to develop an ML-based framework that predicts if the contract is vulnerable or not and, if vulnerable, classifies the type of vulnerability.

One of the most important aspects of opcode analysis is reducing the number of false-negative predictions, i.e., to decrease the number of cases when the contract is classified as normal but, in reality, it is vulnerable. On the other hand, decreasing the false-positive rate is also important, which means the system classifies the contract as vulnerable when it is not. The developer needs to check it manually. Consequently, it takes time to analyze the code, especially when it needs to be correctly labeled. Considering the mentioned problems, this research aims to achieve high accuracy and F1 score and reduce the number of false-negative results.

We begin with the introduction of several scenarios. In the first scenario, the classification is performed in a single phase, and in the second one in two phases. Both of them utilize ML algorithms, including optimization techniques for the definition of optimal hyperparameters. Flow-wise, the dataset description is provided first, and the experiment details follow.

3.1. Dataset Description

The Ethereum dataset [3] was used to train and test the ML model. The dataset contains the following parameters: contract's address and opcode. An opcode is the sequence of numbers corresponding to operation types for execution. The selection of code can be explained by the fact that opcodes are very effective for the training of ML algorithms for the task of detection of malware.

The authors provided the initial discussion on opcodes in [3]. The authors used a dataset from Google BigQuery. They prepared the opcodes by parsing the contract's bytecode using the EVM instruction list. The labels were obtained using the MAIAN tool. The resulting dataset was cleaned to remove false-positive prodigal and suicidal contracts according to findings in [16]. A more detailed description regarding the dataset's creation is introduced in [3].

The contracts were categorized into 5 classes. The resulting dataset contains the following labels:

- *Normal* contracts where no vulnerability is detected.
- *Suicidal* contract that can be killed arbitrarily. In spite of the ability of some contracts to kill themselves in emergency situations, in the case of improper implementation, this ability can be exploited by any arbitrary account by executing the "suicide" instruction [16].
- *Prodigal contracts* refund the funds to owners, in the case of attacks, to the addresses that have previously sent Ether or that present the specific solution. The contract is considered vulnerable when a contract distributes to an arbitrary address which does not belong to the owner, has never made a deposit of Ether in the contract, and has not provided any data, which is difficult to forge [16].
- *Greedy contracts* remain alive and indefinitely lock Ether, allowing it to be released without any conditions. Straightforward errors can occur in contracts which accept Ether but either entirely lack instructions to transfer Ether out or these instructions are unreachable [16].
- *Suicidal and prodigal* contracts have appeared after the cleaning and processing phases of dataset creation according to the description in [3]. The contracts were flagged as both classes.

The dataset contains 892, 913 samples: normal—884, 273, greedy—5801, prodigal—1461, suicidal—1207, and suicidal and prodigal—171. The class distribution is depicted in Figure 1.

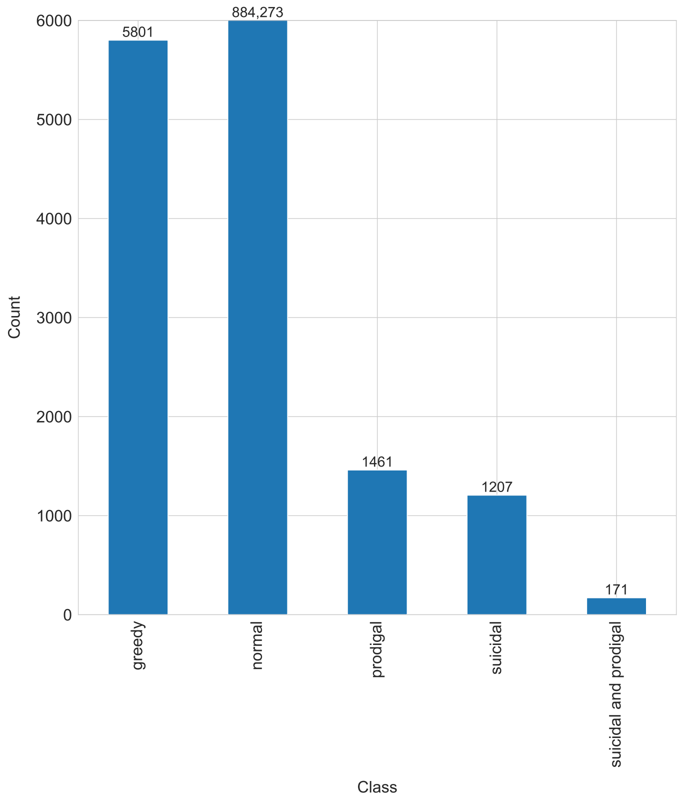


Figure 1. Class distribution in the selected dataset.

3.2. Description of ML Methods for Analysis

The following algorithms were used for the experiments:

- *Decision tree* predicts the target value using the sequence of simple rules.
- *k-Nearest Neighbors (k-NN)* assigns the class based on the most common class among neighbors.
- *Support Vector Machine (SVM)* transforms the classes into a higher dimensionality and searches the hyperplanes, which will separate the classes.

- *Random forest* is the decision trees ensemble.
- *Multilayer Perceptron (MLP)* is the perceptron where backpropagation is applied.
- *Logistic regression* is the statistical technique to find the relationships between independent variables and outcome values.

As an optimization technique, we used randomized search with 5-fold cross-validation. This technique is based on the random selection of hyper-parameters from a given distribution in a defined number of iterations; thus, the combination of hyperparameters for each algorithm was found in 50 iterations. On the one hand, this algorithm does not guarantee the best combination of hyperparameters, compared with Grid search [21], which tries every possible combination of parameters from a given search space. However, the Random Search algorithm still can outperform Grid search. Taking into consideration that the dataset is relatively large and the search space is relatively big, consequently, it will be time-consuming and computationally demanding. Random Search is the preferable optimization method in this case. The criterion for the selection of the best combination is the achieved accuracy on the testing set.

The search space for each of the ML algorithms is presented below:

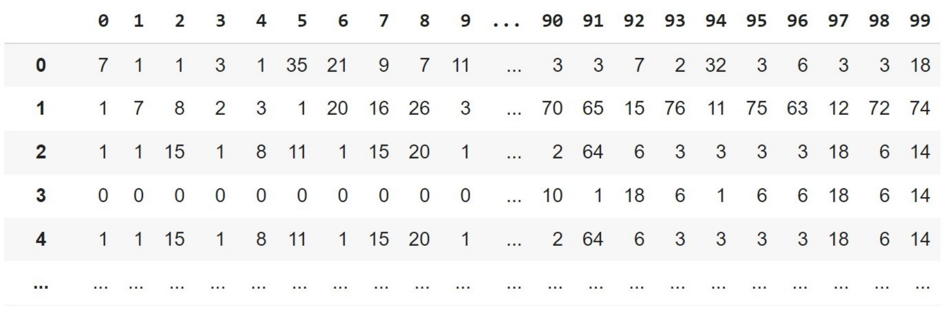
- Space for random forest:
  - Number or estimators: 1 to 10;
  - Number of features: 1 to 30;
  - Depth: 2 to 30;
  - Criterion: $gini$, $entropy$.
- Space for the logistic regression:
  - C: 0.0001 to 10;
  - Solver: $newton - cg$, $lbfgs$, sag, $saga$;
  - Iterations: 1 to 200.
- Space for the decision tree:
  - Max features: $auto$, $sqrt$, $log2$;
  - Depth: 10 to 30;
  - Criterion: $gini$, $entropy$.
- Space for MLP:
  - Solver: $lbfgs$, sgd, $adam$;
  - Hidden layer size: 100 to 250;
  - maximum iterations: 10 to 150.
- Space for k-NN:
  - Number of neighbors: 1 to 10;
  - Weights: $uniform$, $distance$;
  - Algorithm: $auto$, $ball\ tree$, $kd\ tree$, $brute$.
- Space for SVM:
  - 0.001 to 10 with number of spaced samples 200;
  - Kernel: $poly$, rbf.

3.3. Scenario 1

3.3.1. Data Preprocessing

As shown in Figure 1, the dataset is imbalanced, where 99% of samples are normal. The model will label all samples as normal and can easily achieve an accuracy of 99%. Because of that, the optimization technique, i.e., resampling, was applied to solve the imbalance problem. Each class has 14,286 samples for the multiclass problem. For the experiment, the dataset was divided into a training set (70%) and a testing set (30%).

Additionally, the opcodes were encoded with a tokenizer as the preparation step, which helps to represent the given opcodes as integer sequences. The final dataset's structure is depicted in Figure 2. Here, the initial opcodes are represented with 100 numbers, which makes it possible to process the data with the ML methods mentioned above.



	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93	94	95	96	97	98	99
0	7	1	1	3	1	35	21	9	7	11	...	3	3	7	2	32	3	6	3	3	18
1	1	7	8	2	3	1	20	16	26	3	...	70	65	15	76	11	75	63	12	72	74
2	1	1	15	1	8	11	1	15	20	1	...	2	64	6	3	3	3	3	18	6	14
3	0	0	0	0	0	0	0	0	0	0	...	10	1	18	6	1	6	6	18	6	14
4	1	1	15	1	8	11	1	15	20	1	...	2	64	6	3	3	3	3	18	6	14
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Figure 2. Training dataset structure.

3.3.2. Classification in One Step

The first scenario is relatively straightforward and has only three steps: data preprocessing, classification, and evaluation. The workflow is depicted in Figure 3.

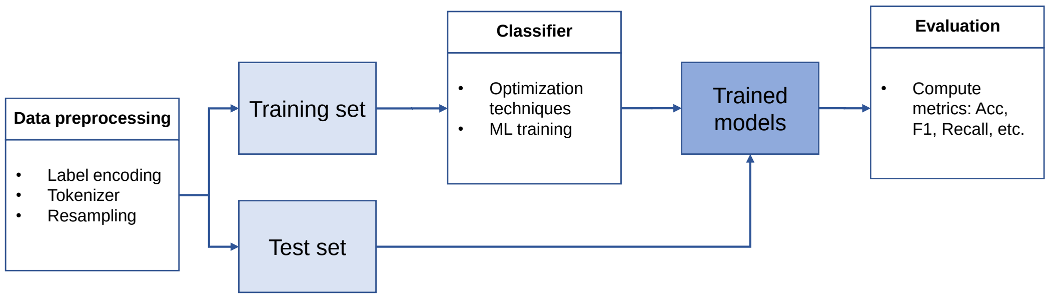


Figure 3. Contract classification executed in a single phase.

The preprocessing step is performed in the data preprocessing module. After that, the prepared data are classified into one of the 5 classes: normal, greedy, prodigal, suicidal, or suicidal and prodigal, using all mentioned ML methods.

To achieve the best results, the optimal hyperparameters for ML methods were found:

1. *Random forest:* number of estimators: 9; max features: 26; max depth: 21; criterion: *gini*;
2. *Logistic regression:* solver: $newton - cg$; max iterations: 178; C: 5.71;
3. *Decision tree:* max depth: 25; criterion: *entropy*; features: *sqrt*;
4. *MLP:* solver: *adam*; max iterations: 65; hidden layer sizes: 234;
5. *k-NN:* weights: *distance*; number of neighbours: 1; algorithm: *ball tree*;
6. *SVM:* kernel: rbf; C: 6.73;

3.4. Scenario 2

3.4.1. Data Preprocessing

In this part of the experiment, all classes of vulnerabilities were replaced with only one class "Vulnerable". Thus, the dataset was prepared for binary classification. The initial class distribution is depicted in Figure 4.

The second phase of this scenario needs samples which are only vulnerable and contain information on vulnerability type. Because of that, for this part, the dataset was prepared according to the mentioned requirements: the normal contracts were removed, and the rest of the samples kept the vulnerability type as a label.

The following data processing step is similar to the first scenario: the data are resampled and tokenized. After resampling, each class has 14,286 samples. The dataset was divided into training (70%) and testing (30%) sets.

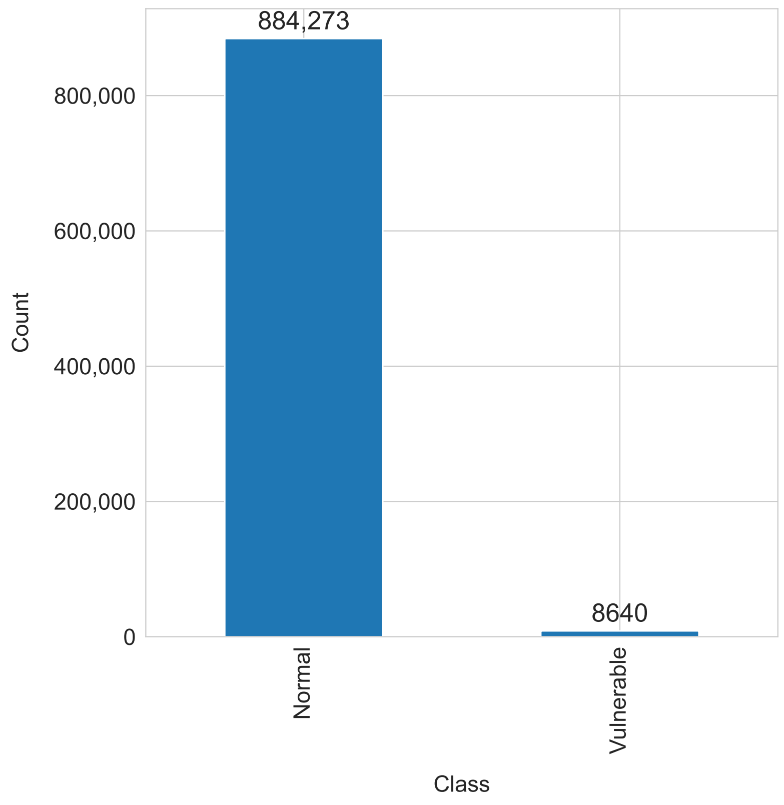


Figure 4. Class distribution for binary classification.

3.4.2. Classification in Two Phases

The second scenario has a more complex algorithm, introduced in Figure 5. The data are preprocessed in the data preprocessing module in the first phase. The next phase is performed with two classifiers: binary and multiclass. The first one is trained to distinguish between vulnerable and normal contracts. The second one performs the classification of vulnerability type: greedy, suicidal, prodigal, or suicidal and prodigal. This way, the two classifiers are prepared for the following part of the experiment, and with the evaluation metrics the best one is determined (more detailed results are introduced in Section 4). Optimization techniques were also utilized in this part of the experiment to find the best parameters.

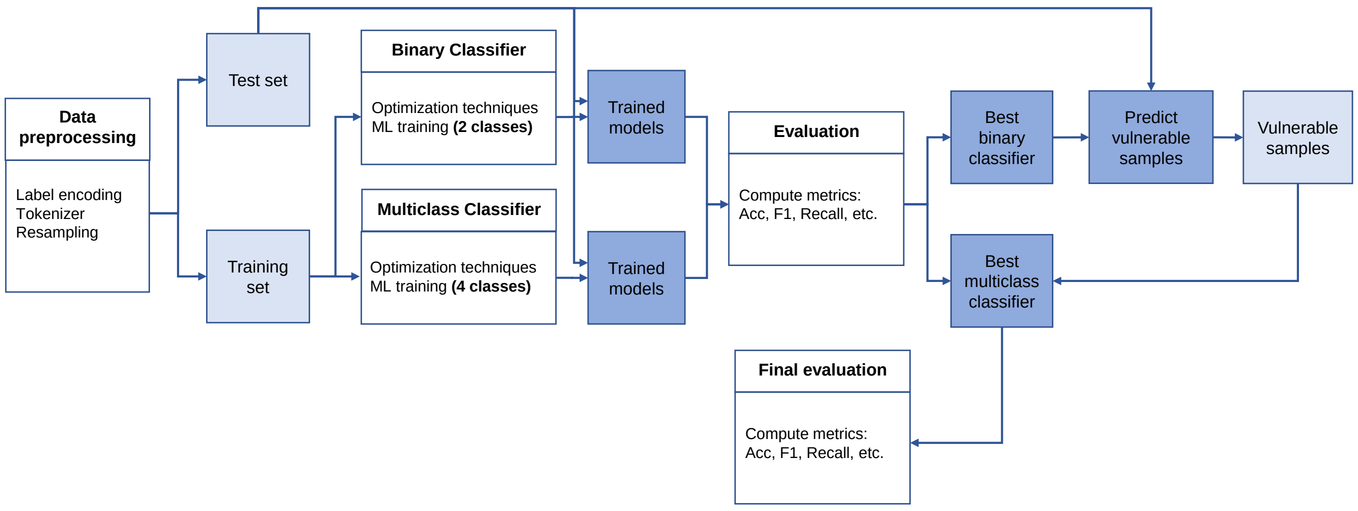


Figure 5. Contract classification executed in two phases.

For binary classification (vulnerable/non-vulnerable), the found parameters are as follows:

1. *Random forest:* number of estimators: 7; max features: 12; max depth: 17; criterion: $entropy$;
2. *Logistic regression:* solver: $newton - cg$; max iterations: 144; C: 0.71;
3. *Decision tree:* max depth: 25; criterion: $entropy$; max features: $auto$;
4. *MLP:* solver: $lbfgs$; max iterations: 147; hidden layer sizes: 136;
5. *k-NN:* weights: $distance$; number of neighbors: 3; algorithm: $auto$;
6. SVM: kernel: rbf; C: 6.23;

For multiclass classification (type of vulnerability) the found parameters are as follows:

1. *Random forest*: number of estimators: 9; max features: 26; max depth: 21; criterion: $gini$;
2. *Logistic regression*: solver: $newton - cg$; max iterations: 178; C: 5.71;
3. *Decision tree*: max depth: 22; criterion: $gini$; max features: $sqrt$;
4. *MLP* solver: $lbfgs$; max iterations: 134; hidden layer sizes: 222;
5. *k-NN* weights: $uniform$; number of neighbors: 1; algorithm: $brute$;
6. SVM kernel: rbf; C: 9.64;

After selecting the best model for binary classification, *k*-NN, the prediction of labels was performed over the testing set. The samples labeled as vulnerable were analyzed with the best model from multiclass classification, SVM. This analysis allows us to determine the class of vulnerability. The last step is the evaluation of predictions. The proposed system thus allows us to filter the non-vulnerable contracts and focus only on vulnerable ones.

4. Evaluation and Results

4.1. Utilized Metrics

To evaluate the trained model, a certain number of metrics were applied and are introduced below.

Accuracy is the metric that shows the ratio of correct predictions of the model, calculated as

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}, \tag{1}$$

where TP—true positive, TN—true negative, FP—false positive, and FN—false negative.

Precision shows the ratio of true positives to the number of positively predicted samples by

$$Precision = \frac{TP}{TP + FP}. \tag{2}$$

Recall is a fraction of correctly predicted positive samples to the number of all true-positive samples

$$Recall = \frac{TP}{TP + FN}. \tag{3}$$

F1 is a combination of the recall and precision metrics, estimated as

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}. \tag{4}$$

The specific metrics for computing accuracies are described below. The Normal Contracts Accuracy (NCA) is calculated as

$$NCA = \frac{NCCQ}{NQ}, \tag{5}$$

where $NCCQ$—the number of correctly classified normal contracts and NQ—the number of normal contracts in the dataset.

The Suicidal Contracts Accuracy (SCA) is the accuracy of the classification of suicidal contracts, which is calculated as

$$SCA = \frac{SCCQ}{SQ}, \tag{6}$$

where $SCCQ$—the number of predicted suicidal contracts and SQ—the number of suicidal contracts in the dataset.

The Prodigal Contracts Accuracy (PCA) is the accuracy of the classification of prodigal contracts, which is calculated as

$$PCA = \frac{PCCQ}{PQ}, \tag{7}$$

where $PCCQ$—the number of predicted prodigal contracts and PQ—the number of prodigal contracts in the dataset.

The Greedy Contracts Accuracy (GCA) is the accuracy of the classification of greedy contracts, which is calculated as

$$GCA = \frac{GCCQ}{GQ}, \tag{8}$$

where $GCCQ$—the number of predicted greedy contracts and GQ—the number of greedy contracts in the dataset.

The Suicidal and Prodigal Contracts Accuracy (SPCA) is the accuracy of the classification of suicidal and prodigal contracts, which is calculated as

$$SPCA = \frac{SPCCQ}{SPQ}, \tag{9}$$

where $SPCCQ$—the number of predicted suicidal and prodigal contracts and SPQ—the number of suicidal and prodigal contracts in the dataset.

False Negative Predictions (FNP) is the ratio of false-negative predictions to the total number of samples in the dataset, which is calculated as

$$FNP = \frac{FNQ}{Q}, \tag{10}$$

where FNQ—the number of false negative predictions and Q—the number of contracts in the dataset.

As an engineering-representative metric, we use the 95% average prediction time, which is the time of processing of the whole testing set divided by the number of samples in this set. Consequently, the presented result is the average time per sample (contract).

4.2. Results for Scenario 1

The results of the classification into five classes are presented in Tables 1 and 2 (the "one-phase scenario" part). According to the introduced results, the SVM algorithm achieved the best results on almost all metrics: SCA—0.9998, PCA—0.9973, GCA—0.9858, and false negative—0.0021. *k*-NN has the highest values for NCA—0.9958 and SPCA—0.9911. Additionally, SVM also performed better for accuracy—0.9888, F1—0.9888, precision—0.9889, and recall—0.9888. Furthermore, it took 0.312428 ms to classify one contract.

Table 1. Accuracy estimation for 5 classes.

Model	NCA	SCA	PCA	GCA	SPCA	FNP
Random forest	0.9910	0.9946	0.9709	0.9564	0.9863	0.0090
Logistic regression	0.8873	0.9029	0.8789	0.7317	0.8190	0.0236
Decision tree	0.9612	0.9779	0.9511	0.9296	0.9669	0.0099
MLP	0.9850	0.9956	0.9943	0.9666	0.9937	0.0082
k-NN	**0.9958**	0.9951	0.9772	0.9732	**0.9911**	0.0106
SVM	0.9614	**0.9998**	**0.9973**	**0.9858**	0.9998	**0.0021**

The confusion matrix in Figure 6 shows the SVM classes detected correctly and which of them were problematic. The algorithm failed to detect 38 greedy samples and labeled them as normal. Also, the algorithm determined 20 samples as prodigal when they were greedy. On the other hand, 131 contracts were predicted as greedy, but they were non-vulnerable.

Table 2. Results of each classification task in two scenarios.

Method	Accuracy	F1	Precision	Recall	Time, ms
			One-phase scenario		
			5 classes		
Random forest	0.9796	0.9795	0.9797	0.9796	0.000863
Logistic regression	0.8433	0.8433	0.8439	0.8433	**0.000327**
Decision tree	0.9573	0.9572	0.9572	0.9573	0.000373
MLP	0.9870	0.9870	0.9870	0.9870	0.00126
k-NN	0.9862	0.9862	0.9864	0.9862	2.707967
SVM	**0.9888**	**0.9888**	**0.9889**	**0.9888**	0.312428
			Two-phase scenario		
			2 classes		
Random forest	**0.9732**	0.9735	0.9569	0.9906	0.000939
Logistic regression	0.9189	0.9189	0.9143	0.9235	**0.0007**
Decision tree	0.9670	0.9668	**0.9667**	0.9669	0.00082
MLP	0.9713	0.9713	0.9665	0.9761	0.000931
***k*-NN**	**0.9732**	**0.9736**	0.9528	**0.9953**	0.18213
SVM	0.9724	0.9726	0.9589	0.9866	0.151349

Table 2. *Cont.*

Method	Accuracy	F1	Precision	Recall	Time, ms
4 classes					
Random forest	0.9698	0.9697	0.9698	0.9698	0.001005
Logistic regression	0.8374	0.8370	0.8368	0.8374	0.000503
Decision tree	0.9500	0.9499	0.9499	0.9500	**0.000431**
MLP	0.9838	0.9837	0.9839	0.9838	0.001292
k-NN	0.9767	0.9764	0.9774	0.9767	0.133743
SVM	**0.9899**	**0.9899**	**0.9900**	**0.9899**	0.17381
Potential example solution					
***k*-NN+SVM**	0.9921	0.9902	0.9883	0.9921	0.329363

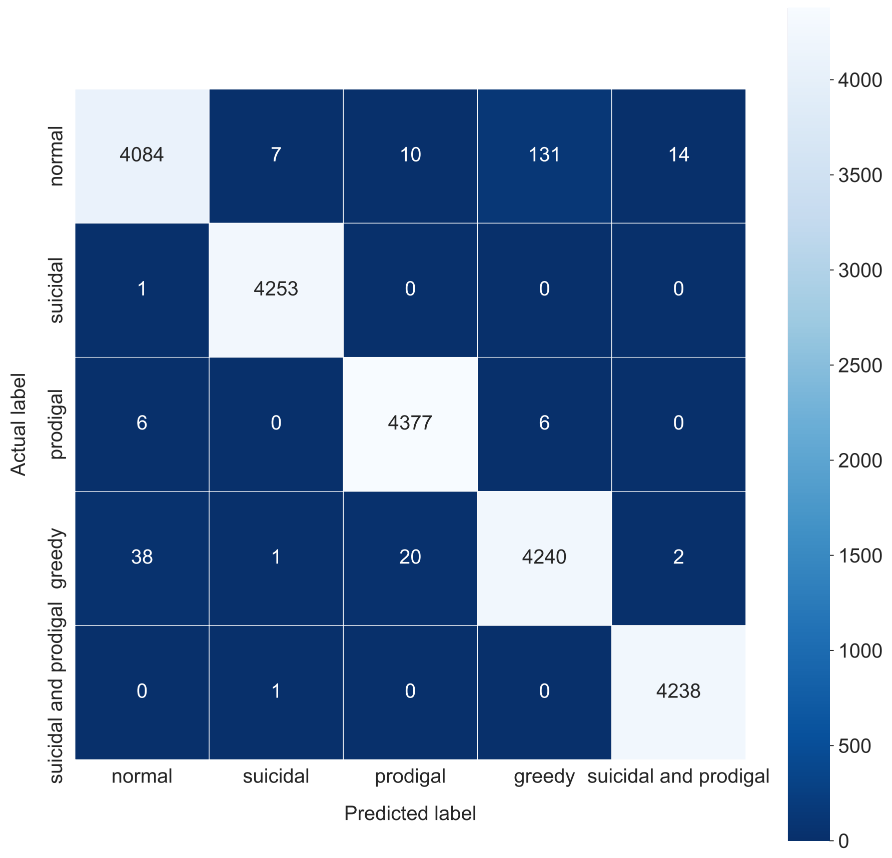


Figure 6. Confusion matrix for SVM in scenario 1.

The other methods provide somewhat worse results. Logistic regression has the worst results: NCA—0.8873, SCA—0.9029, PCA—0.8789, GCA—0.7317, SPCA—0.8190, false negative—0.0236, accuracy—0.8433, F1—0.8433, precision—0.8439, and recall—0.8433. However, other methods have results for all metrics above 0.95, comparable with the best results. However, logistic regression is the fastest algorithm for this task. It was able to process one contract in 0.000327 ms because of the simplicity of the algorithm. Conversely, the slowest method is *k*-NN, which classified the contract in 2.707967 ms, but it performed relatively well for accuracy—0.9862, F1—0.9862, precision—0.9864, and recall—0.9862.

4.3. Results for Scenario 2

4.3.1. Classification into Two Classes

The results for classification into two classes, i.e., binary classification, are shown in Table 2 (two-phase scenario). All methods perform well and achieve results of more than 90% for all metrics. As can be seen, the *k*-NN algorithm achieved the best results for F1-score—0.9736, accuracy—0.9732, and recall—0.9953. This model is relatively time-consuming: it identified a contract in 0.18213 ms. It wrongly labeled 210 normal contracts as vulnerable, and 20 vulnerable ones as normal (see confusion matrix in Figure 7).

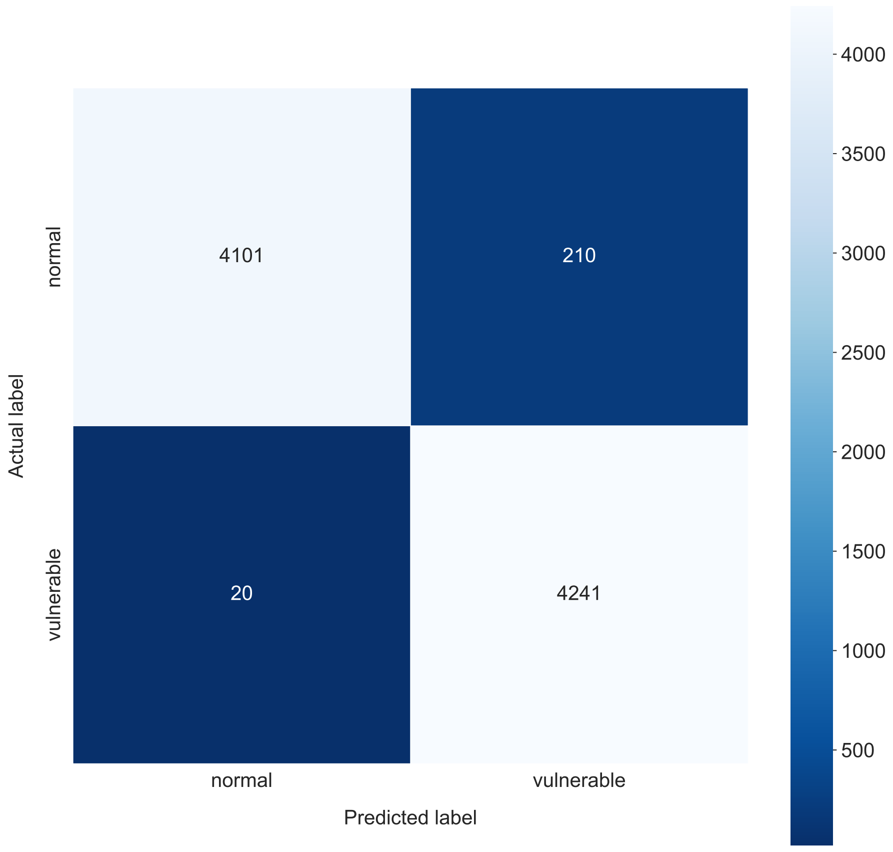


Figure 7. Confusion matrix for binary classification (*k*-NN) for scenario 2.

Alternatively, the random forest performed with the same accuracy as *k*-NN, but took less time for processing: 0.000939 ms.

The decision tree is the best for precision 0.9667 and has a comparable processing speed to the fastest algorithm of 0.00082 ms. Despite its effectiveness in terms of processing time, 0.0007 ms, logistic regression has the worst results: accuracy—0.9189, F1—0.9189, precision—0.9143, and recall—0.9235.

4.3.2. Classification into Four Classes

This scenario's second part is classifying detected vulnerable contracts into four classes. A comparison of the methods during the training phase for this classification is introduced in Tables 2 and 3. Like in binary classification, the SVM is the best according to the metrics of accuracy—0.9899, F1 score—0.9899, precision—0.9900, recall—0.9899, time for processing—0.17381 ms, SCA—0.9937, PCA—0.9777, and SPCA—0.9954. *k*-NN has the best result for GCA—0.9981. It is worth mentioning that for almost all metrics, the classification of four classes is performed better than for five classes. It is interesting to note that the decision tree has worse performance according to such metrics as accuracy—0.95, F1—0.9499, precision—0.9499, and recall—0.95, but it was the fastest algorithm in this task—0.000431 ms.

Table 3. Accuracy estimation for 4 classes.

Model	SCA	PCA	GCA	SPCA
Random forest	0.9805	0.9581	0.9691	0.9715
Logistic regression	0.8961	0.8771	0.7613	0.8132
Decision tree	0.9675	0.9436	0.9358	0.9527
MLP	0.9931	0.9688	0.9899	0.9836
k-NN	0.9803	0.9587	**0.9981**	0.9721
SVM	**0.9937**	**0.9777**	0.9930	**0.9954**

The confusion matrix is also introduced in Figure 8. Here, the main problem for SVM is to distinguish between greedy and prodigal classes. The algorithm detected 38 greedy contracts as prodigal and 12 prodigal contracts as greedy. However, mostly it did up to 7 mistakes in predictions.

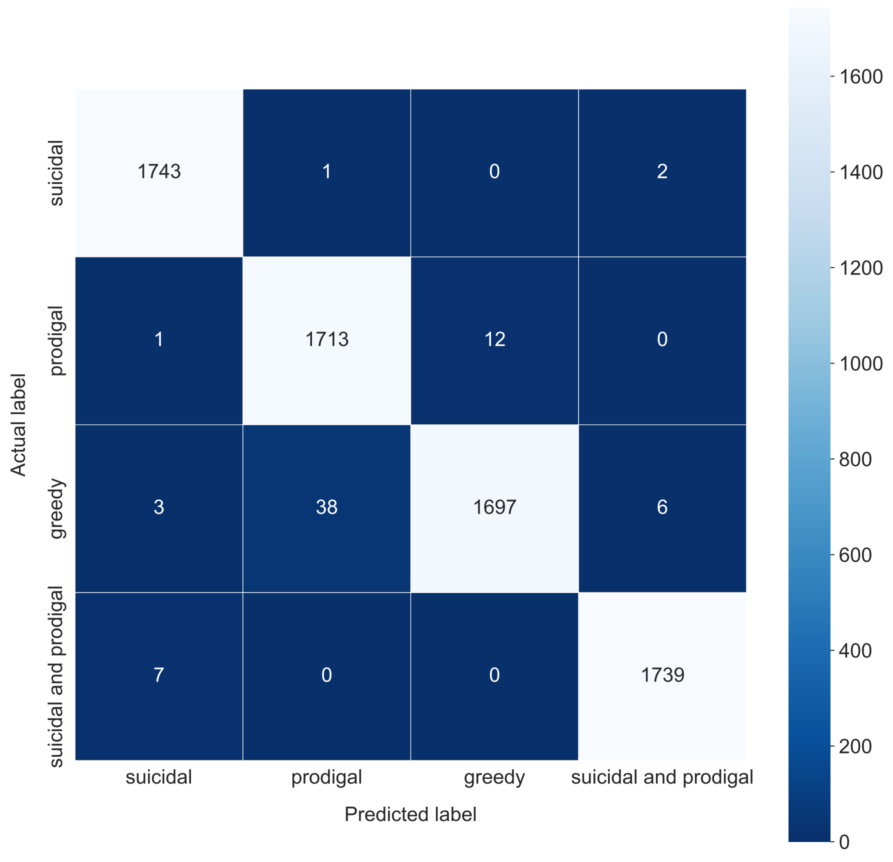


Figure 8. Confusion matrix for multiclass classification (with SVM) for scenario 2.

4.3.3. Final Results for Proposed System

The final evaluation is performed according to the mentioned workflow in Figure 5. After filtering all contracts labeled by the binary classifier as non-vulnerable, the rest were evaluated with a multiclass classifier. The total number of filtered samples is 16, 930 from the initial testing set (21,429 samples). The final results were also summrized previously in Table 2. Here, the classification of filtered samples is more accurate than the classification of all samples, like in the first scenario. The confusion matrix for the final evaluation is introduced in Figure 9. The final model performance was evaluated with the following metrics: accuracy—0.9921, F1—0.9902, precision—0.9883, recall—0.9921, SCA—0.9988, PCA—0.9918, GCA—0.9804, and SPCA—0.9982. Naturally, these correspond to both phases and, if classification is required by the integrator, it might take more time than just a classification alone; see the discussion below.

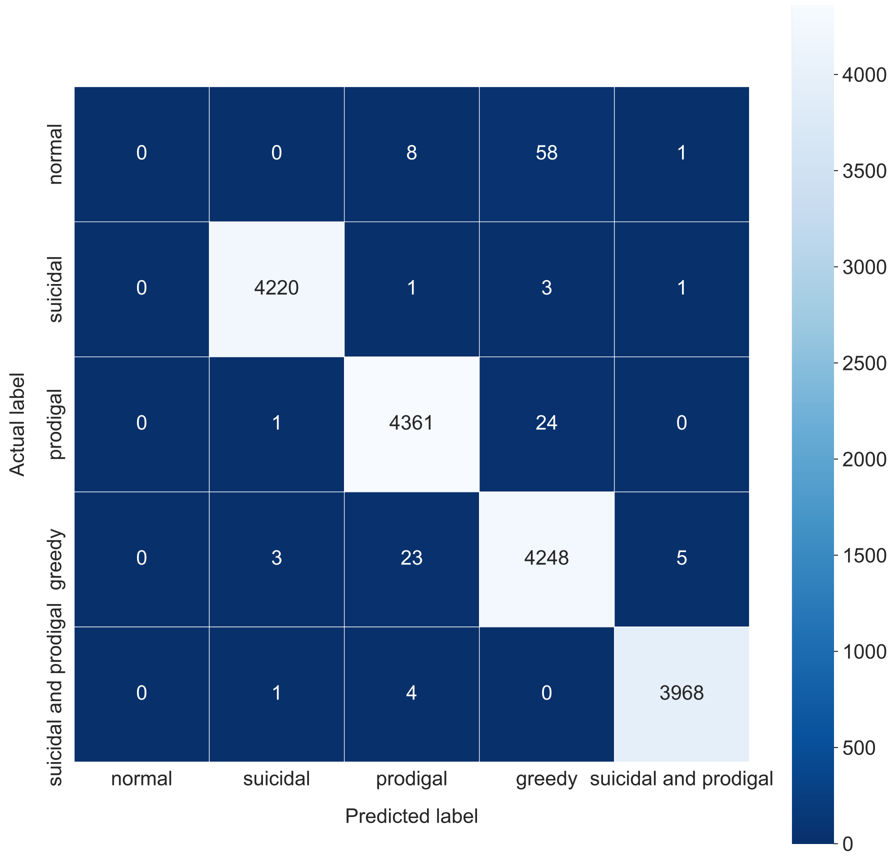


Figure 9. Final confusion matrix for scenario 2.

Interestingly, the binary classifier has labeled some normal contracts as vulnerable, shown in the confusion matrix's first row (actual label—normal). In total, 67 samples were wrongly detected. Also, the multiclass classifier, in most cases, determined the correct type of vulnerability. The main diagonal shows, with high numbers, the number of samples. However, the classifier confused different prodigal and greedy classes: 23 greedy samples were labeled as prodigal and 24 as greedy. This fact is shown with metrics PCA and GCA. In the other cases, the model performed well. Finally, the processing time is computed: 0.329363 ms. This time shows how long it will take to process vulnerable contracts, which includes binary classification and four-class classification for each contract.

5. Discussion

5.1. Numerical Perspectives

Two scenarios for contract analysis were implemented in this work. The first is a classifier in a single phase, and in the second one, the two classifiers were pretrained to perform the analysis.

The first scenario aimed to classify the given opcode into one of the five classes. The main advantage of this is the simplicity of the workflow. The SVM algorithm achieved the best results, and its accuracy is 0.9888. The MLP model provided relatively good results. Considering that this is the simplest version of the neural network, some advanced architectures of neural networks can be suitable for this task. The most complicated aspect is to determine the greedy vulnerability correctly. Since this class has more samples in the dataset, finding the patterns that would characterize it is more difficult. On the other hand, the other classes have fewer samples. These contracts were duplicated during the resampling, and, consequently, the models were adapted for these samples, or, in other words, overfitted.

For the second scenario, two classifiers were trained. For binary classification, the best one was *k*-NN with an accuracy of 0.9732. These results show that the model can effectively distinguish between normal and vulnerable contracts but sometimes needs to be corrected. The main thing is that it has a low number of false negatives—just 20 samples among a total of 8572.

The second part is classifying the vulnerability of contracts filtered in the previous step. Here, the SVM model pretrained on four classes was used. According to the results from the pretraining phase, this model achieved accuracy and F1 of 0.9899, i.e., the algorithm could distinguish between different classes. However, as in the first scenario, the main problem appears to be in differentiating between the prodigal and greedy classes. This also can be seen in the PCA and GCA metrics, which are lower than the others. The final classification also shows this problem: the PCA and GCA metrics are worse than SCA and SPCA.

Another essential aspect is the time spent processing the contracts by these algorithms. In the case of a one-phase scenario, the models which achieved higher performance (MLP, *k*-NN, and SVM) need more time to process one contract. The logistic regression is the fastest but with poor performance results. The choice of a model for a real-world scenario depends on the situation: whether it is important to process the data fast or more accurately. However, the difference in the achieved results between logistic regression and any of the MLP, *k*-NN, and SVM models is significant, so it would be preferable to use any of these three models.

In the case of the two-phase scenario, the behavior is similar, i.e., more complex models, such as *k*-NN and SVM, achieved high results but were time-consuming. However, the random forest can also be taken into consideration. It has good results and is one of the fastest algorithms according to measures. It is worth noting that results for binary classification regarding the time are better than for the five classes, at least when comparing the mentioned *k*-NN, SVM, and MLP. The next step in the two-phase scenario is the categorization of vulnerability. The most interesting models are also *k*-NN, SVM, and MLP, since they achieved high values in different metrics. The time for classification is lower than in the first scenario. On the one hand, the most successful is SVM in all metrics, but because of the complexity of the model (which showed up in the time results), the MLP can also be considered for real-world applications.

In this experiment, accuracy is the primary metric used for model selection in the two-phase scenario; in this case, *k*-NN and SVM. Their achieved times are comparable with the one-phase scenario. However, it should be mentioned that the provided time is measured when the contract is vulnerable. In other words, it will go through two phases. Despite almost the same processing time as in the one-phase scenario, time is still saved in the second scenario. Not all contracts are vulnerable, so they will be processed during the time in the same way in binary classification (as mentioned before, binary classification is less time-consuming than multiclass classification). However, in the case that a contract is vulnerable, it will be additionally categorized. On the other hand, the first scenario will spend the same time on any contract.

Notably, the evaluation with accuracy, recall, precision, and F1 metrics showed that the second scenario performed better than the first scenario. This fact suggests that two classification phases are more suitable for real-world applications. Additionally, the traditional ML methods achieved high results, which means it is unnecessary to apply some complex methods, which would take more time to process.

It is important to notice that training the models on more vulnerable samples for real-world tasks is necessary. Even the resampling method cannot prevent the overfitting problem, which can appear with a small number of samples.

The proposed model can be considered for deployment in real-world applications. Firstly, the model will filter the vulnerable contracts, and only after that will it continue categorizing contracts.

The second aspect is the possibility of retraining. Since the two phases work separately, they can also be retrained independently. As mentioned earlier, ML algorithms require a lot of data for efficient execution, which is why enhancing the system during usage in real life is possible. Even the growing number of attacks can be helpful for the model—the more vulnerable samples it can use for training, the less incorrect detection it will perform in the future. It is important to note that the proposed system is based on supervised learning.

Consequently, the model needs to be trained on labeled data. Here, the assistance of an expert would be needed. Despite that, the model can determine the vulnerability patterns with high probability, reducing the work for this expert, who would need to categorize the contracts.

5.2. Future Perspectives

Detecting vulnerable smart contracts can provide several benefits for developers and users of the Ethereum network (not focused specifically on financial segments but in broader sense). Here are some key advantages:

- Overall Network Operation: Exploits targeting vulnerable smart contracts can disrupt the Ethereum network's operation and stability. Detecting and repairing vulnerabilities with ML on the fly contributes to a more resilient and reliable network infrastructure, lowering the likelihood of service disruptions, congestion, or cascading effects caused by breached contracts.
- Enhanced Security: The Ethereum ecosystem could be made more secure overall by identifying vulnerable smart contracts [22]. Malicious actors may use vulnerabilities to act unlawfully or affect the funds. The likelihood of such assaults can be greatly decreased by identifying and addressing vulnerabilities, safeguarding user assets, and upholding network trust.
- Protection of User Privacy: Certain smart contract flaws can reveal sensitive user data or transaction details [23]. Detecting these vulnerabilities enables immediate correction, limiting illegal access to personal information and protecting user privacy.
- Trust in the System's Operation: Identifying and providing appropriate countermeasures to vulnerabilities in smart contracts contributes to the development of trust among Ethereum developers and consumers [24]. Increased trust in the security of smart contracts can lead to a greater adoption of Ethereum-based applications and services, supporting network innovation and growth beyond state-of-the-art versions.
- Evolution of Best Practices: Vulnerability detection provides significant insights into emerging threats in the smart contract world [24]. This knowledge can help the Ethereum community define best practices, code standards, and security recommendations. Sharing this information aids in the entire ecosystem's maturation and develops a security-conscious development culture.

In summary, detecting vulnerable smart contracts in Ethereum offers numerous benefits, including improved security, user privacy, network stability, trust and adoption, and the evolution of best practices.

5.3. Integration Issues

While ML can be a useful technique for discovering vulnerabilities, it also brings several limitations and obstacles when used. We further list some of the drawbacks of employing ML for this purpose:

- Limited Training Data: A large volume of labeled training data is needed to train ML models. Nevertheless, obtaining a significant and diversified collection of identified vulnerabilities in the case of vulnerable smart contracts remains close to impossible.

The restricted availability of labeled data may hamper the capacity to train precise and reliable ML models.

- Evolving Attack Techniques: The landscape of smart contract flaws and attack methods constantly changes [25]. ML models naturally use historical data to find trends and predict future outcomes. Ethereum's smart contract ecosystem is extensive and diverse, with many contract types and functionalities. If trained on a single set of contracts, ML algorithms might not generalize effectively to new contract kinds or vulnerabilities. Adapting models to different contract architectures and keeping them current with changing smart contract standards and practices can be challenging and time-consuming.
- "Back box" issue: It can be difficult to comprehend how ML models come to their conclusions because they frequently operate as black boxes when employed. In security-critical applications, explainability is essential to promote openness and confidence. It could be challenging for developers and integrators to comprehend the logic behind found vulnerabilities if a machine learning model cannot explain its predictions concisely.
- False Positives and False Negatives: ML models are prone to false positives, where they mistakenly identify a non-vulnerable contract as vulnerable, and false negatives, when they fail to recognize a contract as vulnerable. False positives can result in pointless audits or interventions, while false negatives can leave vulnerabilities and potential vulnerabilities undiscovered. It is still difficult to balance reducing false alarms and correctly identifying risks.
- Scalability: Applying ML techniques for smart contract vulnerability detection in real-life applications often requires significant computational resources and integration phases. Training complex models, deploying them in production, and maintaining them over time can be costly. Moreover, as the Ethereum network continues to grow, the volume of smart contracts increases, posing scalability challenges for machine learning-based detection approaches.

ML is becoming a central part of smart contract vulnerability detection. Nonetheless, it is still often complemented by other techniques, such as static analysis, formal verification, and manual auditing, to address these challenges and enhance the overall effectiveness of smart contract security practices [20,26,27]. However, automation appears to be the essential part of the process.

6. Conclusions

To take a step forward towards achieving better vulnerability detection, this work proposed a system for vulnerability detection in smart contracts using ML algorithms. The proposed method was tested on a real-world dataset and showed high performance on all metrics. Here, two solutions were proposed: (1) The classification of five classes (one normal and four vulnerable) is performed in a single phase. (2) The classification is performed in two phases: firstly, with binary classification, and secondly, with a classification of vulnerable contracts. Considering that the priority is to minimize false-negative results, the second solution is preferable because the rate of false negatives is lower with binary classification than with the classification of five classes.

The proposed system has some advantages compared to existing solutions. First, the proposed one is based on ML; thus, it is possible to retrain the model on newer data compared to other systems based on symbolic execution. The proposed solution predicts fewer false-negative results and outperforms them. Compared with [3], the system provides the same accurate results but allows for the vulnerability types to be distinguished within good execution time bounds.

In the future, it is planned to add other vulnerability types. Also, it is possible to use the approach of contract analysis described in [16], e.g., by the classification of contracts using ML methods and automatic execution in the private fork [28], to test and check if the classification is correct.

Author Contributions: Data curation, A.M.; formal analysis, A.M.; roles/writing—original draft, A.M.; visualization, A.M.; investigation, A.M.; software, A.M; writing—review and editing, A.O.; funding acquisition, A.O.; investigation, A.O.; methodology, A.O.; project administration, A.O. All authors have read and agreed to the published version of the manuscript.

Funding: The work of the last author was supported by the Jane and Aatos Erkko Foundation through the CONVERGENCE of Humans and Machines project, as well as through the Finnish Foundation for Technology Promotion (TES).

Data Availability Statement: Data are available in Open Access.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kushwaha, S.S.; Joshi, S.; Singh, D.; Kaur, M.; Lee, H.N. Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract. *IEEE Access* **2022**, *10*, 6605–6621. [CrossRef]
2. Gupta, R.; Patel, M.M.; Shukla, A.; Tanwar, S. Deep Learning-based Malicious Smart Contract Detection Scheme for Internet of Things Environment. *Comput. Electr. Eng.* **2022**, *97*, 107583. [CrossRef]
3. Tann, W.J.W.; Han, X.J.; Gupta, S.S.; Ong, Y.S. Towards Safer Smart Contracts: A Sequence Learning Approach to Detecting Security Threats. *arXiv* **2018**, arXiv:1811.06632.
4. Mehar, M.I.; Shier, C.L.; Giambattista, A.; Gong, E.; Fletcher, G.; Sanayhie, R.; Kim, H.M.; Laskowski, M. Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack. *J. Cases Inf. Technol. (JCIT)* **2019**, *21*, 19–32. [CrossRef]
5. Parity Technologies. A Postmortem on the Parity Multi-Sig Library Self-Destruct. Parity Technologies. 2023. Available online: https://parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/ (accessed on 25 January 2023).
6. Chen, T.; Cao, R.; Li, T.; Luo, X.; Gu, G.; Zhang, Y.; Liao, Z.; Zhu, H.; Chen, G.; He, Z.; et al. SODA: A Generic Online Detection Framework for Smart Contracts. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2020, San Diego, CA, USA, 23–26 February 2020 .
7. Zhou, H.; Milani Fard, A.; Makanju, A. The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support. *J. Cybersecur. Priv.* **2022**, *2*, 358–378. [CrossRef]
8. Vacca, A.; Di Sorbo, A.; Visaggio, C.A.; Canfora, G. A Systematic Literature Review of Blockchain and Smart Contract Development: Techniques, Tools, and Open Challenges. *J. Syst. Softw.* **2021**, *174*, 110891. [CrossRef]
9. Liao, J.W.; Tsai, T.T.; He, C.K.; Tien, C.W. Soliaudit: Smart Contract Vulnerability Assessment Based on Machine Learning and Fuzz Testing. In Proceedings of the 6th International Conference on Internet of Things: Systems, Management and Security (IOTSMS), Granada, Spain, 22–25 October 2019 ; IEEE: Piscataway, NJ, USA , 2019; pp. 458–465.
10. Google BigQuery. Kaggle Dataset. Available online: https://www.kaggle.com/datasets/bigquery/ethereum-blockchain (accessed on 25 January 2023).
11. Luu, L.; Chu, D.H.; Olickel, H.; Saxena, P.; Hobor, A. Making Smart Contracts Smarter. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016 ; pp. 254–269.
12. Qian, P.; Liu, Z.; He, Q.; Huang, B.; Tian, D.; Wang, X. Smart Contract Vulnerability Detection Technique: A Survey. *arXiv* **2022**, arXiv:2209.05872.
13. Grishchenko, I.; Maffei, M.; Schneidewind, C. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In Proceedings of the International Conference on Principles of Security and Trust, Thessaloniki, Greece, 14–20 April 2018 ; Springer: Berlin/Heidelberg, Germany, 2018; pp. 243–269.
14. ConsenSys: Mythril. Mythril GitHub. Available online: https://github.com/ConsenSys/mythril/ (accessed on 25 January 2023).
15. Tsankov, P.; Dan, A.; Drachsler-Cohen, D.; Gervais, A.; Buenzli, F.; Vechev, M. Securify: Practical Security Analysis of Smart Contracts. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 67–82.
16. Nikolić, I.; Kolluri, A.; Sergey, I.; Saxena, P.; Hobor, A. Finding the Greedy, Prodigal, and Suicidal Contracts at Scale. In Proceedings of the 34th Annual Computer Security Applications Conference, San Juan, PR, USA, 3–7 December 2018; pp. 653–663.
17. Xing, C.; Chen, Z.; Chen, L.; Guo, X.; Zheng, Z.; Li, J. A New Scheme of Vulnerability Analysis in Smart Contract with Machine Learning. *Wirel. Netw.* **2020**, 1–10. [CrossRef]
18. Hwang, S.J.; Choi, S.H.; Shin, J.; Choi, Y.H. CodeNet: Code-targeted Convolutional Neural Network Architecture for Smart Contract Vulnerability Detection. *IEEE Access* **2022**, *10*, 32595–32607. [CrossRef]
19. Liu, Z.; Qian, P.; Wang, X.; Zhuang, Y.; Qiu, L.; Wang, X. Combining Graph Neural Networks with Expert Knowledge for Smart Contract Vulnerability Detection. *IEEE Trans. Knowl. Data Eng.* **2021**, *35*, 1296–1310. [CrossRef]
20. Wang, W.; Song, J.; Xu, G.; Li, Y.; Wang, H.; Su, C. Contractward: Automated Vulnerability Detection Models for Ethereum Smart Contracts. *IEEE Trans. Netw. Sci. Eng.* **2020**, *8*, 1133–1144. [CrossRef]
21. Liashchynskyi, P.; Liashchynskyi, P. Grid search, random search, genetic algorithm: A big comparison for NAS. *arXiv* **2019**, arXiv:1912.06059.

22. Ren, M.; Ma, F.; Yin, Z.; Fu, Y.; Li, H.; Chang, W.; Jiang, Y. Making Smart Contract Development More Secure and Easier. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021; pp. 1360–1370.
23. Bhardwaj, A.; Shah, S.B.H.; Shankar, A.; Alazab, M.; Kumar, M.; Gadekallu, T.R. Penetration Testing Framework for Smart Contract Blockchain. *Peer Peer Netw. Appl.* **2021**, *14*, 2635–2650. [CrossRef]
24. Wang, H.; Li, Y.; Lin, S.W.; Ma, L.; Liu, Y. Vultron: Catching Vulnerable Smart Contracts Once and for All. In Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), Montreal, QC, Canada, 25–31 May 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 1–4.
25. Atzei, N.; Bartoletti, M.; Cimoli, T. A Survey of Attacks on Ethereum Smart Contracts (SOK). In Proceedings of the Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, 22–29 April 2017; Proceedings 6; Springer: Berlin/Heidelberg, Germany, 2017; pp. 164–186.
26. Krichen, M.; Lahami, M.; Al-Haija, Q.A. Formal Methods for the Verification of Smart Contracts: A Review. In Proceedings of the 15th International Conference on Security of Information and Networks (SIN), Sousse, Tunisia, 11–13 November 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 1–8.
27. Abdellatif, T.; Brousmiche, K.L. Formal Verification of Smart Contracts based on Users and Blockchain Behaviors Models. In Proceedings of the 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), Paris, France, 26–28 February 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 1–5.
28. Zhidanov, K.; Bezzateev, S.; Afanasyeva, A.; Sayfullin, M.; Vanurin, S.; Bardinova, Y.; Ometov, A. Blockchain Technology for Smartphones and Constrained IoT Devices: A Future Perspective and Implementation. In Proceedings of the IEEE 21st Conference on Business Informatics (CBI), Moscow, Russia, 15–17 July 2019; IEEE: Piscataway, NJ, USA, 2019; Volume 2, pp. 20–27.