Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning

Mojtaba Eshghie
eshghie@kth.se
KTH Royal Institute of Technology
Stockholm, Stockholm, Sweden

Cyrille Artho
artho@kth.se
KTH Royal Institute of Technology
Stockholm, Stockholm, Sweden

Dilian Gurov
dilian@kth.se
KTH Royal Institute of Technology
Stockholm, Stockholm, Sweden

ABSTRACT

In this work we propose Dynamit, a monitoring framework to detect reentrancy vulnerabilities in Ethereum smart contracts. The novelty of our framework is that it relies only on transaction metadata and balance data from the blockchain system; our approach requires no domain knowledge, code instrumentation, or special execution environment. Dynamit extracts features from transaction data and uses a machine learning model to classify transactions as benign or harmful. Therefore, not only can we find the contracts that are vulnerable to reentrancy attacks, but we also get an execution trace that reproduces the attack. Using a random forest classifier, our model achieved more than 90 percent accuracy on 105 transactions, showing the potential of our technique.

CCS CONCEPTS

• **Security and privacy** → *Distributed systems security*; *Software security engineering*; • **Computing methodologies** → **Machine learning**; *Neural networks.*

KEYWORDS

Smart Contracts, Vulnerability Detection, Machine Learning for Dynamic Software Analysis, Ethereum, Blockchain

ACM Reference Format:
Mojtaba Eshghie, Cyrille Artho, and Dilian Gurov. 2021. Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning. In *Evaluation and Assessment in Software Engineering (EASE 2021), June 21–23, 2021, Trondheim, Norway.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3463274.3463348

1 INTRODUCTION

A blockchain is a distributed ledger that manages assets between users. A *smart contract* encodes rules to handle the transfer of these assets. The transfers happen within transactions that are stored on the blockchain and are persistent. Therefore, smart contracts can implement a wide range of use cases, including financial and governance applications [17]. For instance, a contract could act like an autonomous agreement between multiple parties to transfer assets to desired accounts when particular conditions are met.

EASE 2021, June 21–23, 2021, Trondheim, Norway

ACM ISBN 978-1-4503-9053-8/21/06…$15.00
https://doi.org/10.1145/3463274.3463348

One of the most popular blockchain platforms that supports smart contracts is Ethereum. As of January 2021, the market capitalization of Ethereum is about 130 billion dollars [4]. Smart contracts on Ethereum allow users and other contracts to interact with them by making calls to functions in the contracts. Smart contracts in Ethereum are commonly written in Solidity, a language influenced by JavaScript. To perform the operations on a smart contract, an execution fee is needed that is called *gas*. Gas fees are paid in Ethereum's native currency, *Ether* (ETH) [9].

The novel semantics and programming model of smart contracts make it challenging to ensure their correct behavior. This makes them susceptible to bugs or vulnerabilities that may be exploited by other accounts in the Ethereum network. In fact, there has been a number of attacks on the Ethereum main network that caused the loss of millions of ETH. The most famous attack so far on Ethereum has been the attack on Decentralized Autonomous Organisation (DAO), which was conducted by exploiting the *reentrancy* vulnerability. As a result of this attack, 3.5 million ETH was stolen (about 50 M USD at the time) [7].

Reentrancy involves (unintended) repeated calls to the same function (or a set of functions) before the first invocation is finished. Such nested invocations can cause the smart contract to behave in unexpected ways, which can be exploited by an attacker, usually to transfer funds away from the victim contract. Reentrancy is known as one of the most dangerous vulnerabilities in Ethereum smart contracts [3]. Existing tools to detect reentrancy vulnerabilities use complex code analysis and handcrafted rules to carefully analyze the control flow and asset transfers in smart contracts. At a transaction level, such attacks are not explicitly observable, though. Our work attempts a completely new direction:

(1) We monitor runtime transactions at the level of the Ethereum blockchain. An example of a transaction trace from which we gather data for our machine learning model is presented in Figure 1. This monitoring does not require complex inspection of the smart contracts themselves and makes it possible to deploy our technique directly at the Ethereum blockchain client, without any modification of the smart contracts or the client involved.
(2) We use machine learning on the monitored transaction metadata. This avoids the need to design (possibly flawed) rules and also paves the way towards recognizing new types of vulnerabilities in the future.

Dynamit is designed to analyse transactions in smart contracts and report malicious ones. Our technique, when used with a random-forest model, showed high accuracy (94 %) on 105 transactions. We averaged our experimental results over ten iterations of a setting

```
blockHash: "0xb44...343c5",
blockNumber: 33614,
contractAddress: null,
cumulativeGasUsed: 162534,
from: "0x66a...1f3c7",
gasUsed: 162534,
logs: [],
logsBloom: "0x0000...00000",
status: "0x1",
to: "0x89e...7ddc7",
transactionHash: "0x3e4...039ad",
transactionIndex: 0
```

Figure 1: A sample trace (transaction receipt) retrieved from the Ethereum blockchain

that used ten-fold cross-validation for the training and test phases of all machine learning models.

The rest of this paper is organized as follows: Section 2 explains smart contracts and reentrancy, and covers related work. Section 3 describes our approach. Our experiments and their results are described in Sections 4 and 5, respectively. Section 6 covers threats to validity; Section 7 concludes and outlines future work.

2 BACKGROUND

2.1 Ethereum

Smart contracts embody a novel programming model that includes a global shared state (managed in a decentralized way on a blockchain). The global state that stores everyone's assets is manipulated automatically by the smart contracts, which are small programs that are expressed in a specific format, such as Ethereum bytecode [9]. This bytecode is usually compiled from a high-level language, e. g., Solidity [6]. The code is executed by a virtual machine, instruction by instruction. Each instruction also incurs a cost, measured in *gas*, which the invoker (user) of a smart contract has to pay. The VM manages the effects of the instructions and their cost on everyone's assets on the blockchain. Potential vulnerabilities can arise at different levels in this architecture; reentrancy is generally regarded as one of the most severe ones [2, 3].

2.2 Reentrancy Vulnerability

Contracts in Ethereum can send Ether to each other. Whenever a contract receives a message without data that contains Ether and does not specify a function, a default unnamed function, the *fallback* function, is invoked. When there is a transfer of funds from contract A to contract B, control is handed over to contract B [9]. In the time that B has control, it can call back into any public function of contract A, even the same function that issued the call to B. This situation is called *reentrancy*.

A simple example of reentrancy is illustrated in Figures 2 and 3. Contract `Vulnerable` donates to a target contract (as specified by `to` parameter in `donate`). We use this over-simplified example to illustrate the way reentrancy is exploited, and the real-world exploits may be much more complex. The intention is that a donation occur only once, but this is not checked in `donate`. An exploit is implemented by contract `Attacker`. Its `startAttack` function issues a call to `donate` in `Vulnerable`. After this call, `Vulnerable` transfers plain Ether to `Attacker`. At this point, control is passed to the fallback function of `Attacker`, which tries to call the `donate` function again. The donations continue until `Vulnerable` runs out of gas or Ether, and because only the last invocation is reverted upon failure, the attacker effectively drains the victim of all its funds [3].

In order to prevent reentrancy, one could use *function modifiers* in Solidity [9] to perform checks before giving the control to the fallback function of another contract. In case of our `Vulnerable` contract, a simple check before sending the Ether prevents the attacker from exploiting reentrancy. The safe version, `NotVulnerable`, checks and updates its state before sending Ether to the interacting contract (see Figure 4).

```
contract Vulnerable {
    function donate(address to_) public payable
    {
        require(to_.call.value(1 ether)());
    }
}
```

Figure 2: A sample contract vulnerable to attacks on reentrancy

2.3 Related Work

Program analysis techniques to detect potential vulnerabilities can be divided into *static analysis*, which analyzes the structure of code without running it, and *dynamic analysis*, which analyzes the runtime behavior of an executing program. The advantage of static analysis is that it does not require a test case to reveal a flaw; conversely, it has the disadvantage that the analysis may be overly strict and reveal *spurious* problems that are not actually exploitable flaws during program execution. Dynamic analysis, on the other hand, always produces actual executions (and thus a witness of a real problem), but may be unsuccessful at finding the right inputs to make this happen. Combinations of these techniques also exist, typically in the form of static analysis to identify parts of the program that might need closer inspection at runtime.

Static analysis tools for smart contracts include tools such as Securify [23], SmartCheck [22], and Slither [10]. These tools check code against problematic patterns that may constitute violations

```
contract Attacker {
    Vulnerable public vul_contract;
    function startAttack(address _addr) public
    {
        vulContract = Vulnerable(_addr);
        vulContract.donate(address(this));
    }
    function() public payable
    {
        vulContract.donate(address(this));
    }
}
```

Figure 3: A sample contract that exploits the vulnerable contract

```
contract NotVulnerable {
    mapping (address => bool) public donated;
    function donate(address to_) public payable
    {
        if (donated[to_] != true) {
            donated[to_] = true;
            require(to_.call.value(1 ether)());
        }
    }
}
```

Figure 4: A sample contract with the vulnerability fixed

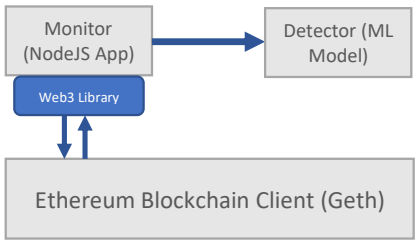

Figure 5: Dynamit framework system diagram

of coding guidelines or even potential vulnerabilities. Symbolic execution is a static analysis technique that uses path conditions (conditions about the feasibility of certain execution paths) to reason about inputs that may reach a potentially unsafe state in a program. Oyente [18] was the first tool to apply symbolic execution to smart contracts. Other tools, such as TeEther [14], MAIAN [19], and Zeus [13], followed and took different approaches at finding harmful inputs or detecting problems in the contracts.

The above-mentioned static analysis tools use rules designed by experts to detect problems. Recent works have adapted the use of machine learning to static analysis, using learned patterns instead of rules to detect problems [21], and extended this idea to the domain of smart contracts [2].

Dynamic analysis for smart contracts focuses on finding inputs that reach a program state exhibiting a problematic execution pattern (such as reentrancy), implemented in tools like Echidna [11], ContractFuzzer [12], and ReGuard [16]. The accuracy of these tools depends on the quality of the hand-coded pattern recognizer. Recent work uses an oracle that tracks the *balance* of each smart contract instance to detect fundamental misuse, thus eliminating the need for specific program patterns to detect a vulnerability [25].

Our work leverages machine learning to detect problematic execution patterns, focusing on reentrancy in smart contracts. It is, to our knowledge, the first work to analyze dynamic execution patterns in smart contracts through machine learning. In the area of malware detection, a metadata-focused approach has also been used successfully [1] by looking at the frequency and size of packets sent over encrypted connections to classify the behavior of an application.

3 METHOD

The Dynamit framework detects reentrancy vulnerabilities in deployed smart contracts without needing their source code. Dynamit considers only the dynamic behavior of the smart contract; that behavior is extracted from metadata describing the transactions between the contracts. This monitoring is based on the existing application programming interface (API) of the unmodified Ethereum blockchain client.

Dynamit consists of two parts (see Figure 5):

(1) The *Monitor,* which observes transactions in the blockchain.
(2) The *Detector,* which classifies behavior as benign or malicious.

Table 1: Features gathered by monitor and used by our ML model in detector

Feature	Monitoring Mechanism
Gas usage of transaction	Event subscription
Contract 1 balance difference	Probing
Contract 2 balance difference	Probing
Average call stack depth	Probing

The detector can be configured against various classifiers, which are first trained on a training set before our tool is put to use to detect malicious transactions in production.

3.1 Monitor

The monitor connects to the Ethereum blockchain client to gather information about desired transactions. It uses the latest version of Web3js [26], which is the official Ethereum Javascript API to connect to and probe the Ethereum network.

The monitoring obtains the data as follows:

- **Subscription to the events emitted by the Ethereum client**. These events are emitted when a transaction related to an account is issued [26]. In our work, we use *pendingTransactions* to get any new transactions related to our accounts under observation.
- **Probing the blockchain at specific intervals until the desired information is retrieved**. This is suitable for getting information about an already mined transaction or getting the state of a contract after an event.

3.2 Detector

The detector is the part of the system that distinguishes the harmful transactions from the benign ones. It consists of a part that processes and cleans the data received by the monitor, and a machine learning model that is trained as the monitor feeds in the data.

3.2.1 Extracted Features. The extracted features and the mechanism used to monitor them are presented in Table 1.

The *contract balance difference* feature is the difference of the balance of a contract before and after the transaction has taken place. In fact, the feature *contract balance difference* may easily be replaced by any other asset that is being transferred by the contracts to match the specific use case.

The *average call stack depth* is the only feature being directly retrieved from the transaction trace. Calling a regular function in a contract does not change the value of this feature significantly. However, recursive external calls will change this value drastically. This is often the case for the reentrancy vulnerability, where a particular function in the victim is recursively called until the attacker contract stops. Intuitively, this feature should have a positive correlation with a transaction being harmful. However, an attacker can easily avoid being detected by limiting the number of recursions. Therefore, we decided to put effort to randomly decrease the average call stack depth for harmful transactions, in order to make it harder for the detector to distinguish them, and to decrease the bias of the model (less complex training data would lead to high bias which increases the prediction error).

As mentioned earlier, as contract code executes on the blockchain, it consumes gas. Gas usage depends on the specific operations that a contract carries out within a transaction. Since a successful attack on a vulnerable contract may exhibit a specific execution pattern, we use this gas usage as a summary representation of the execution.

3.2.2 Classifier. To find the best model, we trained and tested the following models in our detector:

- **Random Forest (RF)**: Random forest is a popular ensemble learning method that use multiple models to increase performance of the model. Although ensemble method results in increased accuracy, it also increases the time to train the model. We built a random forest classifier combined with bagging with 100 decision trees.
- **Naive Bayes (NB)**: This classification method works based on the Bayes Theorem, and computes how probabilities of certain events are related.
- **Logistic Regression**: This model works by finding the probabilities of certain events occurring and computing their correlation.
- **K-Nearest Neighbours (KNN)**: This is a simple algorithm that works by finding the closest k neighbors to a point, and take their vote for the class label. In our case, a K-NN with 5 neighbors had the best prediction accuracy.
- **Support Vector Machine (SVM)**: A SVM maps training examples to points in space so as to maximise the gap between the classes. Using SVM it is possible to effectively perform classification on non-linearly separable data. This is done using the kernels, which can be non-linear. To build our SVM-based classifier, both linear and polynomial kernels were tested. The model with linear kernel outperformed the other one.
- **Neural Network Classifier**: We tried to build a neural network for this binary classification task using two hidden layers with 10 and 5 ReLU units for the first and second hidden layers, respectively. We used the Sigmoid activation function for the output layer. Furthermore, the binary cross-entropy and Adam optimizers were used, respectively, as the loss function and optimization method.

All of our models were built using the Scikit-learn library [20]. The Random Forest model is composed of 100 trees of type DecisionTreeClassifier in Scikit-learn.

Table 2: Set of contracts used in the experiments

Service contract	User contract
13 robust contracts	11 benign contracts
12 vulnerable contracts	9 malicious contracts

3.3 Usage of Dynamit

Let us assume developers of an application deploy it as a smart contract on top of Ethereum. Dynamit can be used by these developers to safeguard their smart contract. An administrator installs Dynamit on their own machine, and configures it to connect to the Ethereum network to monitor their deployed contract.

As the transactions are issued to the monitored smart contract, Dynamit collects and processes their metadata. The previously trained machine learning model then classifies transactions as benign or harmful. Confirming the safety of deployed smart contracts is useful as part of AIOps [5], where software is automatically deployed and monitored as in DevOps [8] with the help of AI. Information about potentially harmful contracts can be used as feedback to the developer or as part of a security information and event management (SIEM) system that may report users to an administrator or block a vulnerable contract from being used further.

4 EXPERIMENTS

We chose 25 open-source contracts for our experiments that implement a certain functionality that we denote as *service contracts* here. These contracts were originally used in [15]. Their source code is available on Etherscan [1]. We wrote 20 *user contracts* that access and utilize the functionality of these service contracts (see Table 2). These 20 user contracts are based on a manually written template, hence our approach is not yet fully automated. A service contract may be robust (not exploitable) or contain a vulnerability; likewise, a user contract may be benign or malicious. Only a combination of a vulnerable service contract with a malicious user may actually reveal the vulnerability in the service contract. The service contracts were manually reviewed, and tested against the reentrancy vulnerability, then labeled as either vulnerable or robust.

For the experiment, we monitored a total of 105 transactions generated from these contracts, with 53 benign and 52 harmful transactions. All of these transactions have been labelled manually before starting the experiment, so they could be used for both training and testing a supervised model. We feed labelled transaction data to our classifier (offline) for the training phase; in production, online (unlabelled) data can be used.

From the 105 transactions, 25 transactions were taken from the 25 open-source service contracts, which we complemented with 20 variants of user contracts. The remaining 80 transactions are generated using two pairs of contract templates (four contracts) that generate both harmful and benign transactions randomly. Contract `Vulnerable2` is one such variant of a service contract, which donates a random amount to the user (see Figure 6). To generate these random transactions, both the service and the user contracts (see Table 2) *fuzz* their behavior to represent different behaviors of real-world scenarios. Another reason for having random behavior

[1] https://etherscan.io/

(fuzzing) in both service and user contracts is that there may be a complex internal computation that has a certain call stack depth or gas usage. This potentially can make an attack harder to detect. It is desirable to have such behavior included in our data, in order to have a less biased classifier in detector. Therefore, these transactions are generated in a way to prevent overfitting in the model. For example, we fuzz the gas usage by injecting a random loop with 50 % probability in the vulnerable contract template (see lines 12–18 in Figure 6). Each use of the counter expends extra gas. Likewise, we randomize the amount donated to the user and the number of times an attacker actually exploits reentrancy, to make it harder to recognize the attacks.

Since each interaction between a service and its user is either benign or harmful, the following outcomes can occur:

- The user contract successfully exploits the reentrancy vulnerability: *harmful transaction.*
- The user contract tries to exploit a reentrancy vulnerability (which may or may not exist in the service contract) but is unsuccessful. This leads to one of the following situations:
  - The transaction and accordingly its effects on the target contract state are reverted by the Ethereum runtime environment. Such failed (reverted) transactions are not made visible through the monitoring API in Ethereum and are therefore not taken into account by our analysis.
  - The transaction is not reverted, and takes the intended original effect: *benign transaction.*
- The peer contract does not try to exploit reentrancy at all: *benign transaction.*

As mentioned earlier, after data is collected by the monitor, it is fed in to the detector for classification. We trained and tested the models in detector using above-mentioned data. For all of our models, we used stratified 10-fold cross-validated training and test sets to get consistent and reliable results. For each number in the plots, the whole experiment (including the cross validation) has been performed 10 times, and the average performance was taken. The numbers of neighbors in K-NN model and number of trees in our RF model are chosen based on empirical observations to maximize the performance of the model.

5 RESULTS

We train and test six different types of classifiers and compare them based on the average false positive rate (FPR) and false negative rate (FNR) as well as accuracy, F1-score, and recall (see Figures 7 and 8). Here is a short description about the mentioned metrics which are used to reason about the performance of machine learning models:

- *FPR:* The number of transactions that are labeled benign, and are predicted to be harmful by our machine learning model (a wrong prediction).
- *FNR:* The number of transactions that are labeled harmful, and are predicted to be benign by our machine learning model (a wrong prediction).
- *Accuracy:* The number of correct predictions divided by the total number of predictions made.
- *Recall:* The number of harmful transactions predicted to be harmful divided by the total number of harmful transactions. It is meant to capture the ability of the model to correctly catch harmful transactions (even if it means that the false positive rate is going to increase).
- *F1-score:* This is harmonic mean of the recall and *precision* (the percentage of transactions predicted as harmful and were really harmful) of the model.

```
1  contract Vulnerable2 {
2      uint public gasFuzzingCounter = 0;
3      uint public c = 0;
4      uint public d_binary = 0;
5      uint public amnt;
6      function random(uint num) private view returns
           (uint8) {
7          return uint8(uint256(keccak256(block.
               timestamp, block.difficulty))%num);
8      }
9      constructor() public payable {
10     }
11     function donate(address to_) public payable {
12         d_binary = random_binary();
13         c = random(10);
14         if (d_binary == 1) {
15             for (uint i = 0; i < c; i++) {
16                 gasFuzzingCounter++;
17             }
18         }
19         amnt = random(1000) * 500000000000000;
20         require(to_.call.value(amount)());
21     }
22 }
```

Figure 6: Source code for the smart contract used to generate random harmful transactions for the experiment

The FPR varies between 1.59 % (logistic regression) and 32.0 % (k-nearest neighbors), while the FNR is the lowest for the random forest (RF) model at 14.42 %.

The RF classifier (detector) achieves the highest accuracy (90 %). Most of the inaccuracy of the models can be attributed to the FNR. In other words, the detector is labelling a considerable number of harmful transactions as benign (even using random forest). Conversely, the low FPR makes Dynamit useful as a monitoring tool in scenarios where the cost of false positives are rather high, such as in testing or when suspending problematic contracts in production for manual review.

As mentioned earlier, the contract sets we used for random transaction generation try to disguise their behavior. We took this measure to build a realistic model and decrease bias. As a result of this, the correlation of the *average call stack depth* and the label of the transaction is very low (see Figure 9). Hence, we decided to also build the same models without the *average call stack depth* feature. The results of this version of the models are shown in Figures 10 and 11. The overall behavior of all models is consistent with the results in Figures 7 and 8. However, there are a few interesting changes. While RF is still the most accurate model and even more accurate than before, the relative reduction in the FPR of RF is higher than the one in the FNR, and as it is obvious the RF model is well-balanced in terms of false positives and false negatives. The highest average accuracy in this experiment belongs to RF (94 %). Furthermore, the accuracy, F1-score, and recall are much closer in the new experiment (11), and as presented by the confidence interval of the bars, the variation in the results is also less than

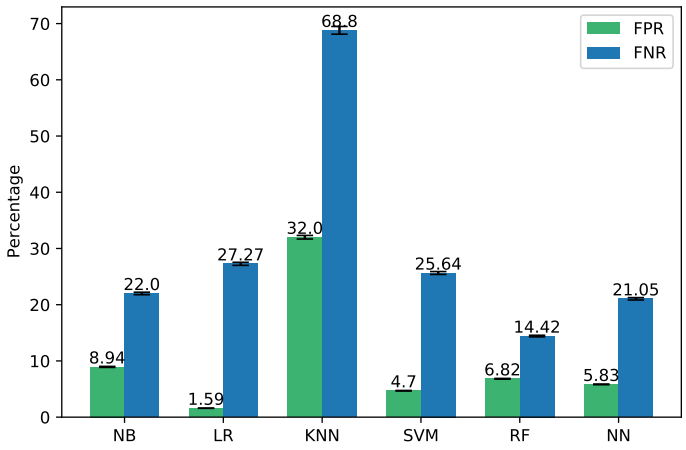

Figure 7: Average false positive rate and false negative rate for detecting vulnerable transactions with different classification models. The caps on bars show the 95 % confidence interval.

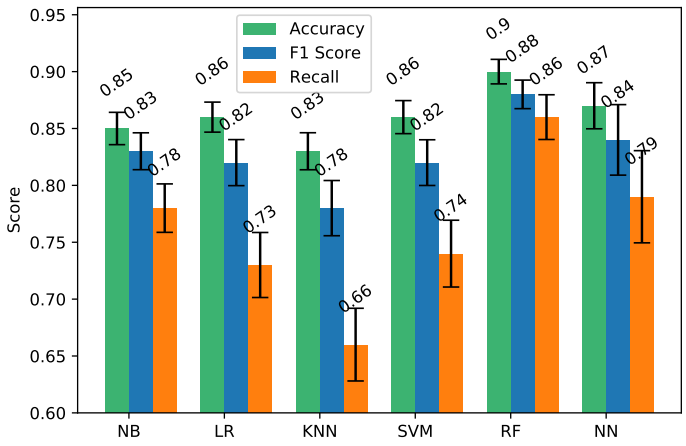

Figure 8: Average Accuracy, F1 Score, and Recall for detecting vulnerable transactions with different classification models.

in the previous set of experiments with *call stack depth* feature included. In both sets of experiments, the neural network classifier is ranked second after random forest model (with the highest mean accuracy of 88 %).

6 THREATS TO VALIDITY

We use a total number of 49 smart contracts (25 service, 20 user, and 4 random transaction generation contracts) in our experiments. In an effort to collect more realistic data, the harmful transactions issued by our own contracts are randomized to disguise their malicious nature. As mentioned in results section, this has rendered our otherwise important *average call stack depth* feature useless

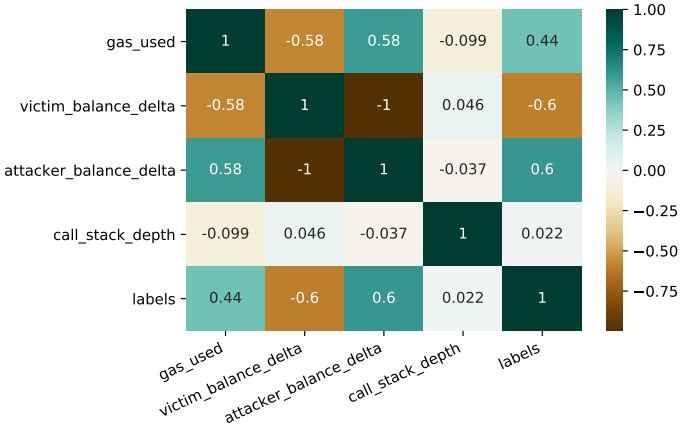

Figure 9: Feature correlation heatmap for the detectors with label 1 for harmful transactions and 0 for benign ones.

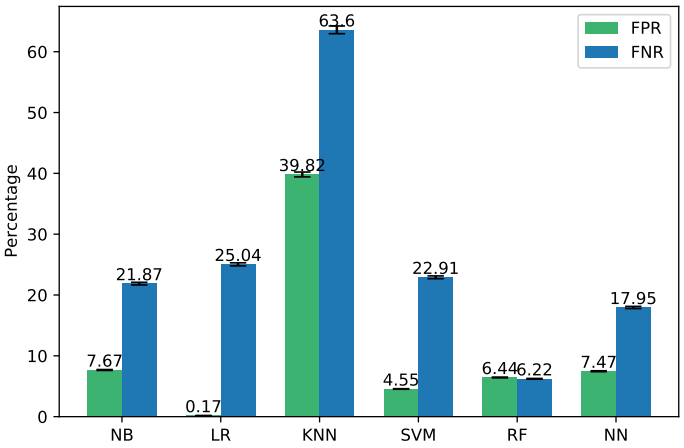

Figure 10: Average false positive rate and false negative rate for detecting vulnerable transactions with different classification models without the *average call stack depth* feature.

by making the *average call stack depth* of a harmful contract seem like a benign one, and vice versa. From a different point of view, this also shows the possibility of tricking a dynamic detector if it only uses checks on certain variables (such as the balance) to detect a vulnerability. Our results suggest that a combination of our machine learning-based detector and an oracle-supported dynamic vulnerability detection [25] may decrease the number of false negatives.

Another consideration is the amount of randomness that our randomly generated transactions have. In case there is not enough randomness, our machine learning model in detector will exhibit bias, making it less likely to catch more complex attacks. Our random transaction generator uses the block's timestamp and difficulty to generate random numbers. Since we have used a private deployment of Ethereum blockchain, the mentioned variables were

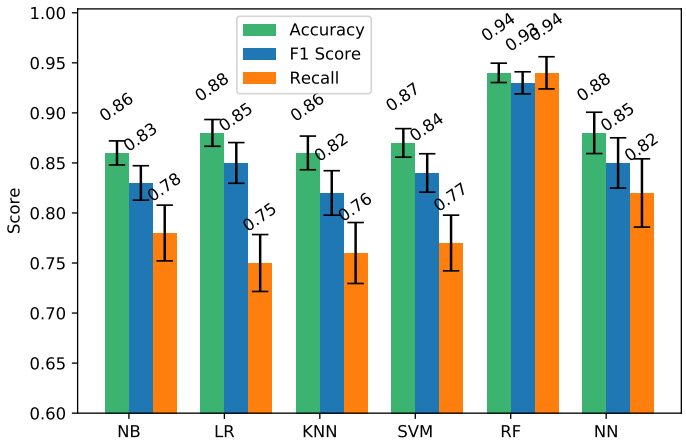

Figure 11: Average Accuracy, F1 Score, and Recall for detecting vulnerable transactions with different classification models without the *average call stack depth* feature.

controllable by increasing the mining frequency and issuing transaction generation commands each 30 seconds. Using this method, we verified that the random transaction generation system has enough randomness.

7 CONCLUSION AND FUTURE WORK

In this work, we present Dynamit, a dynamic vulnerability detection framework for Ethereum smart contracts. Dynamit detects vulnerable smart contracts by classifying harmful transactions in a blockchain using machine learning on transactional metadata. We achieve 94 % accuracy on a data set of 105 transactions. Using machine learning to detect vulnerability one can avoid writing rules by hand, and rely on the learning ability of the machine learning algorithms.

To further develop Dynamit, we will investigate automatic test-case generation tools such as Vultron [24]. Such tools can generate labeled transactions and create benign and malicious user contracts to reproduce them. Another direction for future work is to find more features to make the detection more accurate. An example would be to observe bookkeeping variables inside the contracts, and the way they change, as additional indicators of a smart contract being exploited. Finally, we will consider analyzing sequences of multiple transactions and applying other types of machine learning to the data, to increase the capabilities of our detector and to analyze other types of vulnerabilities as well.

REFERENCES

[1] Zeeshan Afzal, Anna Brunström, Stefan Lindskog, and Johan Garcia. 2020. Using Features of Encrypted Network Traffic to Detect Malware. In *25th Nordic Conference on Secure IT Systems (LNCS)*. Springer, Online.

[2] Nami Ashizawa, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. 2021. Eth2Vec: Learning Contract-Wide Code Representations for Vulnerability Detection on Ethereum Smart Contracts. *arXiv preprint arXiv:2101.02377* (2021).

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust*, Matteo Maffei and Mark Ryan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 164–186.

[4] coinmarketcap.com. 2021. Cryptocurrency Prices, Charts And Market Capitalizations. https://coinmarketcap.com/

[5] Yingnong Dang, Qingwei Lin, and Peng Huang. 2019. AIOps: real-world challenges and research innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, Montreal, Quebec, Canada, 4–5.

[6] Chris Dannen. 2017. *Introducing Ethereum and Solidity*. Vol. 1. Springer, New York, NY, United States.

[7] dasp.co. 2021. DASP - TOP 10. https://dasp.co/

[8] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. 2016. DevOps. *Ieee Software* 33, 3 (2016), 94–100.

[9] Ethereum. 2021. Home. https://ethereum.org

[10] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE Press, Montreal, Quebec, Canada, 8–15.

[11] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Event USA, 557–560.

[12] Bo Jiang, Ye Liu, and WK Chan. 2018. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, ACM, Montpellier France, 259–269.

[13] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, San Diego, California, US. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf

[14] Johannes Krupp and Christian Rossow. 2018. TeEther: Gnawing at Ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium USENIX Security*. ACM, Baltimore, MD, USA, 1317–1333.

[15] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-2_Li_paper.pdf

[16] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, Gothenburg, Sweden, 65–68. https://doi.org/10.1145/3183440.3183495

[17] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016-10-24) *(CCS '16)*. ACM, Vienna Austria, 254–269. https://doi.org/10.1145/2976749.2978309

[18] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *ACM Conference on Computer and Communications Security*. ACM, ACM, Vienna Austria, 254–269.

[19] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, San Juan PR USA, 653–663.

[20] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of Machine Learning Research* 12 (2011), 2825–2830.

[21] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Berkeley, CA, United States, 611–626.

[22] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static analysis of Ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. ACM, Gothenburg, Sweden, 9–16.

[23] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *ACM Conference on Computer and Communications Security*. ACM, Toronto Canada, 67–82.

[24] H. Wang, Y. Li, S. Lin, L. Ma, and Y. Liu. 2019. VULTRON: Catching Vulnerable Smart Contracts Once and for All. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)* (2019-05). IEEE Press, Montreal, Quebec, Canada, 1–4. https://doi.org/10.1109/ICSE-NIER.2019.00009

[25] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu. 2020. Oracle-Supported Dynamic Exploit Generation for Smart Contracts. *IEEE Transactions on Dependable and Secure Computing* (2020), 1–1. https://doi.org/10.1109/TDSC.2020.3037332

[26] Web3JS. 2021. Web3.Js - Ethereum JavaScript API — Web3.Js 1.0.0 Documentation. https://web3js.readthedocs.io/en/v1.3.0/