Towards Safer Smart Contracts: A Sequence Learning Approach to Detecting Vulnerabilities

A. Wesley Joon-Wie Tann
wesley001@ntu.edu.sg

Xing Jie Han
xhan017@e.ntu.edu.sg

Sourav Sen Gupta
sg.sourav@ntu.edu.sg

Yew-Soon Ong
asysong@ntu.edu.sg

School of Computer Science and Engineering, Nanyang Technological University, Singapore

Abstract

Symbolic analysis of security exploits in smart contracts has demonstrated to be valuable for analyzing predefined vulnerability properties. While some symbolic tools perform complex analysis steps (which require predetermined invocation depth to search the execution paths), they employ fixed definitions of these vulnerabilities. However, vulnerabilities evolve. The number of contracts on blockchains like Ethereum has increased 176 fold since December 2015 [33]. If these symbolic tools fail to update over time, they could allow entire classes of vulnerabilities to go undetected, leading to unintended consequences. In this paper, we aim to have smart contracts that are less vulnerable to a broad class of emerging threats. In particular, we propose a novel approach of sequential learning of smart contract vulnerabilities using machine learning — long-short term memory (LSTM) — that perpetually learns from an increasing number of contracts handled over time, leading to safer smart contracts. Our experimental studies on approximately one million smart contracts for learning revealed encouraging results. A detection accuracy of 97% on contract vulnerabilities has been observed. In addition, our machine learning approach also correctly detected 76% of contract vulnerabilities that would otherwise be deemed as false positive errors by a symbolic tool. Last but not least, the proposed approach correctly identified a broader class of vulnerabilities when considering a subset of 10,000 contracts that are sampled from unflagged contracts.

1 Introduction

Smart contracts provide automated peer-to-peer transactions while leveraging on the benefits of the decentralization provided by blockchains. As smart contracts are able to hold virtual coins worth upwards of hundreds of USD each, they have allowed the automated transfer of monetary values or assets via the logic of the contract while having the correctness of its execution governed by the consensus protocol [31]. The inclusion of automation in blockchain resulted in rapid adaptation of the technology in various sectors such as finance, healthcare, and even insurance [42], with Ethereum, the most popular platform for smart contracts having a market capitalization upwards of $21 billion USD [2]. Due to the fully autonomous nature of smart contracts, vulnerabilities are especially damaging as they are largely irreversible due to the immutability of blockchains.

On Ethereum alone, over 3.6 million Ether (virtual coins used by Ethereum) were stolen from a decentralized investment fund called The DAO (Decentralized Autonomous Organization) in June 2016, incurring losses of up to $70 million USD [18]. In November 2017, $300 million USD was frozen because of Parity's MultiSig wallet [34]. Both hacks that occurred were due to exploitable logic within smart contracts themselves, and these incidents highlighted a strong imperative for the security of smart contracts.

The tools used in smart contract symbolic analysis are mainly based on formal methods of verification. While most analysis tools have applied both static and dynamic analyses to automatically detect bugs in smart contracts [27, 21, 37], some have focused on finding vulnerabilities across a long sequence of innovations of a contract [25]. OYENTE is one such example of an automatic bug detector. It was proposed to act as a form of pre-deployment mitigation, by analyzing smart contracts for vulnerabilities at a bytecode level [27]. It uses symbolic execution to capture traces that match the characteristics of the classes of vulnerabilities as defined. However, it is not complete as confirmations of flagged contracts being vulnerable were only done manually in the presence of contract source code.

Recently, it has been shown that MAIAN, the first tool for precisely specifying and reasoning about trace properties, which employs inter-procedural symbolic analysis and concrete validation for exhibiting real exploits [33], was able to capture many well-known examples of unreliable bugs. Using predefined execution trace vulnerabilities directly from the bytecode of Ethereum smart contracts, Maian labels vulnerable contracts as one or two of the three categories —

suicidal, prodigal, and greedy. Maian is able to detect different classes of vulnerabilities that may only appear after multiple invocations, while verifying its results on a private fork of Ethereum. However, the degree of accuracy in its detection can be limited by its invocation depth, whereby states that vulnerabilities may occur in were not reached due to a tradeoff between analysis time and exhaustiveness of search. In addition, analysis of such characterization only allows concrete checking for bugs by running contracts with source code access.

On the other hand, in the field of machine learning, recurrent neural networks are exceptionally expressive and powerful models adapted to sequential data. The LSTM model is a compelling variant of recurrent networks mainly used to solve difficult sequential problems such as speech recognition [19, 12], machine translation [11, 41], and natural language processing [17, 26]. In recent years, there has been an increasing interest in the security of smart contracts and machine learning in security, with papers on automated vulnerability analysis [3], neural networks for guessing passwords [28], exploits for a contract developed from bytecode [24], and taxonomy of common programming pitfalls [5], with some even employing machine learning techniques for improved performance and added adaptiveness with availability of new contracts.

In this work, we introduce a machine learning model for detecting smart contract vulnerabilities at an opcode level. To the best of our knowledge, this is the first machine learning approach to smart contract vulnerabilities detection. We study the applicability of using a LSTM model to the task of functioning as a simple, efficient yet adaptive smart contract vulnerability detector. As smart contracts become available in a sequential order, they could be used to update the LSTM model for future contracts at each point in time. Since only around 1% of the smart contracts (we refer to Etherscan [1]), have original (Solidity) source code, it highlights the utility of a LSTM learning model as a smart contract security tool.

Contributions. Our contributions in this work are as follows:

- We show that a LSTM learning algorithm is able to supersede the performance of symbolic analysis tool [33] in detecting smart contract security vulnerabilities.
- We experimentally demonstrate that the LSTM performance continually improves over time with new contracts, achieving (1) an overall detection accuracy of 97% (2) improvement to 76% on false positive errors.
- We analyze a subset of 10,000 contracts unflagged by Maian, but were classified as vulnerable by the LSTM. Running these contracts with available source code through another symbolic analysis tool SECURIFY [37], we confirmed that multiple contracts were indeed vulnerable. Our LSTM tool detects a broader class of security issues.
- By demonstrating that the LSTM tool is a competitive alternative to symbolic analysis tools, we set a benchmark for future work on machine learning models that detect smart contract vulnerabilities.

2 Background

2.1 Smart Contracts

Smart contracts are autonomous, state-based executable code that are stored, verified, and executed on the blockchain. Ethereum smart contracts are predominantly written in a Turing-complete, stack-based bytecode language – Solidity. A smart contract is deployed onto the Ethereum blockchain in the form of a transaction by a sender, in which an address is assigned to the contract. Each smart contract contains a state (account balance and private storage), and executable code. Once deployed, a smart contract is immutable and no modifications can be made to the contract. However, it may be killed if a `Suicide` instruction in the contract is executed.

Contracts, once deployed on blockchain, may be invoked by sending transactions to the contract addresses, along with input data and *gas*, the "fuel" for smart contract execution. In Ethereum, gas is assigned proportionately to the amount of computation required for each instruction in its instruction set [14]. This gas is used as an incentive within the proof-of-work system for executing the contracts. If gas is insufficient or exhausted before the end of execution, no gas is refunded to the caller and the transaction (including state) is reverted. No transactions can be sent to or from a killed contract.

In Ethereum, an invocation of a smart contract is executed by every full-node in the network to reach a consensus on the input, considering the current state of the blockchain and the state of the executing contract. The contract would then update the contract state, transfer values to other contract addresses, and possibly execute functions of other contracts.

2.2 Contracts with Vulnerabilities

Due to the autonomy and immutability of smart contracts, once an attack is executed successfully on a contract, it is impossible for the transaction to be reversed without performing a hard-fork [10] on the underlying blockchain. As the distribution of smart contracts within Ethereum is heavily skewed towards the financial sector (primarily used for transfer of assets or funds) [6], some of the past attacks have incurred multimillion-dollar losses. This highlights a strong need for security of smart contracts. Although there are several existing studies and classifications of vulnerabilities of smart contracts [27, 21, 9], we primarily focus on the classifications proposed by Nikolic et al. [33], due to their extensive coverage, as well as the availability of their open-source tool Maian. We will briefly go over some of the concepts and the vulnerability classifications highlighted in the paper.

An execution trace of a smart contract is a series of contract invocations that occurred during its lifetime. Vulnerabilities that occur over a sequence of contract invocations are classified as *trace vulnerabilities*. In [33], the vulnerabilities in Ethereum smart contracts were classified under three categories — suicidal, prodigal, and greedy.

Suicidal Contracts. Smart contracts that can be killed by any arbitrary address were classified as *suicidal*. Although some contracts have an option to kill themselves as a mitigation against attacks, if improperly implemented, the same feature may allow any other user the option of killing the contract as well. This occurred during the ParitySig attack [34], where an arbitrary user managed to gain ownership of a library contract and killed it, rendering any other contract that relied on this library useless, effectively locking their funds.

Prodigal Contracts. Smart contracts that can leak funds to arbitrary addresses other than the owner, or has never deposited Ether in the contract are classified as *prodigal*. Contracts often have internal calls to send funds to other contracts or addresses. However, if there are insufficient mechanisms in place to guard the availability of such calls, attackers may be able to exploit this call to funnel Ether to their own accounts, draining the vulnerable contract of its funds.

Greedy Contracts. Smart contracts that are unable to release Ether are classified as *greedy*. Following the ParitySig attack [34], many accounts dependent on the parity contract were unable to release funds, resulting in an estimated loss of $30 million USD. Within the greedy class, the vulnerable contracts are subdivided into two categories – (a) contracts that accept Ether but completely lack instructions to send funds, and (b) contracts that accept Ether and contain instructions to send funds, but are unable to perform the task.

2.3 Deep Neural Networks

Deep Neural Networks (DNNs) are exceptionally expressive and powerful machine learning models. Recurrent Neural Networks (RNNs) are DNNs adapted to sequence data. These models can learn and achieve outstanding performance on many hard sequential learning problems such as speech recognition, machine translation, and natural language processing.

2.3.1 Recurrent Neural Networks

RNNs possess a remarkable ability to learn highly accurate models using only two hidden layers [32]. While neural networks are related to traditional probabilistic models, these networks learn a much more complicated computation. Moreover, whenever labeled datasets are available, large networks utilize a supervised learning technique called backpropagation for training network parameters. If there exists a set of parameters whereby a large neural network can achieve good performance on the problem, supervised backpropagation converges to these parameters to solve the task.

RNNs were proposed in the 80s [39, 36] for performing time series prediction. The architecture of the recurrent network is similar to a standard neural network. The main difference is that they are allowed connections among hidden units associated with a time delay. These models are able to retain memory about inputs observed in the past, enabling them to discover temporal relationships in the data. As shown in Figure 1, a single recurrent neural network layer takes the current input x_t and one of the hidden states in the previous time-step $h^1_{t-1(2)}$ to produce an output y_t.

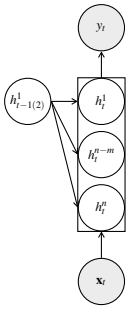


Figure 1: Schematic of a single layer in an RNN.

However, standard RNNs, shown in Figure 2, are hard to properly train in practice. The main reason why the model is so unmanageable is that it suffers from both exploding and vanishing gradients [7]. Both issues are due to the RNN's recurrent nature. The gradients are approximately equal to the recurrent weight matrix raised to a high power. These repetitive matrix powers lead to either growing or shrinking gradients at a rate that is exponential in the number of timesteps.

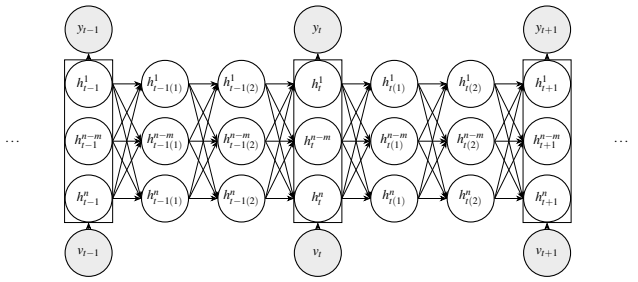


Figure 2: Schematic of a standard RNN.

While the exploding gradients problem is relatively easy to solve by simply shrinking gradients with norms pass-

ing a certain threshold, a method known as gradient clipping [35, 29], the vanishing gradient issue is much more challenging. This is because vanishing gradients do not cause the gradient itself to be small. In fact, the gradient's component in directions that correspond to short-term dependencies is large, while the component in directions that correspond to long-term dependencies is small. As a result, recurrent networks are able to easily learn the short-term dependencies but not the long-term ones.

2.4 Long Short-Term Memory RNN

In order to address the vanishing gradient and long-term dependency issues of standard RNNs, the long short-term memory (LSTM) network was proposed [15, 20]. In the LSTM, gate functions were recommended to be used for controlling information flow in any given recurrent unit — an input gate, a forget gate, and an output gate. An input gate functions as a gate keeper to allow relevant signals through into the hidden context. On the other hand, the forget gate is used to determine the amount of prior information remembered for the current time-step, and the output gate functions as a prediction mechanism. In practical implementations, the element-wise sigmoid function which outputs soft values between 0 and 1 is usually chosen as a function for the gates. This allows for ease of convex optimization. By introducing such information gate controls, the LSTM almost always performs much better than standard RNNs.

RNNs take a sequence $\{x_1, x_2, \ldots, x_T\}$ as input and construct a corresponding sequence of hidden states (or representations) $\{h_1, h_2, \ldots, h_T\}$. In the simplest case, a single-layer recurrent network uses the hidden representations $\{h_1, h_2, \ldots, h_T\}$ for estimation and prediction. On the other hand, in deep RNNs, each hidden layer uses the hidden states of the previous layer as inputs. That is, the hidden states in layer $k-1$ are used as inputs to layer k. In RNNs, every hidden state in each layer performs memory-based learning to place importance on relevant features of task using previous inputs. Previous hidden states and current inputs are transformed into a new hidden state, and it is achieved through a recurrent operator that takes in (h_{t-1}, x_t), such as:

$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b),$$

where W_h, W_x, and b are parameters of the layer and $\tanh(\cdot)$ represents the standard hyperbolic tangent function.

As for LSTM, the architecture is specifically designed to handle recurrent operations. In the architecture, a memory cell c_t, as shown in Figure 3, is introduced for internal long-term storage. As we recall that the hidden state h_t is an approximate representation of state at time-step t, both c_t and h_t are computed via three gate functions to retain both long and short term storage of information. The forget gate f_t, via an element-wise product, directly connects c_t to the memory cell c_{t-1} of the previous time-step. Using large values for the forget gates would cause the cell to retain almost all of its previous values. In addition, input gate i_t and output gate o_t control the flow of information within themselves. Each gate function has its own weight matrix and a bias vector. We denote the parameters with subscripts f for the forget gate function, i for the input gate function, and o for the output gate function respectively (e.g., W_{xf}, W_{hf}, and b_f are parameters of the forget gate function).

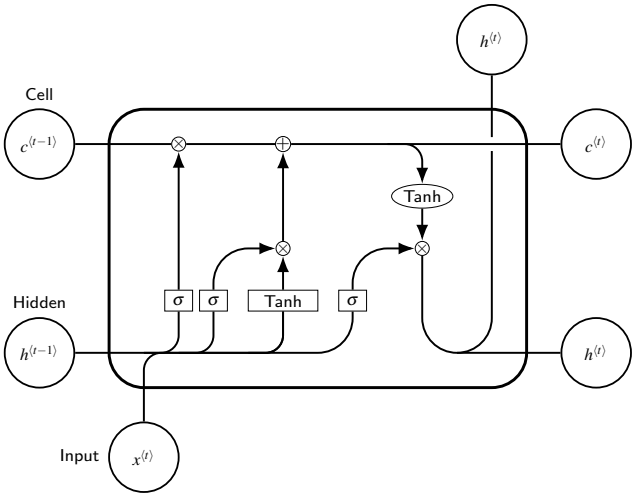


Figure 3: Schematic of a Long Short-Term Memory Cell.

Practitioners across various fields in sequence modeling use slightly different LSTM variants. In this work, we follow the model of leading natural language processing research [17], used to handle complex sequences with long-range structure. The following is the formal definition of our full LSTM architecture, without peep-hole connections,

$$i_t = \sigma(W_{xi} x_t + W_{hi} h_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{xf} x_t + W_{hf} h_{t-1} + b_f) \quad (2)$$

$$o_t = \sigma(W_{xo} x_t + W_{ho} h_{t-1} + b_o) \quad (3)$$

$$g_t = \tanh(W_{xg} x_t + W_{hg} h_{t-1} + b_g) \quad (4)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad (5)$$

$$h_t = o_t \odot \tanh(c_t) \quad (6)$$

where $\sigma(\cdot)$ is the sigmoid function, $\tanh(\cdot)$ is the hyperbolic tangent function, and $\odot$ denotes element-wise product.

3 Learning Smart Contract Vulnerability

In this section, we propose modeling of smart contract vulnerabilities using a sequential machine learning approach, and explain how a LSTM learning model handles the semantic representations of smart contract opcode. We present the vulnerability detection objective, required optimization

to improve detection model accuracy, and properties of smart contract opcode as a sequence.

3.1 Classification of Contract Vulnerability

The objective of our LSTM learning model is to perform a two-class classification, in order to detect if any given smart contract contains security vulnerabilities. Motivated by the concepts in optimization, the objective in the LSTM learning is to minimize the detection loss function, in order to maximize classification accuracy. Through the loss provided for each training smart contract, we ideally expect the sequence model to learn from the errors.

3.1.1 Measure of LSTM Prediction Loss

Loss functions of learning models are mostly application specific and are selected based on how they affect the performance of the classifiers [4, 8]. The most common ones used to measure the performance of a classification model are the cross-entropy loss (logarithmic loss), softmax, and squared loss.

In our case, we have chosen the logarithmic loss or the binary cross-entropy loss function. It is preferred as we formalized the smart contract vulnerability detection into a binary classification problem. To understand the cross-entropy loss, let us take a closer look at a neuron in a single neural network. Suppose we are trying to train the network using smart contract opcode as input. In Figure 4, a neuron is shown with several input opcodes (x_1, x_2, x_3), corresponding weights (w_1, w_2, w_3), and a bias term x_{+1}:

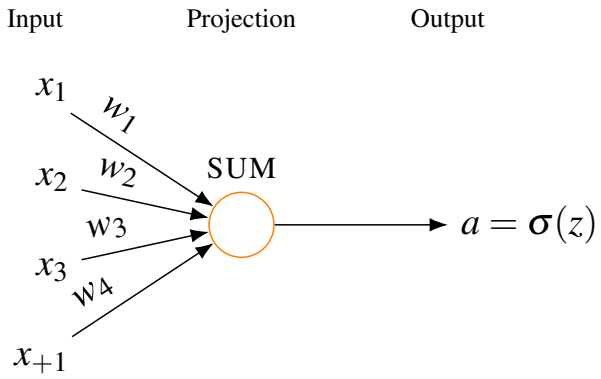


Figure 4: Training one conventional neuron.

The output from the single neuron is $a = \sigma(z)$, where $z = \sum_j w_j x_j + x_{+1}$ is the weighted sum of the opcodes, which is a value between 0 and 1 estimating the likelihood a contract is vulnerable. Now that we have formally defined a neuron's operation during training phase, we can proceed to define the

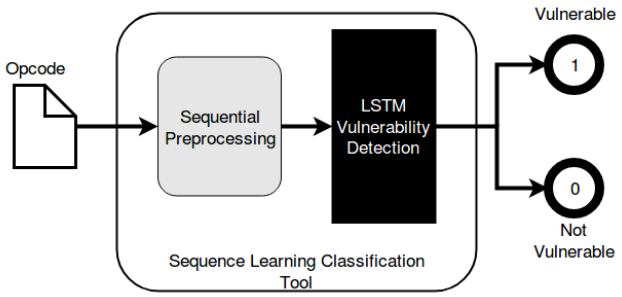


Figure 5: LSTM Smart Contract Vulnerability Classification Tool.

derivation of binary cross-entropy loss function $\mathcal{L}$:

$$\mathcal{L} = -\frac{1}{N}\sum_x [y \ln a + (1-y)\ln(1-a)], \tag{7}$$

where N is the total number of contract opcodes in training dataset, the sum over all training opcodes x, and the corresponding desired vulnerability estimate y. As the neuron improves its estimation of desired outputs $\mathbf{y}$ for all training opcodes $\mathbf{X}$, the summation of cross-entropy loss is positive and tends toward zero. This means that as a model learns to be more accurate in classifying smart contracts over time, it minimizes the distance between output estimate a and the desired output y. A perfect classifier would achieve a log loss of precisely zero.

3.1.2 Maximizing Detection Accuracy

In order to maximize LSTM classification accuracy, we employ the most successful algorithm for minimizing loss functions - backpropagation [36]. Back-propagation applies chain rule to derive derivatives of the loss function $\mathcal{L}$ with respect to local weight and bias parameters in the neural network, which are used to propagate vulnerability detection errors back into the LSTM network. These parameters are then adjusted by a user-set learning rate for descent along the gradient to reach maximum detection accuracy.

Likewise, we trained our LSTM learning model using a gradient based method in mini-batches of opcodes. A standard stochastic gradient descent (SGD) update has the following form:

$$w \leftarrow w + \alpha \nabla_W \eta(w) \tag{8}$$

where α is the learning rate, and $\nabla_W \eta(w)$ is the gradient of the maximum detection objective with respect to the parameters w as computed from mini-batches of training opcodes $(\mathbf{X}, \mathbf{y})$.

In our LSTM learning model, we use the efficient Adam [23] optimizer for stochastic gradient descent, in order

60 60 52 36 15 61 57 60 35 7c 90 04 63 16 80 63 14 61 57 80 63 14 61 57 80 63 14 61 57 5b 61 5b 60 60 90 54 90 61 0a 90 04 73 16 73 16
34 60 51 80 90 50 60 60 51 80 83 03 81 85 87 61 5a 03 f1 92 50 50 50 15 61 57 7f 60 60 90 54 90 61 0a 90 04 73 16 60 51 80 82 73 16 73
16 81 52 60 01 91 50 50 60 51 80 91 03 90 a1 61 56 5b 60 60 fd 5b 5b 56 5b 00 5b 34 15 61 57 fe 5b 61 60 80 80 35 73 16 90 60 01 90 91
90 50 50 61 56 5b 00 5b 34 15 61 57 fe 5b 61 60 80 80 35 73 16 90 60 01 90 91 90 50 50 61 56 5b 00 5b 61 60 80 80 35 73 16 90 60 01 90
91 90 80 35 90 60 01 90 82 01 80 35 90 60 01 90 80 80 60 01 60 80 91 04 02 60 01 60 51 90 81 01 60 52 80 93 92 91 90 81 81 52 60 01 83
83 80 82 84 37 82 01 91 50 50 50 50 50 50 91 90 50 50 61 56 5b 00 5b 60 60 90 54 90 61 0a 90 04 73 16 73 16 33 73 16 14 15 15 61 57 60
60 fd 5b 80 60 60 61 0a 81 54 81 73 02 19 16 90 83 73 16 02 17 90 55 50 5b 5b 50 56 5b 60 60 90 54 90 61 0a 90 04 73 16 73 16 33 73 16
14 15 15 61 57 60 60 fd 5b 80 60 60 61 0a 81 54 81 73 02 19 16 90 83 73 16 02 17 90 55 50 5b 5b 50 56 5b 60 60 90 54 90 61 0a 90 04 73
16 73 16 33 73 16 14 15 15 61 57 60 60 fd 5b 81 73 16 34 82 60 51 80 82 80 51 90 60 01 90 80 83 83 60 83 14 61 57 5b 80 51 82 52 60 83
11 15 61 57 60 82 01 91 50 60 81 01 90 50 60 83 03 92 50 61 56 5b 50 50 50 90 50 90 81 01 90 60 16 80 15 61 57 80 82 03 80 51 60 83 60
03 61 0a 03 19 16 81 52 60 01 91 50 5b 50 91 50 50 60 60 51 80 83 03 81 85 87 61 5a 03 f1 92 50 50 50 15 15 61 57 60 60 fd 5b 5b 5b 50
50 56 00 a1 65 20 15 8a 30 b8 8e 51 e2 52 a0 8c 15 9c 7b

Figure 6: Sample opcode sequence used as input data to the LSTM learning model.

to improve convergence and cross-validation results. Adam is a popular variant of the heuristic basic SGD used to accelerate learning by adaptively tuning the learning rate for each opcode. Adam maintains learning rates for each parameter and adapts them based on the average of recent magnitudes of gradients. We found that it is an optimizer that leads to fast convergence, and it aids non-stationary problems such as the smart contract security vulnerability detection task in improving performance.

3.2 Sequential Modeling of Smart Contracts

In this section, we first introduce the Ethereum opcode sequence processed by the LSTM model, followed by the usage of smart contract opcode sequence as input for our detection tool to detect smart contract vulnerabilities.

3.2.1 Ethereum Opcode Sequence

Smart contract vulnerability detection, like many sequence learning tasks, involves processing sequential opcode data. More precisely, opcodes are a sequence of numbers interpreted by the machine (virtual or silicon) that represents the type of operations to be executed. In the Ethereum environment, opcodes are a string of low level human-readable instructions specified in the yellow paper [40]. The machine instruction language is processed by Ethereum Virtual Machine (EVM), a stack-based architecture with a word size of 256-bit. Each instruction is defined with an opcode (value), name (mnemonic), δ value, α value, and a description. For each instruction, the α value is the number of additional items placed on the stack for that instruction. Similarly, the δ value is the number of items required on the stack for that instruction.

To generate the labels required for supervised machine learning, the contracts were processed by passing bytecodes through Maian to obtain vulnerability classifications. In the process, opcodes were also retrieved. A sample EVM opcode thus produced, which the LSTM model takes as input is shown in Figure 6. The addresses of the contracts were saved, along with the valid corresponding EVM opcodes, and vulnerability classifications (category) into a data-frame, Figure 7.

	ADDRESS	OPCODE	CATEGORY
0	0x219f4ee903f78e78773e5d1e3520cfd507485bc6	60 60 52 36 15 61 57 60 35 7c 90 04 63 16 80 6...	1 0 0 0
1	0x560ed796aa8d23411b94b9d047ecdda39d4fcdeb	60 60 52 36 15 61 57 60 35 7c 90 04 63 16 80 6...	1 0 0 0
2	0xe726f97ff3c63dd71c1520c102adc39d1a2693ea	60 60 52 36 15 61 57 60 35 7c 90 04 63 16 80 6...	1 0 0 0
3	0x8ebac490495958b3804bb079e259340f0f53b69c	60 60 52 36 15 61 57 60 35 7c 90 04 63 16 80 6...	1 0 0 0
4	0x190700d69031db6b072a30577f9b3dbc53a320a1	60 60 52 36 15 61 57 60 35 7c 90 04 63 16 80 6...	1 0 0 0

Figure 7: Dataset: Contract address, opcode, and category.

3.2.2 Opcode Sequence for Vulnerability Detection

Numerous tasks with sequential inputs and/or sequential outputs can be modeled with RNNs [22]. For our application in opcode smart contract vulnerability detection, where inputs consists of a sequence of opcodes, opcodes are typically fed into the network in consecutive time steps. The most straightforward way to represent opcodes is to use a binary vector with length equal to the size of machine instruction list for each opcode in the directory — *one-hot encoding*, as shown in Figure 8.

$$\begin{bmatrix}1\\0\\0\\\vdots\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\\vdots\\0\\0\end{bmatrix}, \begin{bmatrix}1\\0\\1\\\vdots\\0\\0\end{bmatrix}, \cdots, \begin{bmatrix}0\\0\\0\\\vdots\\0\\1\end{bmatrix}$$

Figure 8: Left to right: one-hot vectors representing the first, second, third, and last opcodes in the instruction list, respectively.

Such a simple encoding [13] has many disadvantages. First, it is an inefficient way of representing opcodes, as large

sparse vectors are created when the number of instructions increases. On top of that, one-hot vectors do not capture any measure of similarity between opcodes in the encoding. Hence, we model opcodes with code vectors. It represents a significant leap forward in advancing the ability to analyze relationships between individual opcodes and opcode sequences. Code vectors are able to capture potential relationships in sequences, such as syntactic structure, semantic meaning, and contextual closeness. The LSTM learns these relationships when given a collection of supervised smart contract opcode data to initialize the vectors using an embedding algorithm [30].

The embedding, shown in Figure 9, is a dense matrix in a linear space, which achieves two important functions. Firstly, by using an embedding with a much smaller dimension than the directory, it reduces the dimension of opcode representations from the size of the directory to the embedding size ($|\mathcal{U}| \ll |\mathcal{D}|$), where $|\mathcal{U}|$ and $|\mathcal{D}|$ are the embedding and directory sizes respectively. Secondly, learning the code embedding helps in finding best possible representations, and groups similar opcodes in a linear space.

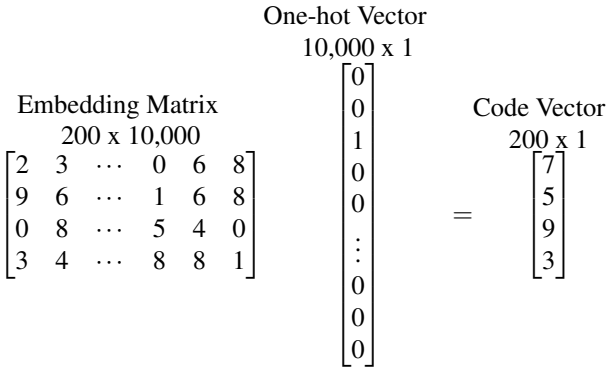


Figure 9: Example of Opcode Embedding.

In our opcode sequence modeling experiments, opcodes are encoded with a one-hot encoding, and input is fed in the LSTM one opcode at a time. As for the output, a sigmoid layer is used. A sigmoid layer is a special case of the logistic function with output value from 0 to 1. Intuitively, the output corresponds to the probability that each opcode sequence is categorized as a predicted class.

4 Experiments

We trained and tested the proposed LSTM algorithm on 920,179 smart contracts, obtained by downloading the Ethereum blockchain dataset from Google BigQuery [16]. This dataset includes the first block of Ethereum, up until block 4,799,998, which was the last block mined on December 26, 2017.

In addition, we tested and analyzed a subset of 10,000 unflagged contracts, and discovered it includes vulnerabilities that were undetected.[1]

4.1 Smart Contract Label Generation

We used the Ethereum dataset downloaded from Google BigQuery [16]. First, we parse the contracts bytecode into opcode using the EVM instruction list. In order to obtain labels for smart contracts in blocks 0 to 4,799,999, we ran the contracts through the Maian tool. A total of 920,179 contracts were processed, producing a number of flagged contracts. Preprocessing our dataset using the Maian tool, we collected the sequential opcodes, which are instructions found in the EVM list of execution code, as inputs for our LSTM learning model. In the data category, a string "1 0 0 0" represents negatives (no vulnerabilities detected by Maian), "0 1 0 0" represents suicidal, "0 0 1 0" represents prodigal, and "0 0 0 1" represents greedy.

Next, we removed the wrongly flagged contracts (false positives) identified in Nikolic et al. [33], which the authors generously shared the prodigal and suicidal cases.[2] Since no data was available for the wrongly flagged greedy contracts, we assumed all contracts in category (b) of greedy contracts as false positives (recall from section 2.2), in accordance with findings in Nikolic et al. [33], and we removed them from our dataset. After cleaning and preprocessing our data, we report the number of distinct contracts, calculated by comparing the contract opcodes. Given the large number of flagged contracts, we then checked for duplicates. We found 8640 distinct contracts that were flagged as suicidal (1378), prodigal (1632), and greedy (5801). From this set, we found 171 contracts, which were both suicidal and prodigal.

Table 1 is a summary of data processed using the Maian tool. The difference of 50,719 processed contracts between the 970,898 contracts previously reported [33] and the 920,179 processed by us was due to empty contracts. In addition, we believe that the difference in number of flagged contracts was due to an updated version of the Maian tool since the numbers were last reported in March 2018.

Category	**Reported by [33] Maian Paper**	**Processed by us Maian Tool**	**Distinct (flagged)**
Contracts Processed	970,898	920,179	
Suicidal	1495	1544	1378
Prodigal (Leak)	1504	1786	1632
Greedy (Lock)	31,201	17,084	5801

Table 1: Processed and categorized contracts by Maian.

Using this dataset, we trained and tested the LSTM learning model on a subset of 920,179 smart contracts consisting

of 884,273 unflagged contracts and 8640 flagged contracts, from which we removed invalid opcode instructions and duplicates. We chose this security vulnerability detection task with this specific subset of the entire Ethereum dataset because of the public availability of smart contracts data, which has been analyzed symbolically [33], and it serves as a baseline for our model.

We used a fixed directory of 150 execution instructions for the smart contracts opcode. Since 150 is a comparatively small number when compared with most other sequence tasks in machine learning, all unique instructions were included as most frequent codes in the learning model.

4.2 Preprocessing Labeled Opcodes

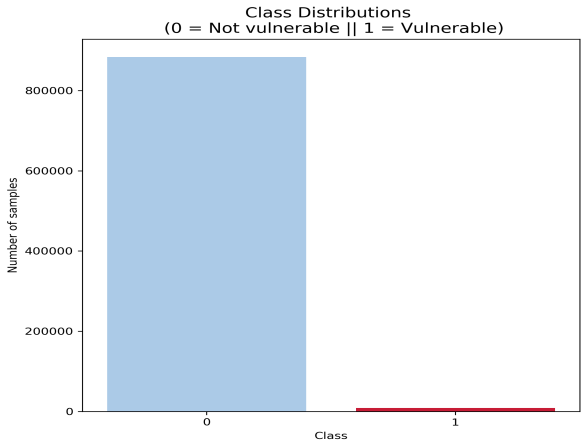
While we collected a moderately large training set, it was highly imbalanced. It is an issue with classification problems where the classes are not represented equally, and one class outnumbers the other classes by a large proportion. Based on the distribution of our dataset, Figure 10a, 99.03% of the contracts are labeled as not vulnerable by Maian, while only 0.97% of contracts are either greedy, suicidal, and/or prodigal. In order to handle the imbalanced set, we grouped all vulnerable contracts together to retrieve 8640 samples in one class, and samples not classified in any of the vulnerable categories were grouped into another class. Hence, samples categorized as "1 0 0 0" with no security vulnerabilities and any other samples not categorized as "1 0 0 0" with vulnerabilities, are labeled into classes with integers 0 and 1 respectively.

Next, we resampled the dataset to achieve a more balanced distribution, where approximately 85% of the contracts are from class 0, while 15% of the data are made up of contracts from class 1. We randomly sampled 450,000 contracts from the class 0 set, and created nine duplicates of all contracts in the class 1 set. Table 2 summarizes the number of contracts in each class of the newly balanced dataset. After performing the resampling, we ended up with a more balanced dataset of 527,760 samples, as shown in Figure 10b.

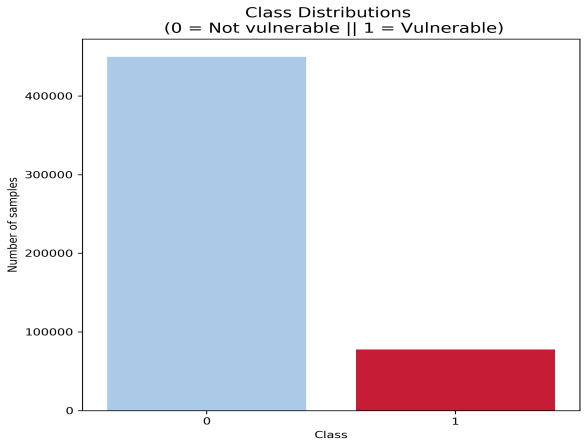
Class	# of contracts	
Unflagged as Vulnerable	450,000	
Flagged as Vulnerable	Suicidal	10,863
	Prodigal	13,149
	Greedy	52,209
	Suicidal & Prodigal	1,539

Table 2: Number of contracts in each class of balanced data.

We found that the LSTM model was fairly easy to train on this new balanced dataset. A shallow one-layer LSTM with 64 units, and 128 dimensional word embeddings, with an input vocabulary of 150 opcode instructions was chosen. The classification task is based on a binary output using the sigmoid activation function. We found shallow LSTMs to generalize well over our rebalanced data set, and it does not overfit the training samples. The resulting LSTM has 177,538 parameters, with training details as follows:



(a) Imbalanced dataset



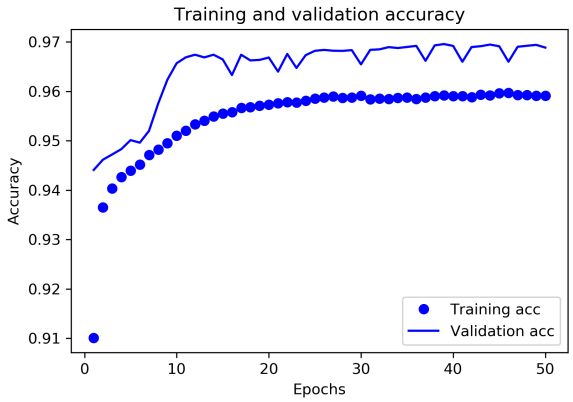
(b) Rebalanced dataset

Figure 10: Imbalanced and rebalanced datasets

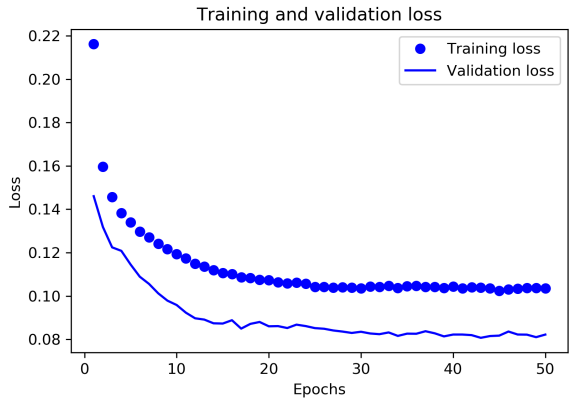
- We divided 527,760 samples into 64% (337,766) training, 16% (84,442) validation, and 20% (105,552) test.
- We used Adam as the adaptive gradient descent optimizer, and trained our LSTM model for a total of 50 epochs.
- We used batches of 256 opcode sequences at a time for the gradient and training exercise in our model.
- We used binary cross-entropy loss (log loss), which measures the performance of a classification model with output of a soft value between 0 and 1.

- As opcode of smart contracts vary in length and complexity, we set the maximum length to 130, and zero-padded the opcode of contracts that were shorter than 130.

The training and validation process plots are shown in Figure 11. We report the classification accuracy and loss of the LSTM model on both training and validation datasets.



(a) Accuracy in Training and Validation



(b) Loss in Training and Validation

Figure 11: Training and validation over 50 epochs.

4.3 Results

The experimental results of our LSTM learning model on the true positive (TP) and false positive (FP) smart contracts, as flagged by the Maian tool are shown in Table 3.

4.3.1 Balanced Dataset

From the results, we can see that our trained LSTM model performs better than the overall performance of the Maian tool. For the test set, which consists of 105,552 samples from the cleaned dataset, the LSTM model achieved an accuracy of 97.22% and loss of 07.15%. On top of the 97.22% classification accuracy in the cleaned set, it was also able to correctly identify contracts that the Maian tool falsely flagged, with an overall accuracy of 76.94%. In summary,

Category	# Correctly classified	# Contracts tested (distinct FPs)	Accuracy %
Test Set	102,622	105,552	97.22
All FPs	347	(451)	76.94
Suicidal FPs	67	(72)	93.06
Prodigal FPs	127	(177)	71.75
Greedy FPs	153	(202)	75.74

Table 3: LSTM classification performance on contracts falsely flagged by Maian tool.

- the LSTM model significantly outperformed Maian in detecting vulnerabilities of smart contracts, and
- the LSTM model was able to correctly classify contracts that the Maian tool falsely flagged, eliminating more than three-quarters of the false positive errors.

We note that while symbolic analyzers are able to perform in-depth analysis of smart contract properties to detect the exact bugs, the LSTM model is able to easily and accurately detect vulnerabilities in the smart contracts.

Moreover, concrete validation of flagged contracts in the case of Maian requires the creation of a private fork of the original Ethereum blockchain, and it can only analyze a contract from a particular block height where the contract is still alive. Not only is it a complicated and time consuming task, it is not clear how one selects a block height to include all flagged contracts that are still alive. Given that the LSTM model is a simple machine learning technique to implement, we recommend to replace concrete validation with a learning model.

4.3.2 Checking Contracts Unflagged by Maian

In addition to testing the balanced dataset, we now present test results of the contracts that were not included in the balanced set. Given the large number of contracts in this additional set of unflagged contracts, we selected a random subset of 10,000 samples from it to test the classification accuracy of the LSTM. We found that our learning tool achieved an accuracy of 81.52%.

Out of the 10,000 samples tested, 1848 contracts were classified as vulnerable. Distribution of the LSTM predicted estimates is shown in Figure 12. 185 in 1848 were distinct contracts, and only 35 in the set of 185 contracts have Solidity source code available on Ethereum. In order to validate the 35 contracts that were flagged, we ran the contracts through Securify [37], a web-based security scanner

for smart contracts. Out of the 35 contracts, only 31 were able to be analyzed by the software tool. 23 out of 31 contracts were flagged to be in violation of several security issues such as transaction ordering dependency (TOD), reentrancy, missing input validation, and unrestricted write to storage. 7 out of 31 contracts were categorized with 'warning', which means the tool could not guarantee a security violation in contract, even though the vulnerability exists.

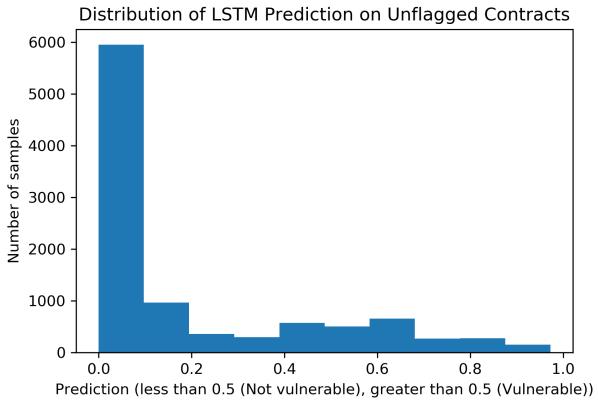


Figure 12: Histogram of LSTM Prediction on 10,000 Unflagged Contracts.

In the end, only 1 out of 31 contracts was found to have no issues. This reinforces our belief in a fast and reliable learning model for identifying vulnerable smart contracts, in contrast with a standard semantic/symbolic analysis tool. Our results also shed light on the importance of a tool that continually learns new security vulnerabilities from recent smart contracts, and quickly produce accurate detection results over a broad class of contract vulnerabilities.

5 Conclusion

We have presented a scientific work — based on machine learning towards safer smart contracts — that learns broad emerging threats with an increasing number of contracts handled over time. Our results suggest that sequential learning of smart contract vulnerabilities provides significant improvement over symbolic analysis tools, achieving detection accuracy of 97% on 105,552 contracts. On top of that, 76% of vulnerabilities in particularly challenging contracts, which were otherwise deemed as false positives by Maian, were correctly detected by our sequential learning model. Furthermore, our approach detected a broader class of vulnerabilities on a subset of 10,000 contracts sampled from unflagged contracts. This makes it possible to build a general smart contract vulnerabilities detection tool based on machine learning, which allows for superior detection capabilities.

In future work, we plan to study the impact of state-of-the-art sequence modeling techniques on smart contract vulnerabilities detection. The Transformer — a new simple network architecture that dispenses with recurrence entirely and is based solely on attention mechanisms — has been shown to be superior to recurrent neural networks [38]. Further experiments using attention models might allow smart contract vulnerabilities to be detected more effectively. While we measure detection accuracy based on classification correctness with respect to given labels, a remaining challenge is obtaining more accurate labels to improve detection capabilities of models during training.

References

[1] Etherscan verified source codes. [Online]. Available: `https://etherscan.io/contractsVerified`.

[2] Ethereum (eth) market cap. `https://coinmarketcap.com/currencies/ethereum/`, 2016. [Online; accessed 20-September-2018].

[3] ALHUZALI, A., GJOMEMO, R., ESHETE, B., AND VENKATAKRISHNAN, V. NAVEX: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, 2018), USENIX Association, pp. 377–392.

[4] ALTUN, Y., JOHNSON, M., AND HOFMANN, T. Investigating loss functions and optimization methods for discriminative learning of label sequences. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing* (Stroudsburg, PA, USA, 2003), EMNLP '03, Association for Computational Linguistics, pp. 145–152.

[5] ATZEI, N., BARTOLETTI, M., AND CIMOLI, T. A survey of attacks on ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204* (New York, NY, USA, 2017), Springer-Verlag New York, Inc., pp. 164–186.

[6] BARTOLETTI, M., AND POMPIANU, L. An empirical analysis of smart contracts: platforms, applications, and design patterns. *CoRR abs/1703.06322* (2017).

[7] BENGIO, Y., SIMARD, P., AND FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks 5*, 2 (1994), 157–166.

[8] BISHOP, C. *Neural networks for pattern recognition*. Oxford University Press, USA, 1995.

[9] BRAGAGNOLO, S., ROCHA, H., DENKER, M., AND DUCASSE, S. Smartinspect: solidity smart contract inspector. *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)* (2018).

[10] BUTERIN, V. Hard fork completed. `https://blog.ethereum.org/2016/07/20/hard-fork-completed/`, Jul 2016.

[11] CHO, K., VAN MERRIENBOER, B., GLEHRE, ., BAHDANAU, D., BOUGARES, F., SCHWENK, H., AND BENGIO, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *EMNLP* (2014), A. Moschitti, B. Pang, and W. Daelemans, Eds., ACL, pp. 1724–1734.

[12] DAHL, G. E., YU, D., DENG, L., AND ACERO, A. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Trans. Audio, Speech and Lang. Proc. 20*, 1 (Jan. 2012), 30–42.

[13] ELMAN, J. L. Finding structure in time. *COGNITIVE SCIENCE 14*, 2 (1990), 179–211.

[14] GAVIN, W. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper 151* (2014).

[15] GERS, F. A., SCHMIDHUBER, J., AND CUMMINS, F. Learning to forget: Continual prediction with LSTM. In *Proc. ICANN'99, Int. Conf. on Artificial Neural Networks* (Edinburgh, Scotland, 1999), IEE, London, pp. 850–855.

[16] GOOGLE. Google BigQuery - Ethereum. `https://bigquery.cloud.google.com/dataset/bigquery-public-data:ethereum_blockchain`.

[17] GRAVES, A. Generating sequences with recurrent neural networks. *CoRR abs/1308.0850* (2013).

[18] HACKING DISTRIBUTED. Analysis of the DAO exploit. `http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`, 2016. [Online; accessed 20-September-2018].

[19] HINTON, G., DENG, L., YU, D., DAHL, G., MOHAMED, A.-R., JAITLY, N., SENIOR, A., VANHOUCKE, V., NGUYEN, P., KINGSBURY, B., AND SAINATH, T. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine 29* (November 2012), 82–97.

[20] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[21] KALRA, S., GOEL, S., DHAWAN, M., AND SHARMA, S. Zeus: Analyzing safety of smart contracts. *Proceedings 2018 Network and Distributed System Security Symposium* (2018).

[22] KARPATHY, A. The Unreasonable Effectiveness of Recurrent Neural Networks, May 2015.

[23] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. *CoRR abs/1412.6980* (2014).

[24] KRUPP, J., AND ROSSOW, C. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, 2018), USENIX Association, pp. 1317–1333.

[25] LIU, C., LIU, H., CAO, Z., CHEN, Z., CHEN, B., AND ROSCOE, B. Reguard: Finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (New York, NY, USA, 2018), ICSE '18, ACM, pp. 65–68.

[26] LUONG, M.-T., PHAM, H., AND MANNING, C. D. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).

[27] LUU, L., CHU, D.-H., OLICKEL, H., SAXENA, P., AND HOBOR, A. Making smart contracts smarter. Cryptology ePrint Archive, Report 2016/633, 2016. `https://eprint.iacr.org/2016/633`.

[28] MELICHER, W., UR, B., KOMANDURI, S., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Fast, lean, and accurate: Modeling password guessability using neural networks. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association.

[29] MIKOLOV, T. *Statistical language models based on neural networks*. PhD thesis, Brno University of Technology, 2012.

[30] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. *CoRR abs/1301.3781* (2013).

[31] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system.

[32] NGUYEN, D., AND WIDROW, B. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *IJCNN* (1990), IEEE, pp. 21–26.

[33] NIKOLIC, I., KOLLURI, A., SERGEY, I., SAXENA, P., AND HOBOR, A. Finding the greedy, prodigal, and suicidal contracts at scale. *CoRR abs/1802.06038* (2018).

[34] PARITY TECHNOLOGIES. Security Alert. https://paritytech.io/security-alert-2/, 2017. [Online; accessed 20-September-2018].

[35] PASCANU, R., MIKOLOV, T., AND BENGIO, Y. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning* (2013), pp. 1310–1318.

[36] RUMELHART, D. E., HINTON, G. E., AND WILSON, R. J. Learning representations by back-propagating errors. *Nature 323* (1986), 533–536.

[37] TSANKOV, P., DAN, A. M., COHEN, D. D., GERVAIS, A., BUENZLI, F., AND VECHEV, M. T. Securify: Practical security analysis of smart contracts. *CoRR abs/1806.01143* (2018).

[38] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L. U., AND POLOSUKHIN, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 5998–6008.

[39] WERBOS, P. J. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks 1* (1988).

[40] WOOD, G. Ethereum: A secure decentralised generalised transaction ledger eip-150 revision (759dccd - 2017-08-07), 2017. Accessed: 2018-01-03.

[41] WU, Y., SCHUSTER, M., CHEN, Z., LE, Q. V., NOROUZI, M., MACHEREY, W., KRIKUN, M., CAO, Y., GAO, Q., MACHEREY, K., ET AL. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[42] ZĪLE, K., AND STRAZDIŅA, R. Blockchain use cases and their feasibility. *Applied Computer Systems 23*, 1 (2018), 12–20.

Notes

[1] Implementation of the LSTM vulnerability detection tool is available at https://github.com/wesleyjtann/Safe-SmartContracts.

[2] We thank Prateek Saxena and Aashish Kolluri for sharing the data.