Sound Smart Contract Security Testing with Just One Tool

Oliver Lutz
University of Würzburg

Huili Chen
University of California, San-Diego

Hossein Fereidooni
Technical University of Darmstadt

Christoph Sendner
University of Würzburg

Alexandra Dmitrienko
University of Würzburg

Ahmad Reza Sadeghi
Technical University of Darmstadt

Farinaz Koushanfar
University of California, San Diego

1 INTRODUCTION

Software vulnerabilities in smart contracts are a non-trivial problem – despite the existence of many vulnerability detection tools developed by academia and industry, the financial loss from cryptocurrency theft is enormous and amounted to $1.9 billion in 2020. The problem is that existing tools cannot provide full coverage in terms of vulnerability types they detect, and cannot be easily extended in the future, e.g., due to limitations of an underlying detection method. Hence, developers of smart contracts are advised to apply several tools to achieve sound testing, which is oftentimes neglected, as it requires non-trivial engineering effort (e.g., for setting up several tools) and significant testing time. To address this problem, we develop ESCORT, a deep learning-based vulnerability detection method that can extract knowledge about vulnerabilities from other tools and consolidate it in one solution. Using transfer learning, ESCORT is easily extensible to new vulnerability types without model re-training. Furthermore, in the transfer-learning phase, it can be trained on a little amount of data, thus addressing the prevalent dilemma of unbalanced data. We show the effectiveness of ESCORT by evaluating it on a dataset of 1.2 million smart contracts and demonstrate its high effectiveness in both, initial training and transfer learning phases, achieving a 97% F1 score on average. ESCORT is also highly efficient at inference – it takes only 0.02 sec on average to scan a smart contract, which is two orders of magnitude faster than any other existing tool. In practice, ESCORT can replace multiple vulnerability scanners and, as such, can significantly simplify security-testing procedures by smart contract developers.

2 ESCORT DESIGN

The key innovation of ESCORT is that we decompose the task of vulnerability detection into two subtasks: (T1) Learning the bytecode features of general contracts (*attack-agnostic*); (T2) Learning to identify each particular vulnerability class (*attack-specific*). To achieve (T1), we design a *feature extractor* that captures the semantic and syntactic information of contract bytecode regardless of its vulnerabilities. To perform (T2), we devise an individual *vulnerability class branch* to characterize susceptibility given the bytecode features extracted in (T1). Our **divide-and-conquer** design is highly effective and modular.

2.1 Neural Network Design

ESCORT aims to design an **extensible** DNN detector that: (i) provides the probability that the smart contract has certain vulnerabilities, instead of making a binary decision about contract's security; (ii) classifies multiple vulnerabilities using a single DNN with the consideration of vulnerability type extension. To this end, we propose a **multi-output architecture** for **concurrent** detection of multiple vulnerability types.

Figure 1 shows the generic architecture of our DNN detector. The stem and branch layers are typical DNN layers such as the Dense layer, Dropout layer, and GRU layer. The multi-output model has two main components which we discuss below:

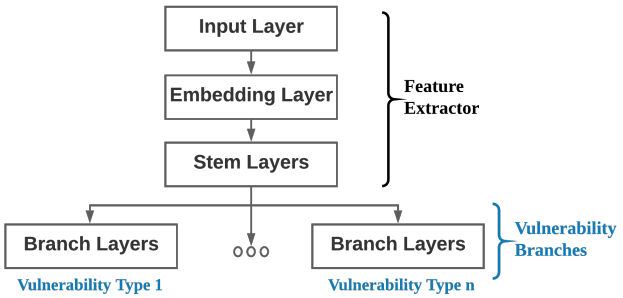

Figure 1: General topology of ESCORT's multi-output model for concurrent detection of multiple vulnerability types.

(i) Feature Extractor. The first component of ESCORT's extensible DNN model is the common feature extractor (i.e., 'stem') shared by all the bottom branches. The feature extractor is a stack of layers that aim to learn the fundamental features in the input data that are general and useful across different attributes. In the context of smart contracts, the feature extractor is trained to learn the semantic and syntactic information from the contracts' bytecode. To this end, we incorporate several key layers in ESCORT's feature extractor:

- *Embedding Layer.* The bytecode of smart contracts are long hexadecimal numbers, while DNNs typically work with fractional numbers to achieve high accuracy. The embedding can solve this discrepancy since it stores the word embedding in the numerical space and retrieves them using indices [5]. The embedding layer provides two key benefits: (i) Compressing the input via a linear mapping, thus reducing the feature dimension; (ii) Learning bytecode in the embedding space (fractional numbers). This facilitates representation exploration and gathers similar bytecode in the vicinity of each other. ESCORT leverages the advantages of the embedding layer to capture the semantics in the input bytecode.

- *GRU/LSTM Layer.* The stem layers and branch layers in Figure 1 can include GRU/LSTM layers for processing sequential inputs. Gated Recurrent Units (GRU) and Long Short-Term Memory (LSTM) are two typical layers in recurrent neural networks that help to overcome the short-term memory constraint and vanishing gradient problem [4] using a *'gating'* mechanism. More specifically, both types of layers have internal gates that regulate the information flow along the time sequences and decide which data shall be kept/forgotten. We mainly use GRU layers in ESCORT's DNN design.

(ii) Vulnerability Branches. The second component of ESCORT's multi-output DNN architecture is the **ensembling** of multiple vulnerability branches. Each branch is a stack of layers that are trained to learn the patterns/hidden representation of the corresponding vulnerability class. While there is no direct dependency between different branches, they share the same feature extractor, i.e., the input to each branch is the same. This is feasible since the branch input (which is also the feature extractor's output) shall capture the semantics in the contract's bytecode, which is common/general information useful for different vulnerabilities.

2.2 ESCORT DNN Instantiation

We instantiate a Multi-Output-Layer (MOL) DNN model based on our generic architecture explained in Section 2.1 with six branches for main model training, and then extend it with two new branches for transfer learning. Our multi-output RNN enables concurrent detection of multiple vulnerability types as discussed in Section 2.1. ESCORT learns the knowledge of multiple detection tools via supervised learning in this process.

We train the above instantiated RNN using the hyper-parameters shown in Table 1. To avoid errors caused by our large dataset and memory limitations, the entire dataset is chunked into batches and ESCORT iterates over all chunks multiple times during model training. We define '#Global_Epochs' to be the total number of times that the full training set is iterated by our DNN model and '#Local_Epochs' as the number of times that each chunk is used to update the model. Before the chunked data are passed to the model's input layer, the bytecode sequence needs to be vectorized. This is realized by a *tokenizer*, which transforms the hexadecimal data into numeric vectors. After tokenization, a hyper-parameter *MAX_SEQUENCE_LENGTH* is applied to the input vectors. Sequences shorter than this length are zero-padded, while sequences longer than this length are truncated. We empirically study the distribution of the bytecode length and find out that more than 98.5% of the contracts have length less than 4100 bytes. Hence, we set the hyper-parameter *MAX_SEQUENCE_LENGTH* to 4100 to avoid excessive padding and mislabeling[1]. Note, that both, excessive padding and mislabeling may worsen performance of the model.

The tokenized data are then passed to the DNN model for training. We assess the classification accuracy of the model after each Local Epoch (L_Ep., train on one chunk) and each Global Epoch (G_Ep., train on all chunks). The evaluation results are stored in a metric history object to keep track of the progress. In the testing phase, the remaining (unseen) data chunks are passed to the model to compute the detection metrics. At the end of the model training stage, we save the converged MOL-DNN model, the used tokenizer, and the evaluation metrics files to wrap the model as an API service.

[1]Truncated labels may end up being mislabeled if a vulnerability resides in the truncated part of the contract

Variable	Setting
#Hidden Units	GRU:128, Dense:[64,64,1]
Optimizer, Loss Function	Adam, BCE
Learning Rate	0.001
#Global_Epochs, #Local_Epochs	6, 1
Batch Size	32
MAX Seq. Length	4100

Table 1: Model Hyper-parameters.

3 EVALUATION

All of our experiments are conducted on a machine with Arch Linux OS having AMD Ryzen 9 3950X and NVIDIA GeForce RTX 3090 GPU with 32 and 24 GB of RAM, respectively. The software versions are as follows: Tensorflow 2.4.1, CUDA 11.3, NVIDIA driver 465.24.02, cuDNN 8.1.1 and kernel 5.10.32.

3.1 Dataset

To build our dataset, we collected ∼ 1.2 million smart contracts from the first 5 million Ethereum blockchain blocks and labelled them using three existing vunlerability detection tools: Oyente [3], Mythril [2], and Dedaub [1]. We consider the following eight vulnerability types in our evaluations: Callstack depth [cl. 1], money concurrency [cl. 2], accessible selfdestruct [cl. 3], DoS (unbounded operation) [cl. 4], tainted selfdestruct [cl. 5], assert violation [cl. 6], reentrancy [cl. 7], multiple sends [cl. 8]. We use the first six vulnerability types (cl. 1-6) for main model training and cl. 7-8 for transfer learning experiments.

For each vulnerability class in our main model (trained on cl. 1-6), we select at least 15.000 samples with this specific vulnerability from our raw dataset (∼ 1.2 million contracts) and concatenate them to construct the vulnerable contract set with the equally-sized distribution. We empirically set the minimal sample number to 15.000 since this is the smallest size of the well-represented vulnerability types in our dataset. The above dataset was then completed with 15.000 completely clean smart contracts where no vulnerabilities were detected.

We take 80% and 20% of the labeled dataset to construct the training and test dataset for ESCORT. Note that we set aside 10% of the training data as the validation data to prevent model overfitting.

3.2 Evaluation Results

3.2.1 Classifier Learning. We train ESCORT's model on the training set with hyper-parameters listed in Table 1.

3.2.2 Sensitivity to Training Configurations. ESCORT's training pipeline is described in Section 2.2. We use the MOL-DNN architecture with six vulnerability branches in ESCORT's training phase. This model has a total number of 87, 622 learnable parameters. We investigate whether the detection performance of ESCORT can be

Metrics (Avg.)		G_Ep.:1 L_Ep.:1	G_Ep.:2 L_Ep.:1	G_Ep.:3 L_Ep.:1	G_Ep.:6 L_Ep.:1
Training	Loss	0.11	0.09	0.08	0.05
	Precision	0.95	0.96	0.97	0.98
	Recall	0.89	0.89	0.91	0.93
	F1 score	0.91	0.92	0.94	0.95
	Hamming Loss	0.04	0.03	0.03	0.02
	Jaccard Similarity	0.86	0.87	0.89	0.92
Test	Loss	0.07	0.06	0.06	0.04
	Precision	0.98	0.98	0.98	0.98
	Recall	0.93	0.94	0.94	0.96
	F1 score	0.95	0.96	0.96	0.97
	Hamming Loss	0.02	0.02	0.02	0.02
	Jaccard Similarity	0.91	0.92	0.92	0.94
Time	Training (H:M)	3:13	6:22	9:32	19:15
	Prediction (Sec.)	0.02	0.02	0.02	0.02

Table 2: The averaged performance of our multi-label classification model across initial vulnerability types (cl. 1-6) with different G_Ep. and L_Ep. configurations.

improved by increasing the number of global epochs. The performance metrics are summarized in Table 2. ESCORT's accuracy has a slight increase with a longer training time. Since the accuracy improvement is inclining w.r.t. the minimal increase of training time, we use the last configuration (G_Ep=6, L_Ep=1) as our default setting for the rest of our evaluations.

3.2.3 Class-wise Detection Performance. We describe ESCORT's overall performance on the vulnerability types introduced in Section 3.1. Here, we provide a fine-grained insight into ESCORT's capability on each vulnerability class. Table 3 shows the class-specific metrics obtained by our MOL-DNN model. ESCORT achieves an average of 98% precision and 97% F1 score across all six vulnerability types. In the case of the *Callstack Depth* vulnerability (cl. 1), ESCORT's recall score is relatively lower than others. The low recall value indicates that ESCORT yields more false negatives on this vulnerability class.

3.3 Extensibility Performance

We perform transfer learning on the trained main model by expanding the initial architecture with two additional branches.The new branches (for cl. 7-8) have in total 8, 450 trainable parameters. For transfer learning, we use the same hyper-parameters listed in Table 1 and train the model on vulnerability types 7 and 8 simultaneously. The detection result on the new vulnerability types are shown in the last two columns of Table 3. It can be seen that ESCORT achieves 97% and 97% F1 score on the two new vulnerabilities cl. 7 and cl. 8, respectively. The empirical results corroborate that ESCORT is extensible to new attacks by enabling lightweight and effective transfer learning.

Effect of transfer learning dataset size. To further evaluate ESCORT extensibility in a more practical scenario where new smart contracts are uploaded to the blockchain in a sequential manner, we gradually reduce the dataset size of transfer learning by a certain

Metrics	Initial vulnerability types						New Vuln. Classes	
	cl. 1	cl. 2	cl. 3	cl. 4	cl. 5	cl. 6	cl. 7	cl. 8
Loss	0.06	0.06	0.03	0.04	0.01	0.02	0.07	0.08
Precision	0.98	0.93	1.00	0.97	1.00	0.99	0.95	0.98
Recall	0.90	0.96	0.97	0.96	1.00	0.97	0.98	0.95
F1 score	0.94	0.95	0.98	0.97	1.00	0.98	0.97	0.97
FPR	0.01	0.02	0.00	0.01	0.00	0.00	0.03	0.01
FNR	0.10	0.00	0.03	0.04	0.01	0.03	0.02	0.05

Table 3: Class-specific metrics (for unseen/test data) retrieved by our multi-output model for initial training phase (cl. 1-6) and after transfer learning phase (cl. 7-8).

percentage until we observe large performance degradation. We further split the smart contract samples based on time, resulting in 8.000 samples per label (cl. 7 & 8) and 23.988 samples in total. We summarize our detection results with different dataset size in Table 4. One can observe that ESCORT can effectively identify new vulnerability types even when only 400 new contracts (for cl. 7 or 8) are available for transfer learning.

Transfer Learning	100%	50%	20%	5%
F1 Score	0.97	0.96	0.96	0.95
FNR	0.03	0.04	0.04	0.02
Training Time (H:M)	3:40	1:44	0:46	0:11

Table 4: Performance comparison of transfer learning between 100%, 50% and 20% dataset size.

4 CONCLUSION

To improve the feasibility of sound security testing of smart contracts by smart contract developers, we present ESCORT, the first deep learning-based automated framework that enables lightweight Transfer Learning on new vulnerability types with minimal model modification and re-training overhead. ESCORT learns about various vulnerability types from multiple vulnerability scanners and enables effective (F1 score 97%) and efficient detection (0.02 second per contract) of all vulnerabilities it learned about in just a single scan. ESCORT does not require source code access, supports concurrent detection of multiple vulnerability types, and is the first framework that allows vulnerability type extension via transfer learning. ESCORT's multi-output RNN design is highly modular, scalable, efficient, and extensible.

REFERENCES

[1] Accessed 2021. Ethereum Contract Library by Dedaub. https://contract-library.com/

[2] Accessed 2021. Mythril. https://github.com/ConsenSys/mythril

[3] Accessed 2021. Oyente: An analysis tool for smart contracts. https://github.com/melonproject/oyente

[4] Jaeyoung Kim, Mostafa El-Khamy, and Jungwon Lee. 2017. Residual LSTM: Design of a deep recurrent architecture for distant speech recognition. *arXiv preprint arXiv:1701.03360* (2017).

[5] Hamed Zamani and W Bruce Croft. 2017. Relevance-based word embedding. In *International ACM SIGIR Conference on Research and Development in Information Retrieval.*