CS230

Reentrancy Detector for Solidity Smart Contracts

Luke Tchang
ltchang@stanford.edu

Abstract

Since 2017, the blockchain and cryptocurrency space has rapidly been gaining unprecedented levels of attention from new developers and users. While the growth of platforms like Ethereum opens up a wealth of new possibilities around decentralized technologies and programmable value, critical security flaws in smart contract code remain a major concern and have already led to the loss of hundreds of millions of dollars of funds. In this project, I implement a static analysis tool that focuses on one of the most common yet damaging attacks—reentrancy attacks. Given a contract's source code, the model I implemented seeks to flag functions vulnerable to reentrancy.

1 Introduction

In the world of Ethereum, one of the most infamous attacks is the reentrancy attack, a type of race condition where a contract function can be repeatedly called before its first invocation is finished. This type of attack leads to unintended control flow and mistimed state updates. As of now, this bug has led to the loss of approximately $50 million worth of ether.

I present a static analysis tool that, given a contract's source code, can identify functions vulnerable to reentrancy attacks. At a high level, this tool parses contract source code to extract particular keywords such as global variables and modifiers, tokenizes the source code of each function in the contract, and feeds embedded representations of these tokens through a model which classifies the function as reentrant or safe. I trained this model on a mixture of both real and augmented contract code and was able to achieve an accuracy or 98% and F1 score of 0.95 on the test set.

```
function withdraw(uint amount) public {
 ① if (credit[msg.sender] >= amount) {
 ②   msg.sender.call.value(amount)();
 ③   credit[msg.sender] -= amount;
   }
}
```

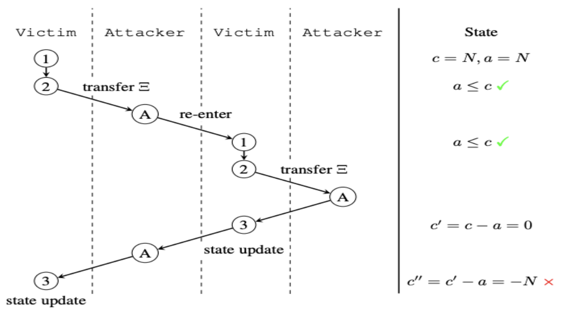


Figure 1: Sample contract function vulnerable to reentrancy attacks with control flow diagram [4].

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

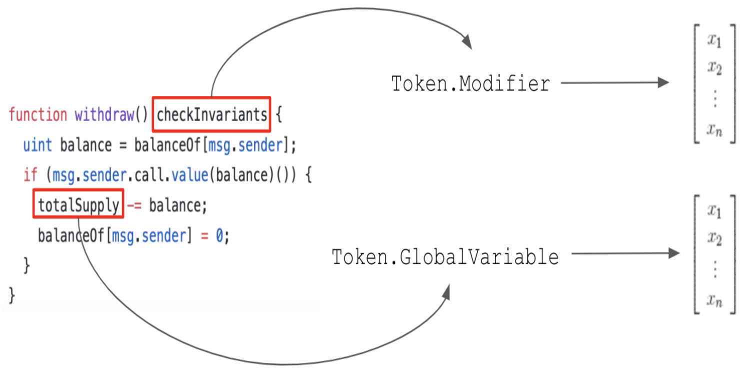
2 Datasets

The data used for this project is drawn from three separate datasets. The first dataset is a curated list of 16 unique contracts currently live on the Ethereum blockchain, each of which has one function vulnerable to reentrancy and have labels at the specific lines containing vulnerabilities [2]. The second is a list of 185 contracts used for a similar project on smart contract vulnerability detection, all of which were labeled as having or not having a reentrancy vulnerability based on the results of the Mythril static analysis tool [3]. The final dataset is a list of 50 augmented contracts which all have injected reentrancy vulnerabilities [4].

I primarily drew from the first and third datasets, as the locations of vulnerabilities were explicitly labeled for those two. Additionally, Mythril and its competitor static analysis tools were listed as only having about 30% accuracy on reentrancy attacks [5], raising questions about the ground truth accuracy of the second dataset's labels. I initially trained on around 450 function examples, 40% which were drawn from the augmented dataset, but found a mismatch between distributions of the real and augmented data. Due to this mismatch and clear overfitting to the patterns of the vulnerability injection tool, I later reduced my training set to under 200 examples and manually altered many existing examples to take on the opposite label. The test set consisted of the most realistic unaugmented data available and the final train/test split was 82/18.

3 Preprocessing

Given the limited amount of labeled data available for this particular problem, there was a substantial amount of preprocessing needed to best represent the source code. This required layering on additional logic to the pre-existing Pygments Solidity Lexer to tokenize function code in a manner more customized to the task of identifying reentrancy vulnerabilities.



```
function withdraw() checkInvariants {
  uint balance = balanceOf[msg.sender];
  if (msg.sender.call.value(balance)()) {
    totalSupply -= balance;
    balanceOf[msg.sender] = 0;
  }
}
```

Figure 2: Converting function source code into token embeddings.

Since reentrancy attacks are mainly centered around the order of global state updates and the guards put in place to halt unintended updates, the first important step was to write scripts to extract the names of global variables and modifiers (function guards) from the overall contract so they could be explicitly represented in the tokenized function representations. Additionally, extra logic was required to increase the specificity of tokens beyond what the pre-built lexer provided.

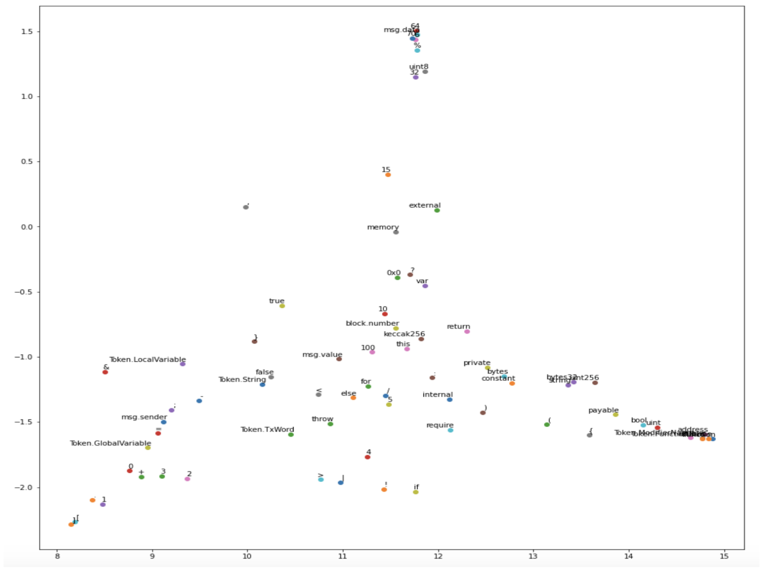


Figure 3: Word2Vec Map Representation in 2D.

After converting a contract's functions into lists of tokens, the next step was to convert tokens into embeddings. I used 15-dimensional Word2Vec embeddings, as around 15-dimensions seemed to be a common number based on other papers focused on deep learning for source code analysis [6, 7, 9]. In addition, each function representation was given a default length of 500 embeddings and was padded with additional zero-vectors when necessary.

4 Methods

4.1 Baseline BLSTM

The first approach was inspired by the work of Li, Z. et al., who fed embedded functions through a BLSTM network when building a classifier for C++ vulnerabilities [8]. When trying a similar approach using a model with one BLSTM layer and two dense layers, the model was able to achieve around 80-85% accuracy on the training set and, surprisingly, around 85-90% on the test set. It is worth noting though that despite the decent accuracy, many of the outputs seemed to indicate confusion between the two classes, as most predictions were in close range of the 0.5 threshold.

4.2 Baseline CNN with One Conv1D Layer

Shortly after my initial experimentations with the BLSTM, I switched to a basic CNN which had a 1D convolution layer, 1D max-pooling layer, and two dense layers. This change was inspired by Harer et al., who found the most success for C++ vulnerability detection with a similar TextCNN approach [7]. This brought the training accuracy up to around 95-98% and the test accuracy up to 85% after 250 epochs of training.

4.3 CNN with One Conv1D Layer and Dropout

After analyzing the examples missed from the initial baseline CNN, it became clear that the model had overfitted to the structures of the most abundant training examples, as high confidence predictions were being output on the test examples most closely reflected in the training set while unexpectedly low values were being output for positive examples that differed in structure. Adding two layers of dropout with keep probabilities of 0.4 raised the predicted probabilities of these less well-represented training examples, helped improve recall, and fix plateauing loss seen in the baseline CNN model.

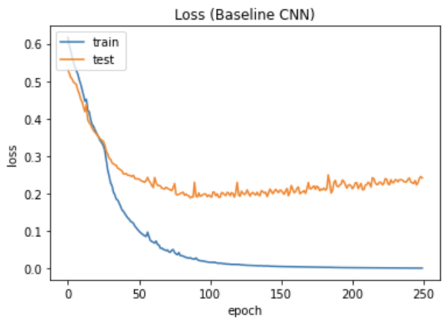


Figure 1: Figure 3: Baseline CNN Loss

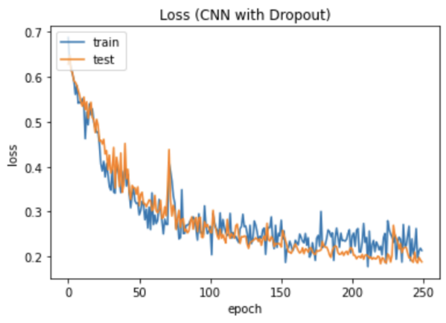


Figure 2: Figure 4: CNN with Dropout Loss

4.4 CNN with Two Conv1D Layers and Dropout

Adding a second 1D convolution layer helped improve the model's overall accuracy and recall, raising the predicted probabilities of the previously missed false negatives from around 0.4 to consistently 0.6 and above. After adding one layer of dropout, this model became the highest performing variant.

Table 1: Summary of Notable Experiments

#	Experiment	Results
1	Baseline BLSTM (BLSTM, Dense, Dense)	Accuracy: 86%, F1: 0.73, Recall: 0.89
2	Baseline CNN with One Conv1D Layer (Conv1D, MaxPool1D, Dense, Dense)	Accuracy: 93%, F1: 0.82, Recall: 0.72
3	CNN with One Conv1D Layer with L2 Regularization (Conv1D, MaxPool1D, Dense with L2 Regularization, Dense)	Accuracy: 93%, F1: 0.80, Recall: 0.67
4	CNN with One Conv1D Layer and Dropout (Conv1D, Dropout, MaxPool1D, Dropout, Dense, Dense)	Accuracy: 95%, F1: 0.90, Recall: 1.0
5	CNN with Two Conv1D Layers (Conv1D, Conv1D, MaxPool1D, Dense)	Accuracy: 97%, F1: 0.94, Recall: 0.89
6	CNN with Two Conv1D Layers and Dropout (Conv1D, Conv1D, Dropout, MaxPool1D, Dense)	Accuracy: 98%, F1: 0.95, Recall: 1.0

5 Experiments and Results

The metrics of priority for this classification problem were F1 score and recall. I put an emphasis on recall in particular because the cost of outputting a false negative prediction on a reentrant function is much higher than that of outputting a false positive.

The initial BLSTM faced a high bias problem and struggled to achieve a training accuracy over 85%. Moreover, the model's predictions, both correct and incorrect, had low confidence and were all relatively close to the 0.5 threshold. In addition, the primary benefit of the BLSTM—communicating information over longer sequences—was likely lost due to the fact that most of the reentrancy vulnerabilities in my dataset manifested themselves usually in just a few consecutive lines.

The 1D CNN approach was immediately more successful and largely solved the high bias problem that the BLSTM faced. Still, for the baseline CNN, I encountered high variance and was seeing test accuracy consistently 10% lower than the training accuracy (95% and 85% respectively). To reduce overfitting, I first tried adding L2 regularization but saw little to no improvement in the probabilities of false negatives. Adding two layers of dropout with keep probabilities of 0.4, however, increased the output probabilities of less well reflected positive examples from below 0.1 to consistently above 0.3.

Given that there was still evidence of overfitting to a structure better reflected in the training set, I added several augmented examples to my test set to check for overfitting to the augmented code generator's style. The model output near-perfect predictions on all of the augmented data. This result, combined with the fact that the augmented data was more similar to the positive examples being correctly predicted in the test set indicated overfitting to the patterns of the augmented data generator.

I then experimented with different distributions of augmented to genuine data and found that reducing the augmented/genuine split from 40/60 to 20/80 substantially improved the overfitting problem without noticeably affecting the model's performance on the realistic examples. The relatively harmless removal was likely due to the fact that the generated examples had low variability in their structure, making even a much smaller number of examples suffice to reflect the pattern provided by the generator without causing the model to overfit on the pattern. With the output probabilities of the less well-represented examples now consistently reaching 0.4, an improvement from the previous probabilities of 0.3, I lowered the positive classification threshold from 0.5 to 0.4, bringing the recall of the CNN with one convolutional layer and dropout up to 1.0 while maintaining an F1 score of 0.9.

My final experiment was to try adding a second layer of 1D convolutions to the original baseline CNN. Initially, I thought adding a second layer might worsen the overfitting problem I had been trying to address. To my surprise though, the predicted probabilities of the positive examples I had been struggling to correctly classify were bumped up to consistently greater than 0.6, suggesting that the issue might have actually been that the previous single-layer models were failing to pick up more complex features. While still correct, some of the predictions on the well represented true positives had been lowered. Adding one layer of dropout with a low drop probability of 0.3 helped mitigate this issue, leaving this model with an accuracy of 98%, F1 score of 0.95, and recall of 1.0.

6 Conclusion and Future Work

I created a processing pipeline to more carefully represent the limited data available and implemented two primary models, focusing most of my efforts on optimizing the 1D CNN variant. The CNN with two 1D convolution layers and one layer of dropout performed the best, achieving 98% accuracy, 1.0 recall, an F1 score of 0.95, and an AUC of 0.99. After struggling to accommodate both the low availability and variability of my data and experimenting with different dataset distributions, it is clear that the most important next step would be to gather more complex examples of reentrant functions or create a sufficiently random reentrant function generator. Towards the end of the project, I briefly tried writing a script to generate reentrant functions with a higher level of variability than the one used to create the augmented portion of my dataset but found the task too complicated given the time remaining. Expanding on these efforts and gathering a wider variety of training data would likely improve the flexibility of my model and bring it closer to being a practical static analysis tool for Ethereum developers.

7 Code

The preprocessing and model code as well as the dataset and labels can be found at:
https://github.com/ltchang2019/Solidity-Reentrancy-Detector

References

[1] SB Curated Data Repository. https://github.com/smartbugs/smartbugs/tree/master/dataset

[2] GNNSCVulDetector Data Repository. https://github.com/vntchain/GNNSCVulDetector

[3] SolidiFI Benchmark Data Repository. https://github.com/smartbugs/SolidiFI-benchmark

[4]Michael Rodler, Wenting Li , Ghassan O. Karame , Lucas Davi (2019). Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. ArXiv abs/1812.05934

[5] Thomas Durieux, João F. Ferreira, Rui Abreu, Pedro Cruz (2020). Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. ArXiv abs/1910.10601.

[6] Anshul Tanwar, Krishna Sundaresan, Parmesh Ashwath, Prasanna Ganesan, Sathish Kumar Chandrasekaran, Sriram Ravi. Predicting Vulnerability in Large Codebases With Deep Code Representation. ArXiv abs/2004.12783.

[7] Harer, Jacob A. et al. (2018). Automated software vulnerability detection with machine learning. ArXiv abs/1803.04497: n. Pag.

[8] Li, Z. et al. (2018). VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. ArXiv abs/1801.0168: n. pag.

[9] Nicolas Lesimple, Martin Jaggi, Una-May (2020). Exploring Deep Learning Models for Vulnerabilities Detection in Smart Contracts.