Exploring Deep Learning for Vulnerability Detection in Smart Contracts

M.S. Project

Colby Utter

Supervised by:

Dr. Scott Spetka

May 2022

Master of Science in Computer and Information Sciences

Department of Computer Science

State University of New York Polytechnic Institute

Exploring Deep Learning for Vulnerability Detection in Smart Contracts,
a project/thesis by Student Name (U00286958) ,
is approved and recommended for acceptance as a final project in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences, SUNY Polytechnic Institute.

______________________	______________________
Date	Supervisor: Scott Spetka, Ph.D.
______________________	______________________
Bruno Andriamanalimanana, Ph.D	Chen-Fu Chiang, Ph.D.

STUDENT DECLARATION

I declare that this project is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

[illegible]

Colby Utter

Exploring Deep Learning for Vulnerability Detection in Smart Contracts

Colby Utter

May 21, 2022

Contents

List of Figures

Abstract

This project explores vulnerability detection in Solidity smart contracts. The following report provides a brief overview of blockchain technology, smart contract specific vulnerabilities and the tooling that exists to detect these vulnerabilities. The application of deep learning as a vulnerability detection tool was explored in more detail. The result of this work is an LSTM trained to detect re-entrancy vulnerabilities in smart contracts. The model is trained on smart contracts identified and labeled in the ScawlD dataset provided by Yashavant et al.

1 Introduction

NIST describes a smart contract as a "collection of code and data that is deployed on the blockchain, where the contract can be executed by a user to perform a service" [3]. Smart contracts are, in short, programs that exist on a Digital Ledger Technology, i.e. Ethereum. These contracts leverage blockchain (or similar) technology to enable transactions between peers independent of a centralized authority. Today, smart contracts are responsible for millions of transactions, handling millions of U.S. Dollars and other financial assets [4].

The key properties of popular DLTs are decentralization, immutability[1], anonymity and auditability [5]. These properties amplify security concerns with respect to smart contracts. Decentralization and anonymity create a permissionless environment where any user can publish a smart contract to the blockchain and any user may be incentivized to call said contract, despite no assurance that the program being executed is free of vulnerabilities. Immutability then results in a situation where losses are solidified in the ledger. When vulnerabilities are identified immutability also prevents them from being easily patched [6] [7].

Despite the increasingly widespread use of both blockchain and smart contracts these security concerns have repeatedly instilled doubt in the space. There have been dozens of high-profile incidents where smart contract vulnerabilities resulted in financial loss. One example is the infamous DAO hack, which was made possible by a re-entrancy vulnerability and led to the loss of $40 million USD[2] [8]. Given the immutable nature of the Ethereum blockchain, this hack resulted in a hard fork of the network. Another popular example is the Parity multi-signature wallet bug, which resulted in over $200 million USD being locked in a contract [5].

Creating smart contracts free of vulnerabilities can be a difficult task. This is compounded by the effectiveness of current vulnerability detection tooling [9]. Additionally, less savvy users benefit more from automated tooling. The goal of this project was to learn about Ethereum and vulnerabilities in Ethereum smart contracts in addition to exploring the potential for deep learning applications in identifying vulnerabilities in those contracts.

2 Technology Overview

2.1 Blockchain

The inspiration for digital currency can be traced to David Chaum's DigiCash. Other projects that attempted to produce a digital currency non-exhaustively include Mondex, CyberCash, E-gold, Hashcash, Bit Gold, B-Money and Lucre. While these projects did see some success, the novelty of today's cryptocurrencies is in the trustless and permissionless environments that the blockchain data structure and consensus protocols afford.

Blockchains are tamper evident and tamper resistant digital ledgers implemented in a distributed fashion [3]. At a high level, they enable users to record transactions in a shared ledger where transactions can not be changed after they are published. The data structure itself resembles a linked list,

[1]It is not strictly true that blockchain ledgers are immutable. It is more accurate to call them tamper evident and tamper resistant given they can be modified in scenarios such as a 51% attack.

[2]Different sources report different values lost because *tokens* were stolen. The value of a token can fluctuate over time.

but there are profound differences. Figure 1 illustrates a typical blockchain.

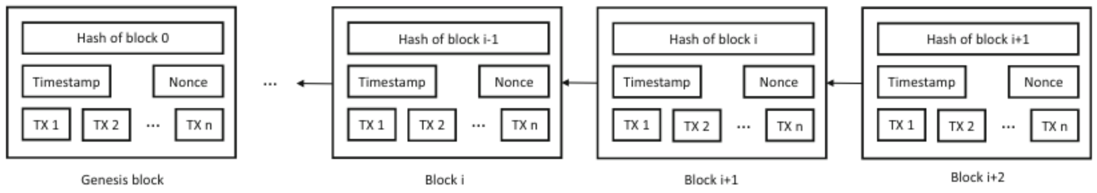


Figure 1: Typical Blockchain [1]

Instead of nodes, “blocks” are chained together (hence “blockchain”). The typical block contains transaction data, timestamp data, a nonce and a hash of the previous block. It is not uncommon for blockchains to differ in their implementations. For example, the Bitcoin blockchain’s blocks contain a merkle tree root hash and the Ethereum blockchain’s blocks contain a hash of the root of a modified merkle patricia tree[3] [10][11]. At the data structure level, tamper resistant properties are a result of each consecutive block containing a hash of the previous block. A modification to a block would create an avalanche effect in all subsequent blocks. A change made to the genesis block (the first block) would require all consecutive blocks to be mined and published again.[4] In the case of networks without a central authority, a consensus protocol prevents the network from publishing those altered blocks.

The original “crypto-currency” consensus protocol, known as Proof-of-Work (POW), was published as part of Satoshi Nakamoto’s paper “Bitcoin: A peer-to-peer Electronic Cash System”[12]. The generalized goal of a consensus protocol is to incentivize and ensure the majority of the network behaves honestly. Network participants “mine” blocks, where mining is finding the nonce that will produce a hash that meets a difficulty requirement. An example of potential difficulty requirements are the total value of the hash or the number of leading consecutive 0s in the hash. When a block is mined, it is broadcasted, a consensus is found amongst the miners, and the miner who produced the correct block first is rewarded. In this system miners are incentivized by the reward and serve to validate others if they themselves were not the first to produce the block. Since 2008, various iterations of consensus protocols have been explored and implemented. Examples of these include Algorand’s “Pure Proof-of-Stake” protocol, Cardano’s “Ouroboros” and Ethereum’s future Proof-of-stake protocol. Ethereum still uses a Proof-of-Work protocol, but is planning to “upgrade” the protocol in the future.

The sum of the blockchain data structure and consensus mechanisms amounts to persistent, anonymous and auditable public ledgers. Since Bitcoin’s conception, blockchains with various features have sprung into existence. The feature that is notable for this project is smart contract functionality.[5]

2.2 Smart Contracts

The term smart contract was coined by Nick Szabo in the early 90s. He wrote an introduction to the concept in 1994 [13]. Smart contracts are digital agreements (or programs) that are enforced by the networks they reside on. Blockchain enables decentralized transactions; as a generalization of those transactions, smart contracts enable decentralized agreements. From Szabo’s introduction, the objective of smart contracts is to satisfy common contractual conditions, like payment terms, liens, confidentiality and enforcement. The ability for smart contracts to satisfy these conditions has legal,

[3]There are actually three hashes of the trie structure included in the Ethereum block. The “stateRoot”, “transactionsRoot” and “receiptsRoot.” They are all Keccak 256-bit hashes of the root nodes where the stateRoot is of the state trie, the transactionsRoot is of the transaction trie and the recieptsRoot is of the trie containing receipts of each transaction. The modified merkle patricia tree is described on the eth.wiki.

[4]This site provides an interactive example of how this works in practice.

[5]Bitcoin does technically support smart contracts, but the functionality is quite limited in comparison to more generalized blockchain platforms.

economic and technological implications.

There is no legal meaning in the context of smart "contracts." However, from a legal perspective, smart contracts can accomplish much of what traditional paper contracts do. Many code-only contracts are enforceable under state laws governing contracts [14]. The Uniform Commercial Code holds that agreements do not always need to be in writing to be enforceable. The Uniform Electronic Transactions Act (UETA) holds electronic records and electronic signatures to the same effect as their written counterparts. In 2018 Arizona and Nevada amended their UETA to explicitly include blockchains and smart contracts. And the federal Electronic Signatures Recording Act recognizes the validity of electronic signatures and electronic records in interstate commerce. One limitation is that smart contracts, in being programs, are explicit. Natural language contracts have the ability to include catch-alls; parties may opt not to explicitly iterate every possible circumstance and instead defer to a third-party (a court, for example) in the case where a condition not included in the contract was to occur. Smart contracts alone cannot account for every possible outcome, only for those that they are instructed to account for. This is a significant in that the purpose of a contract is to minimize risk and smart contracts are handicapped in the scope of risk that they can alleviate [15].

The economic impact of smart contracts is already being realized. In the literal sense, smart contract platforms account for a significant portion of the digital ledger technology market. Ethereum alone accounts for 20% of the 1.5 trillion dollar total market cap. Another high market cap token called Uniswap is itself a smart contract and functions as a decentralized exchange (also known as an automatic market maker). Uniswap embodies the potential impacts of smart contracts on industry, serving as a decentralized format of an interaction that is ultimately between peers but has traditionally required a central authority for enforcement. Vitalik Buterin, a founder of Ethereum, has been quoted saying "Whereas most technologies tend to automate workers on the periphery doing menial tasks, blockchains automate away the center. Instead of putting the taxi driver out of a job, blockchain puts Uber out of a job and lets the taxi drivers work with the customer directly."[6] In addition to reducing the necessity of middlemen in various industries, economic impacts fall from the legal legitimacy of smart contracts.[7] The contract is the basic building block of a free market economy [16]. In under-developed nations, where a State might not exist to enforce contracts, smart contracts have the potential to enable economic collaboration and contribution.

Use cases of smart contracts extend to all digital agreements and, by way of layer 2 and layer 3 solutions, can branch into the real world. Some of the "contracts" we make day-to-day go unnoticed, like using a vending machine or receiving a reward for an action in a video game. These are examples of automated agreements. Examples of more serious use cases can be seen in the impact of smart contracts on supply chains and ownership.

Groschopf et al. reported that smart contracts enable new approaches to product origin and status tracking, supplier identity and reputation, and fraud reduction [17]. An example of the benefits of smart contract approaches can be seen in combating counterfeit medications. The World Health Organization estimates that more than 50% of medications sold online can be categorized as fake or substandard [18]. Atala SCAN in combination with EMURGO Solution (products of the Cardano blockchain ecosystem) provide a means to combat this by enabling authentication and verification of the origin of pharmaceutical products.

Nick Szabo documented the concept of "smart property" in his introduction. Today, ownership at large is being explored as an application of smart contracts. A popular example of smart contract enabled real estate ownership can be seen in Lofty AI. Lofty tokenizes properties, automating real estate investments as low as $50. As a token holder you automatically receive rental payments proportional to your stake in the estate. The token appreciates with the value of the property. The tokens can be sold at any time and token holders vote on all property decisions.

[6] In an interview with "futurethinkers" he describes the centralized portion of Uber as a reputation system and offers alternative, more decentralized, means for fulfilling that function.

[7] If not legally legitimate, at least enforced.

Vitalik Buterin described Ethereum as a platform that allows users to build decentralized applications. Before smart contracts, using a networked application often required sharing information with a centralized party that may not have had a role in the interaction otherwise. Now, users can interact in a peer-to-peer manner and rely on the decentralized network to enforce the terms of their interactions. This enables both privacy and security, removes the bloat of middle men and streamlines processes.

2.3 Ethereum Virtual Machine

Smart contracts are made possible by the Ethereum Virtual Machine (EVM).[8] The EVM can be viewed as a distributed transaction-based state machine. Blockchains are known to hold data like accounts and balances, but in Ethereum the blockchain also holds a machine state. The machine state can change from block to block according to predefined rules, not unlike how the balances of accounts would. The rules governing the state change from block to block are defined by the EVM [19] [20]. There are two types of transactions that could result in changes to the EVM state: message calls and contract creation. Contract creation results in the creation of a contract account that contains the contract bytecode. Message calls execute the bytecode associated with a contract account.

The EVM executes as a stack machine with a depth of 1024 items. Each item is a 256-bit word. The EVM maintains a transient memory during execution. The memory does not persist between transactions, but does for contracts in the form of the merkle patricia storage trie associated with the account that is part of the world state [20]. This means that smart contracts don't have a lot of execution context. They are knowledgeable of the contract calling them, their own state, and whatever information about recent blocks is contained in the storage trie.

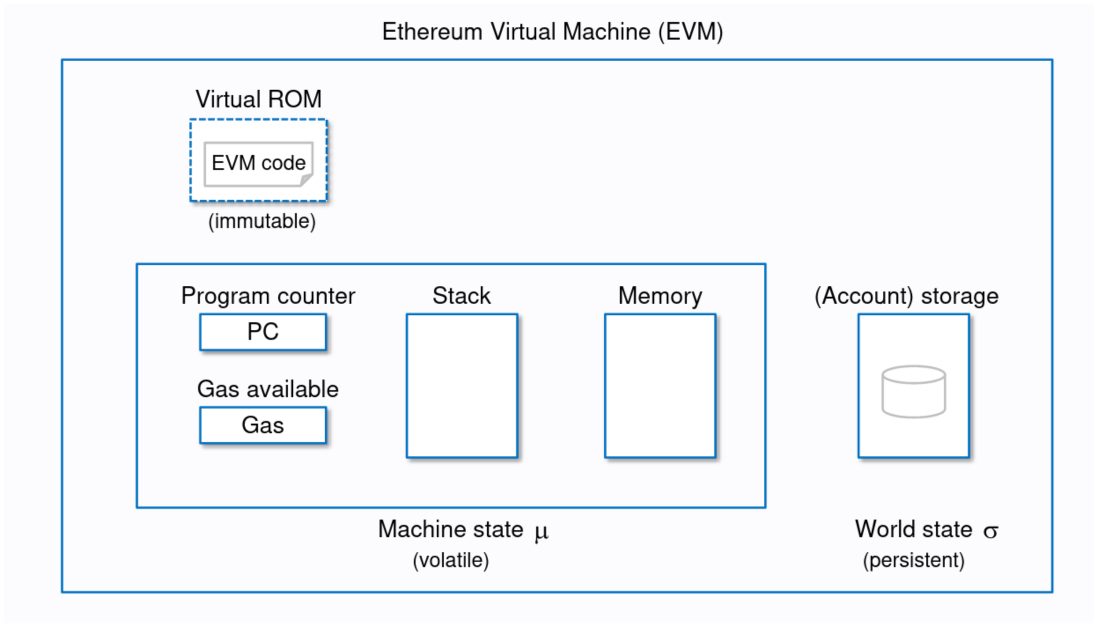


Figure 2: Ethereum Virtual Machine Architecture [2]

The EVM runs as a local instance on every Ethereum node. When a contract is called and the call is valid, the bytecode is executed on every node. Contracts have to be deterministic for consensus. Every contract has an application binary interface (ABI). The ABI is specified as a JSON array of function descriptions that inform other applications on how to interact with the contract. The ABI

[8]Ethereum is not the only platform that supports smart contracts. It is the platform that has garnered the most attention for smart contracts and consequently the platform where most smart contract research has been done. Every dataset located is Solidity or EVM Bytecode. This project focuses exclusively on Ethereum from this point forward because information regarding other blockchains would not be relevant to the goal of classifying vulnerabilities in these datasets.

and contract address are all that is needed to interact with the contract.

As is the case with most programs, smart contracts are not usually written in bytecode. Some projects do take advantage of being able to program at such a low level. For most users, though, high-level languages like Solidity are used instead. A Solidity compiler compiles smart contracts into their corresponding opcodes. When a contract is deployed, these opcodes are encoded and stored as bytecodes on the Ethereum blockchain as a space optimization. Every opcode is encoded as one byte with the exception of PUSH opcodes, which take an immediate value [21]. The result is a maximum of 256 opcodes. Less than 256 opcodes currently exist.

Contract execution starts at the beginning of the deployed bytecode. Luit Hollander's article on medium provides a good example of how the bytecode 0x6001600101 would be executed [22]. First, the bytecode would be split up into its bytes. The first instruction is 60, which is a PUSH1 instruction. The byte (01) following the PUSH1 instruction is the data to be pushed onto the stack. The next instruction (60) is again a PUSH1 instruction and again the byte (01) is pushed onto the stack. The last instruction (01) is the ADD instruction. The ADD instruction requires two items to be in the stack (in this case, 01 and 01) and, as a LIFO structure, pushes their sum (02) onto the stack.

3 Vulnerabilities in Smart Contracts

Smart contracts are prone to the same vulnerabilities as standard programs, such as integer overflows, but are also afflicted by a unique subset of blockchain specific vulnerabilities [23]. Precise definitions for these blockchain specific vulnerabilities are still being formalized. For example, the vulnerability type referred to as "unpredictable state" by Atzei et al. is referred to as "time order dependency" in other works [5].The vulnerability type referred to as "Re-entrancy" in some works is referred to as "Effectively Callback Free Objects" in other works [24].

The definitions being used here are taken from the Decentralized Application Security Project and the Smart Contract Weakness Classification Registry. The Smart Contract Weakness Classification Registry contains a total of 36 unique vulnerabilities. The Decentralized Application Security Project provides a "top 10" list of vulnerabilities, though the vulnerabilities listed are not strictly limited to smart contracts. An overview of five well-known smart contract specific vulnerabilities follows.

3.1 Re-Entrancy (SWC-107)

In solidity, the fallback function is executed on a call to the contract if none of the other functions match the given function signature or if no data was supplied and there is no receive Ether function [25]. A re-entrancy vulnerability refers to a recursive call vulnerability where a malicious contract calls back via a specially crafted fallback function before the first call and relevant check are completed [5]. If the callback function invokes a call method, the process can be repeated until gas is depleted or the call stack limit is reached. Only the last transfer of assets is reverted when gas is depleted or the call stack limit is reached. The key to this vulnerability is repeated or external calls being made before internal state changes are completed. An example of a contract that can be exploited by a re-entrancy vulnerability can be seen in the following code:

```
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    (bool success, ) = msg.sender.call.value(amountToWithdraw)("");
    require(success);
    userBalances[msg.sender] = 0;
}
```

In an attack scenario, the attacker's malicious contract would call the withdrawBalance() function again, before the account balance was set to 0. As previously mentioned, the DAO hack was made possible by a re-entrancy vulnerability. In fact, multiple re-entrancy vulnerabilities were exploited in the DAO hack, including a version of the re-entrancy vulnerability known as "Cross Function Re-entrancy." The basis of the vulnerability remains the same in the cross function format, but highlights that more than one function, or even more than one contract, may be vulnerable to re-entrancy if they share the same state [26].

3.2 Timestamp Dependency (SWC-116)

The timestamp dependence vulnerability exists because of two characteristics of the blockchain environment. First, some smart contracts need to rely on the current time. Second, miners have influence over the timestamp of a block. A vulnerable contract relies too heavily on the block's timestamp. An example of this can be seen in the following code [27]:

```
function play() public {
    require(now > 1521763200 && neverPlayed == true);
    neverPlayed = false;
    msg.sender.transfer(1500 ether);
}
```

The play() function in the above code requires that calls come after the date 1521763200, but an inclined miner could attempt to mine a block with a timestamp set in the future. A degree of timestamp variance is expected so if it is close enough it may be accepted. In the case of the code above, the miner would be able to receive the 1500 Ether before others had a chance to.

3.3 Transaction Order Dependency (SWC-114)

Transaction order dependency is colloquially referred to as the "front-running" vulnerability [5][27][28]. The front-running name falls from the vulnerability's relation to race conditions. This vulnerability is present when contract logic depends on the order of transactions executed and processed in a block. In the Ethereum network, transactions and the amount of gas attached to them are sent to each node for processing. As a result, each node can tell which transactions will occur in a given block before they actually occur. This information enables someone to submit the same transaction with a higher gas fee, "front-running" the original transaction.

3.4 Frozen Ether/Token (SWC-106)

The "Frozen Tokens" vulnerability is characterized by a contract that can receive tokens (or Ether), but has no way of sending Ether [5]. This vulnerability is often the result of reliance on external contract libraries. If an external library is modified or terminated, core functionalities may be impacted. This is the vulnerability that led to the aforementioned Parity multi-signature wallet "bug." A user was able to (and did) kill the library responsible for releasing Ether.

3.5 Authorization through tx.origin (SWC-115)

SWC-115 is an authentication problem that falls from an inappropriate usage of tx.origin. The tx.origin should never be used for authentication. The function msg.sender() should be used instead. This vulnerability stems from the functions behavior, which returns the address of the account that sent the transaction [29]. The solidity docs provide the following two contracts for clarity:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract TxUserWallet {
    address owner;
   constructor() {
       owner = msg.sender;
   }

   function transferTo(address payable dest, uint amount) public {
       // THE BUG IS RIGHT HERE, you must use msg.sender instead of tx.origin
       require(tx.origin == owner);
       dest.transfer(amount);
   }
}
```

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
interface TxUserWallet {
    function transferTo(address payable dest, uint amount) external;
}

contract TxAttackWallet {
    address payable owner;

    constructor() {
        owner = payable(msg.sender);
    }

    receive() external payable {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

Using the address of the account that sent the transaction is benign at first glance, but if the vulnerable wallet sends Ether to a malicious wallet its funds can be transferred out of the wallet. The intention of the require() function is to stop execution and throw an error when the condition isn't true. In the first contract however, the tx.origin would be the address of the attacker's contract. If the recommended msg.sender() function is used instead, the attack wallet's address would be returned (instead of the address of the account that sent the transaction) and authorization would fail [29].

All of the above vulnerabilities can be prevented without external tooling. Remediation steps are documented in the Solidity docs. Re-entrancy can be avoided by using the checks-effects-interactions pattern, avoiding external calls, handling errors for external calls, etc.. Both Timestamp and Transaction Ordering dependency vulnerabilities can be avoided by removing transaction order and time constraints from contract logic or by utilizing commit schemes. The Parity vulnerability was the result of an unprotected 'SELFDESTRUCT' instruction, which could have been protected. And remediation of SWC-115 is as simple as not handling authorization using tx.origin.

Despite this, the consensus in related work is that creating secure contracts remains a challenging task [4]. This is unfortunately empirically supported by the frequent occurrence and too often costly consequences of these vulnerabilities.

4 State of Vulnerability Detection Tools

Smart contract security analysis has seen increased attention in recent years. As of August 2021 there are at least 140 smart contract security analysis tools [5]. These tools work on a variety of inputs (EVM bytecode and high-level languages) and employ both static and dynamic methods of analysis. The formal methods being used include symbolic execution, formal verification and theorem proving, model checking and abstract interpretation. Dynamic methods include fuzzing, runtime verification, concolic testing and mutation testing. Pattern matching and statistical analysis, code instrumentation, machine learning and taint analysis tools exist.

Still, state-of-the art techniques are far from perfect [9] [30]. Durieux et al. analyzed 9 automated analysis tools on two datasets and found that only 42% of the vulnerabilities were detected by all tools.[9] Their report states that only a small number of vulnerabilities (of only two vulnerability categories) were detected simultaneously by four or more tools. This suggests that these tools are better used in conjunction. The tools analyzed include Mythril, Osiris, Oyente, Securify, Slither and Smartcheck.

None of the tools Durieux et al. analyzed utilized machine or deep learning techniques. Of the 140 smart contract security analysis tools identified in Rameder's systematic review, only 10 used a machine learning approach.

4.1 Related work:

The key advantage of deep learning applications in software vulnerability detection compared to other methods is automation. The greatest disadvantage is reportedly the lack of quality in datasets. Existing tooling is less than perfect and manually labeling smart contracts is an expertise and time intensive task. This results in somewhat of a "chicken or egg" situation.[10] In order for deep learning approaches to succeed, existing tooling needs to be able to provide ground truths.

The deep learning approach is well recognized in other cyber security domains, but there is a limited amount of work applying deep learning methods to smart contracts specifically. What work does apply to smart contracts is almost exclusively to Ethereum smart contracts. Two general approaches are seen in the related works. The first approach identifies sequential features in Ethereum Virtual Machine opcodes or bytecodes. The second approach maintains the high-level language and aims to identify semantic features.

Of the related works only Zhuang et al., Wu et al. and Tann et al. have both hosted and functional githubs. Yu et al. has an associated github, but the repository is empty aside from a README.md.

Li et al. [30] [31] [32]: found success in their deep learning-based systems for vulnerability detection. Li et al.'s work does not apply specifically to smart contracts or Solidity, but provides evidence in favor of a deep learning approach to vulnerability detection in programs. Their VulDeePecker tool and paper has influenced the work of others in this domain. The team made three major contributions in their work on VulDeePecker. First, they explore how deep learning can be applied to the task of vulnerability detection. They propose representing programs as "code gadgets", where a code gadget is a number of lines of code that are semantically related to each other that can be vectorized as input to a neural network. Second, they design and evaluate their deep learning based vulnerability detection system. VulDeePecker is able to detect 4 vulnerabilities missed by three out of four of the vulnerability detection systems they were benchmarked against. Third, they produced a dataset for evaluating deep learning-based vulnerability detection systems.

Zhuang et al. [33] [34] [35]:[11] followed in the steps of VulDeePecker and utilized the "code gadgets" of Li et al.. These works explored using graph neural networks for vulnerability detection in smart contracts. In addition to the code gadgets, they utilize an approach that involves constructing a "contract graph" to represent the semantic structure of a smart contract function. These contract graphs are experimented with as input for two separate neural networks. The first is a degree-free graph convolutional neural network (DR-GCN). The second is a temporal message propagation network. They report that their approach significantly outperforms state-of-the-art methods in detecting

[9]The vulnerabilities categories in the smart contracts analyzed included access control (tx.origin), arithmetic, bad randomness, denial of service, front running, re-entrancy, short addresses, time manipulation, unchecked low level calls and unknown unknowns.

[10]Current deep learning applications largely rely on labels generated by existing tools. If non-machine or deep learning approaches to vulnerability detection sufficiently detect vulnerabilities, the "only" advantage is efficiency.

[11]Zhuang is an author on all three papers, but not the primary author. The papers featured all utilize the basis of code gadgets.

three different types of vulnerabilities.

Qian et al. [36]: again built upon the work done by Li et al.. Their work used an abstraction called "code snippets" (not too different from "code gadgets"). A bidirectional-LSTM with an attention mechanism was chosen for a model. The model was trained to classify reentrancy vulnerabilities on a dataset of 2000 contracts. The contracts considered included call.value and were labeled manually. They report that the model achieves 88.47% re-entrancy classification accuracy.

Tann et al. [37]: used an LSTM with smart contract bytecodes as input. The dataset is derived from Google BigQuery. A dynamic analysis tool called MAIAN was used to generate labels and collect bytecodes. This model classifies smart contracts as vulnerable or not vulnerable, where vulnerable contracts are defined as suicidal, prodigal or greedy as produced by MAIAN [38]. Their evaluation reports that the model achieved 99.57% test accuracy.

Gogineni et al. [39]: experimented with a modified Long Short Term Memory network. The resulting model is an "Average Stochastic Gradient Descent Weighted Dropped Long Short Term Memory network" (AWD-LSTM). They used a pre-trained encoder instead of randomly initializing the network. The model, like Tann et al.'s model, is trained on input in the form of opcodes. One key difference is that this model was used for multi-class classification. Their experimentation concludes that the model produces acceptable results with 90% classification accuracy. The dataset used was sourced from Tann et al.

Ahsizawa et al. [7]: produced a machine learning based static analysis tool, Eth2Vec, that takes EVM Bytecode as input. The authors note that the bytecode approach is an advantage for users because high-level (Solidity) smart contract source code does not necessarily have to be shared. The team uses a natural language processing approach. They report that vulnerabilities can be detected with 77% precision. Re-entrancy vulnerabilities specifically can be detected with 86.6% precision. Eth2Vec outperformed the SVM model produced by Momeni et al. [40].

Wu et al. [41]: produced a tool called "Peculiar", which uses a pre-training technique based on crucial data flow graphs. Crucial data flow graphs are less complex data flow graphs. They report that Peculiar can achieve 91.8% precision and 92.4% recall in detecting re-entrancy vulnerabilities.

Yu et al. [42]: proposed a modularized and systematic Deep Learning-based framework to automatically detect smart contract vulnerabilities called DeeSCVHunter. Their data processing utilizes "vulnerability candidate slices" to help capture the key points of vulnerability. These slices are similar to the "code gadgets" of others. They report that extensive experiments show their framework significantly outperforms state-of-the-art methods.

Eshgie et al. [43]: combined dynamic analysis and machine learning to produce a tool called "Dynamit" that detects re-entrancy vulnerabilities in smart contracts. On the dynamic end, the tool monitors transactions on the Ethereum blockchain to acquire transaction metadata. Machine learning is then used on the transaction meta-data. The machine learning model used is a random forest classifier. They report that their model achieved more than 90 percent accuracy on the metadata of 105 transactions.

5 Datasets

A common theme amongst work related to deep learning's applications toward smart contract vulnerability detection is a lack of appropriate data. Properly annotated data is hard to come by. Labeling vulnerabilities requires expert input, time, or both. Ren et al.'s work highlights that different choices of experimental settings could significantly affect a tool's performance. The tools around vulnerability detection are limited and produce varied results. And the environment itself changes rapidly. This makes a proper dataset somewhat of a moving target.

6 potential datasets were identified for this project:

1. Zhuang et. al released the benchmark dataset used in their three papers. The dataset is unlabeled by default but tooling exists for generating labels according to the expert patterns they identified. The vulnerability labels for which this tooling exists include: time-stamp dependency, re-entrancy, integer overflow/underflow and dangerous delegate calls.

2. A smart contract vulnerability dataset containing the top 542 token contracts on coinmarketcap.com was released under the github profile AFT2020. This dataset can be opened and modified using SQLite. For each smart contract, the contract name, coinmarketcap link, etherscan token, contract links, source code, compiler version (used to compile the contract), and vulnerability label for re-entrancy attacks is provided. I can not locate an explanation as to how the vulnerability labels are generated.

3. Chen et al. released the dataset they used in their paper "Defining Smart Contract Defects on Ethereum". The full dataset contains 587 real world smart contracts. This dataset is labeled, but with respect to twenty different "contract defects." These defects do include vulnerabilities such as re-entrancy and transaction state dependency (tx.origin misuse) but are few in comparison to the other labels. For example, only 84 contracts have a re-entrancy label and only 12 contracts have a transaction state dependency label.

4. The dataset used in Ren et al.'s work was made open to the public. This dataset contains 46,186 contracts from four influential organizations. The dataset reportedly involved different code characteristics, vulnerability patterns and application scenarios. There are limited annotations. This is the original source of the ScrawlD contracts.

5. Yashavant et al. created and published the ScrawlD dataset in February 2022. ScrawlD is an annotated data set of real-world smart contracts taken from the Ethereum network. The original source of the contracts is the work done by Ren et al. (Dataset 4). Due to difficulties in the labeling process, this dataset only contains 6,780 total contracts. The datasets labels include 8 vulnerability types. Labels were produced using 5 separate tools: Slither, Mythril, Smartcheck, Oyente and Osiris. This stands out as quality methodology because my previous findings suggested that these tools be used together. The authors' goal in producing this dataset is to address the issue of a lack of standardized data existing to evaluate security analysis tools in smart contracts.

6. SmartBugs is a framework for analyzing solidity smart contracts [44]. The providers have made three smart contract datasets, annotated by Smartbugs, available to the public. The first dataset is called SB Curated and contains 144 annotated contracts. The second dataset comes from Durieux et al. and contains 47,498 contracts [9]. The third dataset contains a limited number of contracts injected with 7 different types of bugs.

6 Methodology 1

The approach taken by Li et al., while not specific to smart contracts, has found use in various later smart contract related work including Qian et al. and Zhuang et al.'s three separate papers. Taking this into account, the following factors informed my first attempt at a deep learning model for smart contracts in this project:

- Zuhang et al.'s graph neural networks maintain semantic features that may be lost in non-graphical approaches. They reported that their model outperformed state-of-the-art security analysis tooling. However, the approach in labeling taken by Zuhang et al. may produce a less robust dataset than the approach taken for the ScawlD dataset. State-of-the-art tools employ more complex tactics than Zuhang et al.'s methods and Durieux et al. reported that state-of-the-art tools still aren't perfect.

- Re-entrancy is a well defined and well known vulnerability. Despite this, a search engine query such as "re-entrancy vulnerability hacks" will result in multiple relatively recent high profile instances (instances resulting in multi-million dollar losses) of exploitation [45] [46] [47]. The re-entrancy vulnerability is not as easily avoidable as other vulnerabilities. It is much easier to avoid using tx.origin for authentication, for example.

- The ScawlD dataset contains at least 532 "confirmed" contract instances of re-entrancy vulnerabilities, where confirmed means two out of three of the tools capable of detecting the vulnerability are in consensus about the location of the vulnerability. "At least" was used here because the dataset is continuing to be updated. This is not a large dataset, but Yashavant et al.'s work implies it will provide a better ground truth than other options.

The ScrawlD data set is novel, having just been released in February, and could provide a better ground truth than the dataset used by Zuhang et al. The resulting work may shed insight on the work done by Zhuang et al.. If the model is effective, it may still be useful because re-entrancy vulnerabilities are still being exploited. And again if the model is effective, it can serve as a basis for classifying other vulnerabilities in future work.

6.1 Dataset

This project uses the ScrawlD dataset. Documentation for using the ScrawlD dataset is hosted on the ScrawlD github repository. The dataset contains annotations and their respective contract addresses. The smart contracts themselves are not part of the dataset. To overcome this, the contract source code needs to be scraped from a blockchain explorer. Etherscan is an ethereum blockchain explorer with an API for doing exactly this.

A small python program was used to separate contracts with and without re-entrancy vulnerabilities from the ScrawlD "majority_unique.txt" text file. The majority_unique.txt file contains ethereum contract addresses and the unique vulnerabilities associated with them. Unique in this context means if a vulnerability is present twice, it will only appear once.

```
allContractResultsLocation = 'scrawld_majority_unique.txt'

rentContractsLocation = 'RENTContracts.txt'
NoRENTContractsLocation = 'NoRENTContracts.txt'
RENTContract = []
NoRENTContract = []

textfile1 = open(rentContractsLocation, "w")
textfile2 = open(NoRENTContractsLocation, "w")


with open(allContractResultsLocation) as f:
    lines = f.readlines()
    for line in lines:
        newLine = line.split()
        for token in newLine:
            if token == 'RENT':
                RENTContract.append(newLine[0])

for element in RENTContract:
    textfile1.write(element + "\n")
textfile1.close

f.close()

with open(allContractResultsLocation) as d:
    lines = d.readlines()
    for line in lines:
        newLine = line.split()
        addressMatch = False
        for contractAddress in RENTContract:
            if (contractAddress == newLine[0]):
                addressMatch = True
        if addressMatch == False:
            NoRENTContract.append(newLine[0])


for element in NoRENTContract:
    textfile2.write(element + "\n")
textfile2.close

d.close()
```

An additional smaller python program was used to convert the separated contract list .txt files into comma separated values. Though trivial, this code is included for following readers.

```
import csv

rentContractsLocation = 'RENTContracts.txt'
NoRENTContractsLocation = 'NoRENTContracts.txt'
rentContractsDestination = 'RENTContractsNOSOL.csv'
NoRENTContractsDestination = 'NoRENTContractsNOSOL.csv'

textfile1 = open(rentContractsDestination, "w")
textfile2 = open(NoRENTContractsDestination, "w")

replacementLine = []
with open(rentContractsLocation) as f:
    lines = f.readlines()
    for line in lines:
        replacementLine.append(line.replace("_ext.sol", ""))
        print(replacementLine)

for element in replacementLine:
    textfile1.write(element)
textfile1.close

f.close()

replacementLine = []
with open(NoRENTContractsLocation) as d:
    lines = d.readlines()
    for line in lines:
        replacementLine.append(line.replace("_ext.sol", ""))
        print(replacementLine)

for element in replacementLine:
    textfile2.write(element)
textfile2.close

d.close()
```

The scripts ScrawlD provided include a python program for scraping the contract source code from Etherscan. API key(s) are required so an account has been created. The ScrawlD script being used is named ExtFromEtherScan.py and is located in the scripts folder of the repository. The only modifications made to this program are to the file location and API key. This scraping script is used twice; once to build a collection of re-entrancy vulnerable contract source code and again to build a collection of contract source code that is not vulnerable to re-entrancy. The Etherscan API does not return the source code directly. Instead, it returns a .json with the following structure:

```
{
      status   :   1   ,
      message   :   OK   ,
      result  : [
       {
             SourceCode  :
              ABI  :
             ContractName  :
             CompilerVersion  :
             OptimizationUsed  :
              Runs  :
             ConstructorArguments  :
             EVMVersion  :
             Library  :
             LicenseType  :
             Proxy  :
             Implementation  :
             SwarmSource  :
       }
    ]
}
```

The following code was used to manipulate the .json replies from Etherscan into solidity source files (.sol).

```
NoRENTContractsLocation = 'NoRENTContractJSONs'
NoRENTContractsDestination = 'NoRENTContractSOLs'

for filename in os.listdir(NoRENTContractsLocation):
    with open(os.path.join(NoRENTContractsLocation, filename), 'r') as f:
        json_object = json.load(f)
        print(json_object['result'][0]['SourceCode'])
        time.sleep(.01)
        filename = filename.replace('_ext.json', '.sol')
        with open(os.path.join(NoRENTContractsDestination, filename), "w") as outfile:
            outfile.write(json_object['result'][0]['SourceCode'])


outfile.close()
f.close()
```

The resulting file directory is two folders: RENTContractSOLs and NoRENTContractSOLs. Each folder contains the source code of the respective contracts saved as solidity source files. The resulting dataset is imbalanced. Out of a total of 5664 contracts there are only 654 reentrancy vulnerable contracts in the resulting dataset.

6.2 Pre-Processing

Pre-processing for the graph based neural network involves roughly four steps:

1. Comment Removal - Solidity programs are heavily commented, but the comments aren't likely to present features that can be made use of, or at least aren't being made use of in this project.

2. Normalizing the contracts - The GNNSCVulDetector github repository contains a tool for creating a graph representation of each contract. To use the tool the contracts need to be stripped of white space and formatted to spec. No tools for formatting contracts are provided.

3. Producing the contract graphs - The GNNSCVulDetector repository contains a tool for this process. The tool is called "AutoExtractGraph.py"

4. Converting the contract graphs into vectors - The tool "graph2vec.py" is included in the GNNSCVulDetector repository.

The first preprocessing step is to remove the comments from the solidity smart contracts. The remove_comment.py script located in the tools directory of the GNNSCVulDetector repository was adapted to do so.

The next step is to normalize the contract files. This pre-processing step was partially completed using the Prettier plugin for Solidity. Normalizing here means:

- Function name, parameters, and return values are on one line.
- White space is minimized as much as possible.

The next step is to produce the contract graphs. The process for producing these graphs is documented in Zhuang et al.'s paper "Smart Contract Vulnerability Detection Using Graph Neural Networks". It was learned through experimentation that the provided tool is impressive, but will not be suitable for the ScrawlD dataset.

The first process of the 'AutoExtractGraph.py' tool identifies major nodes, where major nodes are nodes that have some feature. In the case of classifying reentrancy vulnerabilities, the major nodes are nodes which contain the call.value function. Using this tool on my reentrancy vulnerable contracts only works for a limited number of them.

I believe the problem is (at least in part) that re-entrancy is identified by the existence of "call.value()". Call.value() is only one part of the reentrancy vulnerability. As covered in the "Vulnerabilities in Smart Contracts" section on reentrancy vulnerabilities, the issue more aptly lies in the practice of updating program state (the balance of an account, for example) after potentially transferring control to another contract. For example, a contract using send() or transfer() is still at risk [48]. This has two implications. First, some of the results in the related works may be able to be improved. Second, my dataset, which has not been restricted to recognizing re-entrancy as requiring "call.value()", does not merge well with the above project's tooling. This limitation exists in Zhuang et al.'s code snippet generation for the ReChecker model and the work reviewed by Qian et al.

To confirm this hypothesis I turned to ScrawlD and took a sample of contracts. The vulnerabilities.json file included in the dataset contains vulnerabilities in each smart contract as reported by the 5 tools used. The following output confirms that both Mythril and Slither identified contract functions that were classified as reentrancy in contract 0x0000000000b3f879cb30fe243b4dfee438691c04_ext.sol, but it is not processed by AutoExtractGraph.py. More information is included in the vulnerability.json than is shown below for this contract. Excess information has been removed for clarity. Figure 3 shows an example of a JSON output from the ScrawlD github and is included here to inform the JSON itself.

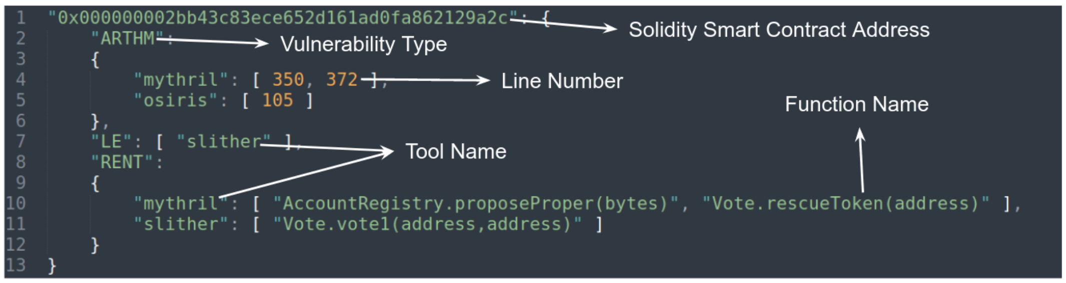

```
"0x000000002bb43c83ece652d161ad0fa862129a2c": {
    "ARTHM":
    {
        "mythril": [ 350, 372 ],
        "osiris": [ 105 ]
    },
    "LE": [ "slither" ],
    "RENT":
    {
        "mythril": [ "AccountRegistry.proposeProper(bytes)", "Vote.rescueToken(address)" ],
        "slither": [ "Vote.vote1(address,address)" ]
    }
}
```

Figure 3: Example ScrawlD JSON Output

```
"0x0000000000b3f879cb30fe243b4dfee438691c04_ext.sol": {
    "RENT": {
        "mythril": [
            "GasToken2.free(uint256)",
            "GasToken2.freeUpTo(uint256)"
        ],
        "slither": [
                "GasToken2.free(uint256)",
                "GasToken2.freeFrom(address,uint256)",
                "GasToken2.freeFromUpTo(address,uint256)",
                "GasToken2.freeUpTo(uint256)"
            ]
        }
    }
```

For each contract sampled the contract source code includes the identified functions. In contract 0x0000000000b3f879cb30fe243b4dfee438691c04.sol for example, GasToken2.free(), GasToken2.freeUpTo() and so on are consistent with the Solidity source code. To show the other half of this, the AutoExtractGraph tool did work on contract 0x0045684552109f8551cc5c8aa7b1f52085adff47.sol. The ScrawlD reported vulnerabilities file has not been updated to include the JSON associated with this contract. However, the call.value function is present in the source code on line 201.

There is an opportunity to combine the "code gadget" idea with the ScrawlD dataset. The dataset contains the line numbers of certain vulnerabilities, like the arithmetic vulnerability in the picture above. Functions are identified for reentrancy vulnerabilities. If the graph extraction tool was adapted the vulnerable snippet could be extracted precisely instead of relying on a pre-defined feature. This work was not done here.

7 Methodology 2

Zhuang et. al's work is not suitable for using the ScrawlD dataset unless the work above is done, yet the ScrawlD dataset could be the strongest (ground truth) of the datasets located. Given this, that Tann et al.'s LSTM approach doesn't rely on specially crafted tooling, and the reported success of LSTMs in vulnerability detection on bytecode; exploring Tann et al.'s work with the ScawlD dataset seems a step in the right direction.

7.1 Dataset

An LSTM is a sequence learning model and Tann et al.'s work demonstrates EVM bytecode sequences as input. In Tann et al.'s work, both labels and EVM bytecode are generated using MAIAN.

I made an attempt to use MAIAN for this project, but it was unnecessary given the ScrawlD dataset is already labeled. Both the setup and use of MAIAN was labored. The original MAIAN github repository is dated but SmartBugs has made an effort to maintain it. After some failure to get it running on a Fedora distro, I was able to get the SmartBugs version running on a virtual machine with Ubuntu 20.04 LTS. Five main dependencies needed to be installed: Go Ethereum (Geth), the Solidity compiler, Z3 Theorem prover, web3 and PyQt5. MAIAN was useful for exploration, but only one Solidity contract in my dataset was analyzed by MAIAN. The Solidity compiler version needs to match the contract being compiled (of course, but this is time consuming). Given the manual input and the nature of the tool (dynamic analysis, a private ethereum blockchain is used to deploy and analyze the compiled contracts), it would have taken an exorbitant amount of time to retrieve the EVM bytecode for my data in this way. Moreover, my goal is to classify re-entrancy vulnerabilities. MAIAN classifies contracts as prodigal, suicidal and greedy.

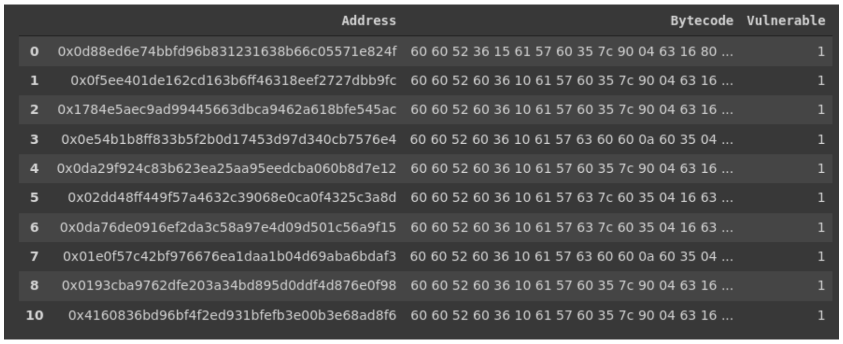
The bytecode associated with a contract address is available on the Ethereum blockchain. Etherscan does not have an api for gathering this data so web3.js was used. Web3.js is a collection of modules that contain specific functionality for the ethereum ecosystem. The web3.js library was well documented and worked as expected. I am not currently running my own node so opted to use a free service called infuria for access to blockchain data. I wrote the following script and used it twice, once for the directory containing reentrancy vulnerable contracts and again for the directory with non-vulnerable contracts.

```
const Web3 = require('web3')
const rpcURL = "https://mainnet.infura.io/v3/2d4fc1565c3940c98120d7a47b7cec2c"
const web3 = new Web3(rpcURL)
const fs = require('fs')
const dir ='DLSmartContracts/ProjectData/NoRENTContractSOLs'
const path = require('path')

const files = fs.readdirSync(dir)

var get_byte_code = async (web3, contractAddress) => {
  var byteCode = await web3.eth.getCode(contractAddress);
  console.log(byteCode)
  fs.writeFile(("NoRENTContractByteCode/" + contractAddress + ".txt"), byteCode, (err) => {if
      (err) throw err;})
  return byteCode;
};

for (const file of files) {
    var contractAddress = path.parse(file).name;
    console.log(contractAddress)
    get_byte_code(web3, contractAddress)
    console.log("success")
}


console.log(get_byte_code(web3, contractAddress));
```

The result of this process was two directories (vulnerable and not vulnerable) each containing the EVM bytecode for their respective contracts. I intended to use pandas so I used this next script to generate a csv containing the address, bytecode and vulnerability label of every contract.

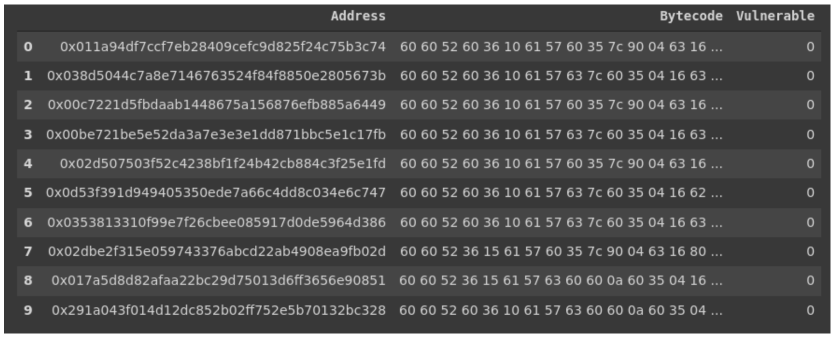
```
rentContractsLocation = 'NoRENTContractByteCodeALL/'
#rentContractsDestination = 'inputdata'
d = open("inputdata/NoRENT_contractsALLNOSCUFF.csv", 'w')
writer = csv.writer(d)

header = ['Address', 'Bytecode', 'Vulnerable']
writer.writerow(header)
row = []

for filename in os.listdir(rentContractsLocation):
    with open(os.path.join(rentContractsLocation, filename), 'r') as f:
        bytecode = f.read()
        bytecodespace = ' '.join(bytecode[i:i+2] for i in range(0, len(bytecode), 2))
        time.sleep(1) #have to sleep here because of jupyter notebook IO rate cap
        row.append(filename.replace('.txt', ''))
        row.append(bytecodespace)
        row.append('0')
        print(row)
        writer.writerow(row)
        row = []

d.close()
f.close()
```

A sample of the resulting contract .csvs loaded as pandas dataframes, retrieved by using dataframe.head(10), is included here as Figure 4 and Figure 5. When the vulnerable column contains 1 it is signifying the presence of a reentrancy vulnerability in the contract. The dataframe containing non-vulnerable contracts is labeled 0.



	Address	Bytecode	Vulnerable
0	0x0d88ed6e74bbfd96b831231638b66c05571e824f	60 60 52 36 15 61 57 60 35 7c 90 04 63 16 80 ...	1
1	0x0f5ee401de162cd163b6ff46318eef2727dbb9fc	60 60 52 60 36 10 61 57 60 35 7c 90 04 63 16 ...	1
2	0x1784e5aec9ad99445663dbca9462a618bfe545ac	60 60 52 60 36 10 61 57 60 35 7c 90 04 63 16 ...	1
3	0x0e54b1b8ff833b5f2b0d17453d97d340cb7576e4	60 60 52 60 36 10 61 57 63 60 60 0a 60 35 04 ...	1
4	0x0da29f924c83b623ea25aa95eedcba060b8d7e12	60 60 52 60 36 10 61 57 60 35 7c 90 04 63 16 ...	1
5	0x02dd48ff449f57a4632c39068e0ca0f4325c3a8d	60 60 52 60 36 10 61 57 63 7c 60 35 04 16 63 ...	1
6	0x0da76de0916ef2da3c58a97e4d09d501c56a9f15	60 60 52 60 36 10 61 57 63 7c 60 35 04 16 63 ...	1
7	0x01e0f57c42bf976676ea1daa1b04d69aba6bdaf3	60 60 52 60 36 10 61 57 63 60 60 0a 60 35 04 ...	1
8	0x0193cba9762dfe203a34bd895d0ddf4d876e0f98	60 60 52 60 36 10 61 57 60 35 7c 90 04 63 16 ...	1
10	0x4160836bd96bf4f2ed931bfefb3e00b3e68ad8f6	60 60 52 60 36 10 61 57 60 35 7c 90 04 63 16 ...	1

Figure 4: Sample of Vulnerable Dataframe



	Address	Bytecode	Vulnerable
0	0x011a94df7ccf7eb28409cefc9d825f24c75b3c74	60 60 52 60 36 10 61 57 60 35 7c 90 04 63 16 ...	0
1	0x038d5044c7a8e7146763524f84f8850e2805673b	60 60 52 60 36 10 61 57 63 7c 60 35 04 16 63 ...	0
2	0x00c7221d5fbdaab1448675a156876efb885a6449	60 60 52 60 36 10 61 57 60 35 7c 90 04 63 16 ...	0
3	0x00be721be5e52da3a7e3e3e1dd871bbc5e1c17fb	60 60 52 60 36 10 61 57 63 7c 60 35 04 16 63 ...	0
4	0x02d507503f52c4238bf1f24b42cb884c3f25e1fd	60 60 52 60 36 10 61 57 60 35 7c 90 04 63 16 ...	0
5	0x0d53f391d949405350ede7a66c4dd8c034e6c747	60 60 52 60 36 10 61 57 63 7c 60 35 04 16 62 ...	0
6	0x0353813310f99e7f26cbee085917d0de5964d386	60 60 52 60 36 10 61 57 63 7c 60 35 04 16 63 ...	0
7	0x02dbe2f315e059743376abcd22ab4908ea9fb02d	60 60 52 36 15 61 57 60 35 7c 90 04 63 16 80 ...	0
8	0x017a5d8d82afaa22bc29d75013d6ff3656e90851	60 60 52 36 15 61 57 63 60 60 0a 60 35 04 16 ...	0
9	0x291a043f014d12dc852b02ff752e5b70132bc328	60 60 52 60 36 10 61 57 63 60 60 0a 60 35 04 ...	0

Figure 5: Sample of Not Vulnerable Dataframe

Index 9 of Figure 4 is missing. Both vulnerable and not vulnerable frames included some NaN values in the bytecode columns. These were removed from the datasets after they were loaded. An example of this process:

```python
import pandas as pd
import numpy as np

dataset = '/content/drive/MyDrive/SCData10/NoRENT_bytecodeALL.csv'
not_vuln_data = pd.read_csv(dataset, usecols =['Address', 'Bytecode', 'Vulnerable'])
not_vuln_data = not_vuln_data.dropna()
not_vuln_data.head(10)
```

7.2 Pre-processing

Pre-processing under Tann et al.'s work is more straight-forward. The following steps were outlined in the publication:

1. The bytecode is separated every two characters so that it can be vectorized.
2. The bytecode is tokenized. The tokenization process amounts to converting each "word" to an integer, as neural networks require numerical data. This was done using the Keras Tokenizer.
3. Sequences are then generated using the Keras texts_to_sequences method. A sequence length has to be chosen. In my case, lengths from 1,000 to 10,000 were experimented with.
4. The ScawlD dataset is not balanced. Following Tann et al.'s work, SMOTE was used to oversample the vulnerable contracts [49].

During these pre-processing steps it became clear that there were more differences between the ScrawlD dataset and Tann et al.'s dataset than previously expected. The most notable of these differences was the length of the bytecodes (encoded as strings) associated with the contracts in the datasets. After adding spaces every two characters of bytecode (so that the bytecode can be tokenized), an overwhelming majority of Tann et al.'s contracts are less than 10,000 characters and the largest outliers are 20,000. The ScrawlD dataset includes contracts up to lengths of 75,000. Worse, a good portion of the longer contracts are of the lesser represented class. Figures 6 and 7 provide a histogram of contract counts and their respective lengths for the ScrawlD dataset. Figure 8 provides a histogram of contract counts and their respective lengths for the dataset used by Tann et al.

It can also be seen, comparing Figures 6 and 7 with Figure 8, that Tann et al.'s dataset has significantly more samples than the ScrawlD dataset. The size difference of the datasets does not come as a surprise. In many of the works reviewed datasets of hundreds of thousands of contracts were used. A handful of the works reviewed were able to find success with contracts in the thousands.

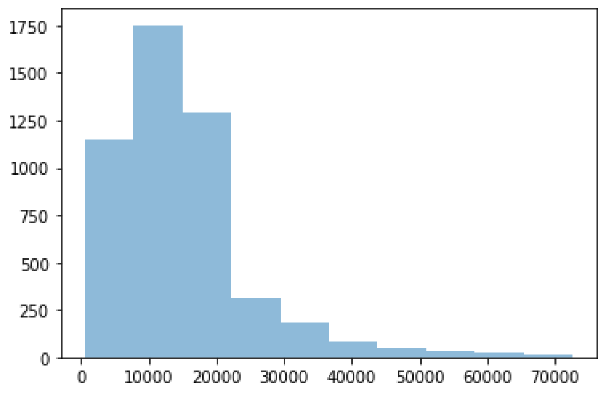


Figure 6: ScrawlD Dataset Not-Vulnerable Contract Length

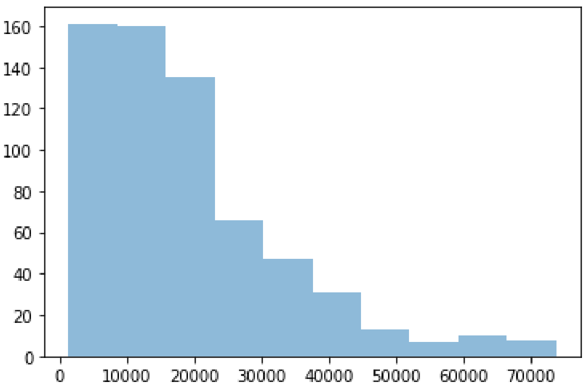


Figure 7: ScrawlD Dataset Vulnerable Contract Length

Some time was spent analyzing my contracts and scraping process. I eventually took the longest contracts in both the non re-entrancy vulnerable dataset and the re-entrancy vulnerable dataset and found them on Etherscan. The contract addresses 0x05c848E3547Bc3Ccd977B84140FDC917Bfff96a1 and 0xa1ad52ff49fc70b7920b7d02483a58beb15d492b contain the same bytecode I had scraped. The bytecode on Etherscan will include both "creation-bytecode" and "deployed-bytecode" when a contract's high-level source code is verified. Both of these contracts are verified, so the bytecode listed is not an exact match but the deployed-bytecode is present after the creation-bytecode. The bytecode can also be decompiled into opcodes on Etherscan. The dates these contracts were published are both after EIP-170 so it is not possible that they existed prior to the contract size limit. Both contracts are in fact less than the 24KB contract size limit. The bytecode associated with the contract address 0x05c848E3547Bc3Ccd977B84140FDC917Bfff96a1 is 23,880 bytes. In investigating this, I did learn about ways for contracts to get around the maximum contract size limit. One of these methods is

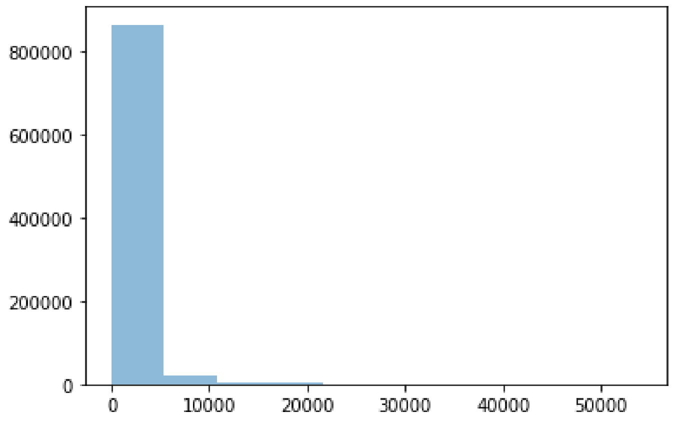


Figure 8: Tann et al. Dataset Contract Length

called the "Diamond Pattern" [50]. Additionally, I learned the purpose of the contract size limit (implemented as EIP-170) and of the resulting debate.

EIP-170's contract size limit was imposed to prevent denial-of-service attacks. Calls to contracts are relatively cheap to make, but the impact of a call increases disproportionately to the contract's code size. This creates a situation where a mal-intended user could cheaply create significant work for the nodes executing the contracts. On the other side of the debate, users argue that application complexity is stifled [51]. EIP-170 being implemented is evidence that the majority of users prefer having a reliable network over a network that supports complex apps but is unreliable. There are a number of work-arounds for reducing the bytecode size of contracts: contracts can be separated into smaller contracts, libraries can be used, proxy systems, removing functions, shortened error functions, etc.

One variable impacting length that I am able to account for is the values immediately following bytecode that is decompiled into PUSH opcodes. PUSH opcodes range from PUSH1 to PUSH32. The integer in the instruction corresponds to the byte size of the value that is being pushed onto the stack. A PUSH1 instruction will push a 1 byte value onto the stack and a PUSH32 instruction will push a 32-byte value onto the stack. Manually checking Tann et al.'s dataset, it appears these values have been removed. No explicit mention of them being removed exists in their paper so I assume this was as a result of the MAIAN process. I am unsure as to whether they might be useful features so will remove them and experiment with both datasets.

To remove these values, each contract was scanned for the following instructions: 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F. When one of these instructions was encountered the appropriate number of characters was removed from the proceeding text. For example, if 7F was encountered, the next 64 characters would be removed. The code used to do this could surely be optimized, but for the size of my dataset it was acceptable. The values removed are in line with the decompilation tools of Etherescan. After running the following code, I compared a few of the overlapping contracts in the datasets to be sure the code didn't remove potentially valuable features. The contracts were not exact at points, but were very close. This process significantly reduced the length of my contracts.

```
import os
import time

opcodes = ['60', '61', '62', '63', '64', '65', '66', '67', '68', '69', '6a', '6b', '6c',
    '6d', '6e', '6f', '70', '71', '72', '73', '74', '75', '76', '77', '78', '79', '7a',
    '7b', '7c', '7d', '7e', '7f']
```

```
mapping = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
    23, 24, 25, 26, 27, 28, 29, 30, 31, 32]
mapping1 = [i * 2 for i in mapping]

def remove(operation, bytecode, location):
    index0 = opcodes.index(operation)
    target_value = bytecode[location+2: (location+2)+mapping1[index0]]
    target_str_len = len(target_value)
    new_bytecode = bytecode[:(location+2)] + bytecode[(location+2)+target_str_len:]
    #removed_value = target_value
    new_length = len(new_bytecode)
    return removed_value, new_bytecode, new_length

count = 0
for filename in os.listdir(rentContractsLocation):
    with open(os.path.join(rentContractsLocation, filename), 'r') as f:
        bytecode = f.read()
        removed_values = []
        #bytecode = file.lower()
        length = 0
        i = 2
        bytecode_length = len(bytecode)
        while i <= bytecode_length:
            byte = bytecode[i:i+2]
            #print(byte)
            if opcodes.count(byte) > 0: #if it's a PUSH opcode
                removed_value, bytecode, new_length = remove(byte, bytecode, i)
                #removed_values.append(removed_value) #commented out for performance, 6k
                    contracts of removed values is a lot
                bytecode_length = new_length
                i+=2
                time.sleep(.01)
                print("working on contract", count)
            else:
                i+=2

        new_filename = os.path.join(rentContractsDestination, filename)
        d = open(new_filename, "w")
        d.write(bytecode)

    print('Finished contract', count)
    count+=1
    d.close()
    f.close()
```

The length of the contracts after this step are significantly reduced, but still larger than the dataset used by Tann et al. Further investigation of where these differences occur needs to be done. Figures 9 and 10 show the same histograms after PUSH values were removed.

In decompiling the longer contracts I could also see that some of the instructions included were not valid. This could simply be a result of modifications to the EVM or high-level compilers. A dictionary was manually created to be used when tokenizing as a resolution. The dictionary includes the following bytecodes (each corresponding to valid opcodes):

00 01 02 03 04 05 06 07 08 09 0A 0B 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 20 30 31 32 33 34 35 36 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 50 51 52 53 54 55 56 57 58 59 5A 5B 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 96 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 F0 F1 F2 F3 F4 F5 FA FD FE FF

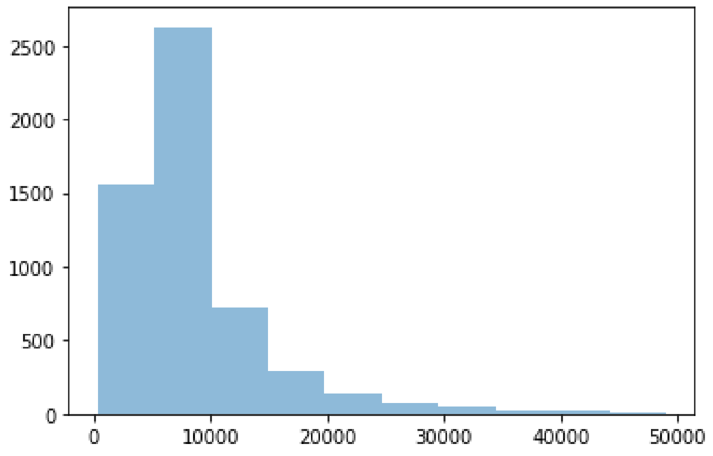


Figure 9: ScrawlD Not-Vulnerable Contract Lengths with PUSH Values Removed

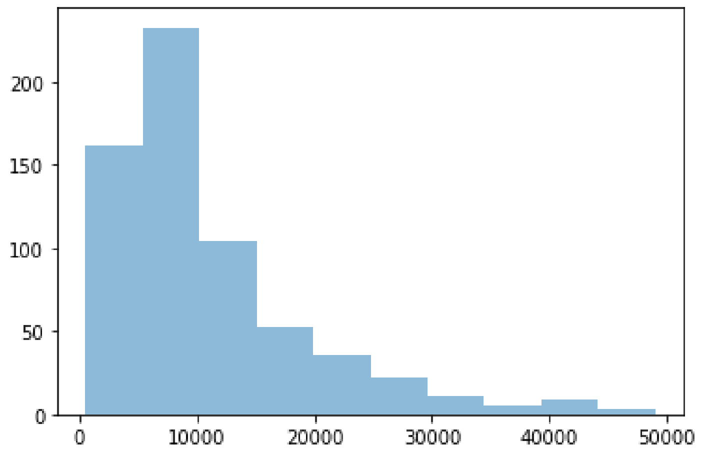


Figure 10: ScrawlD Vulnerable Contract Lengths with PUSH Values Removed

Various train, test and validation splits were experimented with. The general split used was 60% training data, 20% validation data and 20% test data. An example of the code used to perform this split in the case of vulnerable data for the LSTM is as follows:

```
train_ratio = .6
test_ratio = .2
validation_ratio = .2

vulnerable_train_count = round(len(vulnerable_data_shuffled) * train_ratio)
vulnerable_test_count = round(len(vulnerable_data_shuffled) * test_ratio)
vulnerable_validation_count = round(len(vulnerable_data_shuffled) * validation_ratio)

validation_data_count = vulnerable_train_count + vulnerable_test_count


vulnerable_train_data = vulnerable_data_shuffled.iloc[0:vulnerable_train_count]
vulnerable_test_data =
    vulnerable_data_shuffled.iloc[vulnerable_train_count:(vulnerable_train_count+vulnerable_test_count)]
vulnerable_validation_data = vulnerable_data_shuffled.iloc[validation_data_count:]

print('Total Vulnerable Train Data', len(vulnerable_train_data))
print('Total Vulnerable Test Data', len(vulnerable_test_data))
print('Total Vulnerable Validation Data', len(vulnerable_validation_data))
```

Other pre-processing steps such as work with data frames, sequence length adjustments, padding (adding 0s to the end of sequences)[12], and tokenizing was done with the following:

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

dictionary_data = '/content/drive/MyDrive/SCData10/opdict.txt'

dictionary = pd.read_csv(dictionary_data)

def label(df):
    df['LABEL'] = 0
    df.loc[df['Vulnerable'] == 1, 'LABEL'] = 1
    df.loc[df['Vulnerable'] != 1, 'LABEL'] = 0

def preprocess(df):
    n_most_common_words = 141
    max_len = 5000
    tokenizer = Tokenizer(num_words=n_most_common_words, lower=False)
    tokenizer.fit_on_texts(dictionary)#df['Bytecode'].values)
    sequences = tokenizer.texts_to_sequences(df['Bytecode'].values)
    word_index = tokenizer.word_index
    print('Found %s unique tokens.' % len(word_index))
    X = pad_sequences(sequences, maxlen=max_len, padding="post")
    return X

def dftoXY(df):
    X_test = preprocess(df)
    label(df)
    print(pd.value_counts(df['LABEL']))
    y_test = to_categorical(df['LABEL'], num_classes=2)
    return X_test, y_test

def XandY(posdf, negdf):
    dfset = pd.concat([posdf, negdf])
    dfset = dfset.sample(frac=1, random_state=39, replace=False)
    dfset.loc[dfset['Vulnerable'] == 1, 'LABEL'] = 1
    dfset.loc[dfset['Vulnerable'] != 1, 'LABEL'] = 0

    X, y = dftoXY(dfset)

    print('Shape of X: {}'.format(X.shape))
    # for sm.fit_sample
    y_labels = np.expand_dims(np.array(np.argmax(y, axis=1)), axis=1)
    y_labels = np.expand_dims(np.array(np.argmax(y, axis=1)), axis=1)
    print('Shape of y: {}'.format(y_labels.shape))

    return X, y_labels
```

Both oversampling and undersampling with imbalanced learn libraries were experimented with. Synthetic Minority Over Sampling (SMOTE) and standard under sampling of the majority class was used to reduce the dataset imbalance. A combination approach was found to perform best. Figure 11 depicts the class imbalance before oversampling and undersampling. Figure 12 depicts the class imbalance after oversampling and undersampling. The distributions in these figures are not necessarily the highest performing. Different sampling ratios need to be explored in more depth.

[12]The embedding layer requires constant lengths.

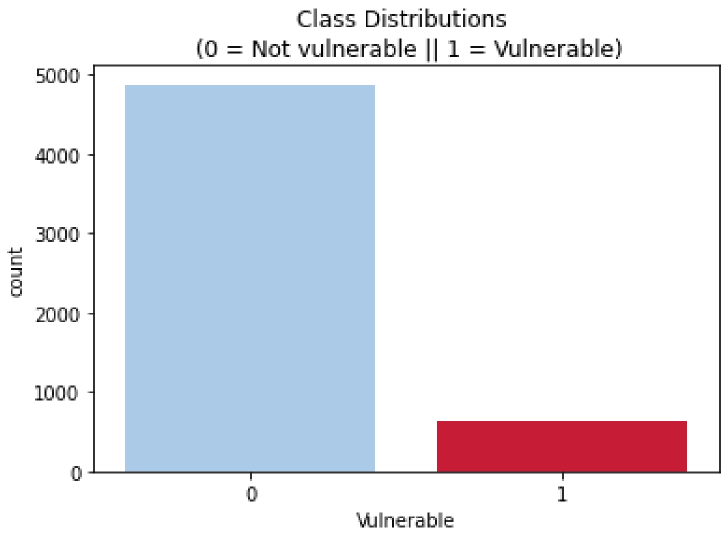


Figure 11: Class Distributions Before Undersampling and Oversampling

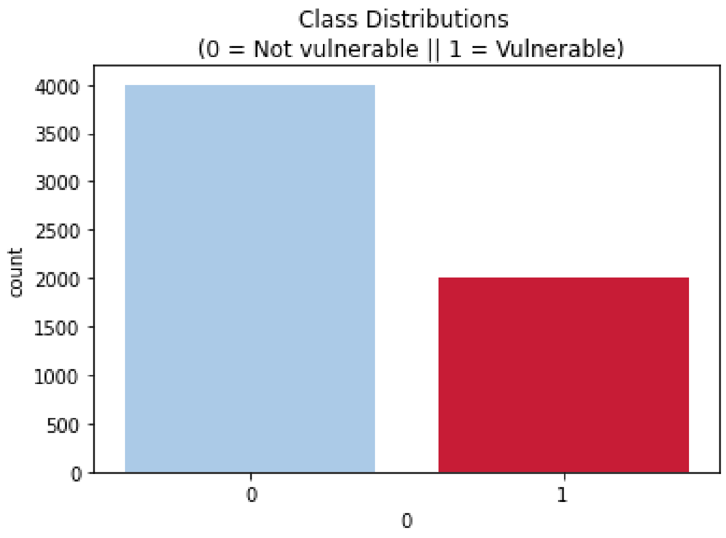
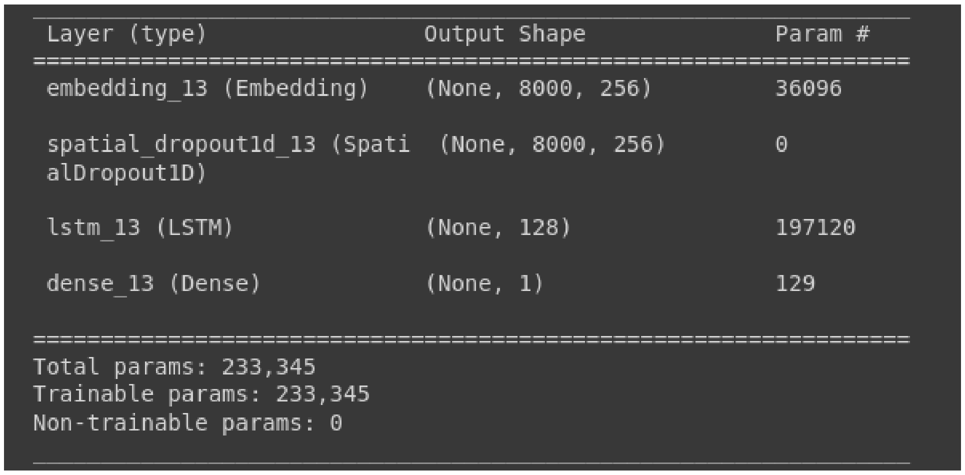


Figure 12: Class Distributions After Undersampling and Oversampling

7.3 Network

A standard LSTM network with word embeddings was used. The use of RNNs to classify sequential data is well known. Li et al. documented their use in program analysis. As of 2018, they had not been used for vulnerability detection purposes [30]. The vanishing gradient problem is addressed in LSTMs. I still experienced exploding gradients in my LSTM modeling, but only when my optimizers learning rate was too high. A general model summary of the LSTM used is included as Figure 13.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_13 (Embedding)    (None, 8000, 256)         36096

 spatial_dropout1d_13 (Spati  (None, 8000, 256)        0
 alDropout1D)

 lstm_13 (LSTM)              (None, 128)               197120

 dense_13 (Dense)            (None, 1)                 129

=================================================================
Total params: 233,345
Trainable params: 233,345
Non-trainable params: 0
_________________________________________________________________
```

Figure 13: Example Model Architecture

The first layer is an embedding layer containing 3 parameters. Embedding layers are often used in natural language processing approaches. The first argument passed indicates the size of the vocabulary. I used a vocabulary restricted to valid bytecodes (141 words). The second argument is the length of the vector for each word. Different values were and are still being experimented with, but 128 proved optimal in my experiments. The best maximum sequence length varies depending on the parameters being used. On some experiments 5000 seemed optimal and on others 8000 performed better. This is impacted by the less than uniform length of the smart contracts in the dataset.

Spatial dropout is used as a second layer in this model. The third layer is the LSTM layer with 128 units. 256 units, all other parameters held constant, did not perform as well. An LSTM was chosen due to the reported success of the reviewed related works. A bidirectional LSTM layer, as used by Li et al. and Qian et al., was also been experimented with. The standard LSTM outperformed on my data. The final layer is a dense layer with 1 unit, producing the probability of the predicted classification (1 or 0). Both the LSTM layer and dense layer utilize the sigmoid function for activation. This is standard for binary classification problems.

Networks were trained on both the bytecode of contracts with PUSH values removed and the contracts with PUSH values in-tact. The bytecode with PUSH values removed performed better than the original data. Different epochs and batch sizes were required for optimal performance in different configurations. A general standard that ”worked” here was 10 epochs and batch sizes of 64. When experimenting with different batch sizes, depending on the batch itself, the network sometimes learned very little. Larger batch sizes generally reduced performance. I believe this was compounded by the synthetic data. I also used statically decided random samples. Changing the seed numbers for these random samples had a large impact on the success of the training. Scaling batch sizes up was possible, but depending on the sequence lengths being used would occasionally result in errors even in Google Colab.

An example network implementation is included here:

```python
import keras
from tensorflow import keras
from keras.layers import Dense, Embedding, LSTM, SpatialDropout1D, Bidirectional
from keras.models import Sequential
from keras.callbacks import EarlyStopping
import tensorflow as tf
import os
import numpy as np
from shutil import copy

epochs = 10
emb_dim = 128
batch_size = 64 #128 #32
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)
loss = tf.keras.losses.BinaryCrossentropy(from_logits=False)

n_most_common_words = 141
model = Sequential()
model.add(Embedding(n_most_common_words, emb_dim, input_length=X_train_res.shape[1]))
model.add(SpatialDropout1D(0.3))
model.add(LSTM(128, dropout=0.3, recurrent_dropout=0, recurrent_activation='sigmoid'))
    #cuDNN reuires recurrent_dropout = 0
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer=optimizer, loss=loss, metrics=['acc'])
print(model.summary())

import time
start_time = time.time()

history = model.fit(X_train_res, y_train_res, epochs=epochs, batch_size=batch_size,
    validation_split=0.0, validation_data=(X_test_res,
    y_test_res),callbacks=[EarlyStopping(monitor='loss',patience=7, min_delta=0.0001)])

end_time = time.time()
print('Time taken for training: ', end_time-start_time)
```

7.4 Results

The goal of the network was to perform binary classification of reentrancy vulnerable and not reentrancy vulnerable contracts on the novel ScrawlD dataset. The model produced by Tann et al. aimed to perform multi-class classification on vulnerable and not vulnerable contracts.[13] Tann et al. reported 99.57% classification accuracy. The model produced here was not as successful. It performed well (90% accuracy, 85% precision accuracy) on synthetic test data, but the prediction accuracy on non-synthetic test data was only 85%. These results were worse (less than 75%) on additional data that was scraped according to labels provided by SmartBugs. There are a number of explanations as to why this could be the case: different problems requiring different feature sets, shorter contract sequences with more dense features[14], significantly (an order of magnitude) more data being used in Tann et al.'s case, and different parameters otherwise. The results described are of just one iteration of my models; I would not say that this is final as I am still experimenting. I believe that my current results can be improved.

The training of a "well" performing model rarely produced a smooth accuracy or loss curve. Figure 14 shows an example training and validation accuracy of a 10 epoch session. Figure 15 shows the respective training and validation loss.

Model evaluation was done on both the synthetic and non-synthetic test sets. Synthetic test sets scored higher in both accuracy and precision. Evaluation on non over sampled test data still achieved acceptable accuracy, but was generally worse. An attempt was made to produce more accurate testing

[13]As noted previously, vulnerable in this context means suicidal, prodigal or greedy.
[14]This further impacts the oversampling produced by SMOTE.

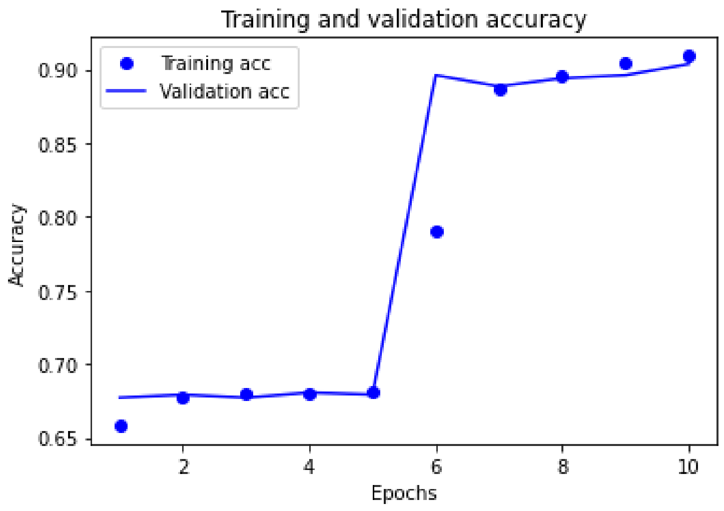


Figure 14: Training and Validation Accuracy

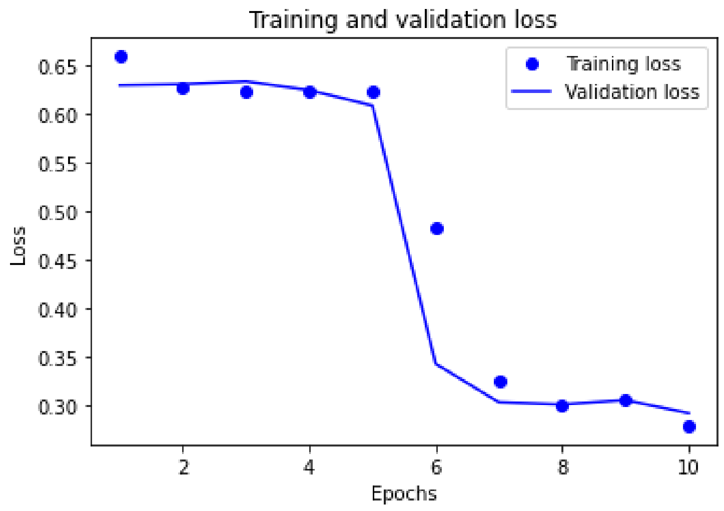


Figure 15: Training and Validation Loss

results. I scraped the contents of the curated SmartBugs dataset for reentrancy vulnerable contracts. This unfortunately only resulted in 20 additional contracts. A total of 4 of the contracts listed were already included in my ScrawlD reentrancy vulnerable dataset. Those contracts were removed, leaving 16. In most experiments I had undersampled the non-vulnerable ScrawlD contracts, so took an additional 20 from the un-used contracts for non-vulnerable data. The model that produced those graphs, produced a classification accuracy of less than 75% on this dataset. The prediction errors made were false negatives, which is indicative of the class imbalance. Unfortunately, increasing the SMOTE oversampling ratio beyond 60% decreased accuracy further. The SMOTE approach is known for generating noisy data. This held true in this project. Loss was rarely below 25% on training and validation data.

7.5 Additional Experimentation

In an attempt to work around the sequence lengths, I additionally experimented with non-sequence approaches including bag-of-words, n-grams, count vectorization and tf-idf on a convolutional neural network. Evidence that these approaches have applications in bytecode classification exists in unrelated research. The n-grams approach, using n-grams of size 6, achieved 65% classification accuracy on balanced test data.

An LSTM model configured as described by Tann et al. was trained on the data hosted in Tann et al.'s github repository. The data hosted is not the same data used in their publication. Only 70% classification accuracy was achieved on synthetic test data. This network did not classify reentrancy vulnerablities, but vulnerable and not vulnerable contracts as a whole. In this case, vulnerable contracts are contracts classified as prodigal, suicidal and greedy by MAIAN.

7.6 Future Work

Informed by my results, I am of the impression that the approach of "code gadgets" used by Li et al., Zhuang et al. and Qian et al. is more appropriate for classifying reentrancy vulnerabilities. In the section, Methodology 1, I identified a way to potentially extend the tooling Zhuang et al. produced to include a more accurate feature set. I think following through on this would have led to greater success. All of the reviewed works that classified reentrancy did so with the use of code gadgets or a related derivation. I don't think the current approach is necessarily useless, though. More time is needed to explore the dataset and problem. Some of my iterations were almost there, predicting a probability of .49 for reentrancy positive contracts and less than .1 probability for non-reentrancy contracts. If I could skew the predictions just slightly in the vulnerable direction, they would be alright. I have not been able to resolve this yet.

It could be that reentrancy as a vulnerability does not have bytecode level features that are easily recognized. All of the work reviewed, other than Ahsizawa et al., used high level languages when classifying reentrancy vulnerabilities. I have started to explore this, but am too late to improve my results at this time. As a first step to exploration I have created and compiled a bare-bones reentrancy vulnerable solidity contract to both bytecode and opcodes. Figure 16 shows the bytecode associated with the reentrancy vulnerable contract. Figure 17 shows the opcodes associated with the reentrancy vulnerable contract.

Re-Entrancy Vulnerable Smart Contract:

```
    contract Reentrance {
     mapping (address => uint) userBalance;

     function getBalance(address u) constant returns(uint){
         return userBalance[u];
     }

     function addToBalance() payable{
         userBalance[msg.sender] += msg.value;
     }

     function withdrawBalance(){
         if( ! (msg.sender.call.value(userBalance[msg.sender])() ) ){
             throw;
         }
         userBalance[msg.sender] = 0;
     }
 }
```

I need to explore the solidity compiler (version specific) in greater detail as the compiler makes optimizations, but identifying which functions relate to which opcodes and bytecodes explicitly would be a step in the right direction.

Other gains could be made in exploring the initilization weights of layers, which were left at default. Further undersampling could be explored. More specifically, it would be beneficial to sample contracts

```
6080604052348015610010576000 80fd5b506102518061002060003960 00f3006080604052600436106100575760003 57c010
    0000000000000000000000000000000000000000000000000000000900463ffffffff1680635fd8c7101461005c578063c0e3
    17fb14610073578063f8b2cb4f1461007d575b600080fd5b348015610068576000 80fd5b506100716100d4565b005b61007b6
    1018f565b005b348015610089576000 80fd5b506100be600480360381019080803573ffffffffffffffffffffffffffffffff
    ffffffff1690602001909291905050506101dd565b6040518082815260200191505060405180910390f35b3373fffffffffff
    fffffffffffffffffffffffffffff166000803373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffff
    ffffffffffffffffffffffff1681526020019081526020016000205460405160006040518083038185875af19250505015156
    1014957600080fd5b60008060003373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffff
    ffffffffffffff1681526020019081526020016000208190555056 5b346000803373fffffffffffffffffffffffffffffffff
    fffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002060008282540192505081
    905550565b60008060008373ffffffffffffffffffffffffffffffffffffffff1673fffffffffffffffffffffffffffffffff
    fffffff1681526020019081526020016000205490509190505600a165627a7a7230582005689e15436cdf2de1744177293939
a63c54412451ef46c2ee5bbdad52430c720029
```

Figure 16: Bytecode Associated with Re-Entrancy Vulnerable Contract

```
PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH2 0x10 JUMPI PUSH1 0x0 DUP1 REVERT
    JUMPDEST POP PUSH2 0x251 DUP1 PUSH2 0x20 PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP
    PUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH2 0x57 JUMPI PUSH1 0x0
    CALLDATALOAD PUSH29 0x100000000000000000000000000000000000000000000000000000000 SWAP1
    DIV PUSH4 0xFFFFFFFF AND DUP1 PUSH4 0x5FD8C710 EQ PUSH2 0x5C JUMPI DUP1 PUSH4
    0xC0E317FB EQ PUSH2 0x73 JUMPI DUP1 PUSH4 0xF8B2CB4F EQ PUSH2 0x7D JUMPI JUMPDEST
    PUSH1 0x0 DUP1 REVERT JUMPDEST CALLVALUE DUP1 ISZERO PUSH2 0x68 JUMPI PUSH1 0x0 DUP1
    REVERT JUMPDEST POP PUSH2 0x71 PUSH2 0xD4 JUMP JUMPDEST STOP JUMPDEST PUSH2 0x7B
    PUSH2 0x18F JUMP JUMPDEST STOP JUMPDEST CALLVALUE DUP1 ISZERO PUSH2 0x89 JUMPI PUSH1
    0x0 DUP1 REVERT JUMPDEST POP PUSH2 0xBE PUSH1 0x4 DUP1 CALLDATASIZE SUB DUP2 ADD
    SWAP1 DUP1 DUP1 CALLDATALOAD PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND
    SWAP1 PUSH1 0x20 ADD SWAP1 SWAP3 SWAP2 SWAP1 POP POP POP PUSH2 0x1DD JUMP JUMPDEST
    PUSH1 0x40 MLOAD DUP1 DUP3 DUP2 MSTORE PUSH1 0x20 ADD SWAP2 POP POP PUSH1 0x40 MLOAD
    DUP1 SWAP2 SUB SWAP1 RETURN JUMPDEST CALLER PUSH20
    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND PUSH1 0x0 DUP1 CALLER PUSH20
    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND PUSH20
    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND DUP2 MSTORE PUSH1 0x20 ADD SWAP1 DUP2
    MSTORE PUSH1 0x20 ADD PUSH1 0x0 KECCAK256 SLOAD PUSH1 0x40 MLOAD PUSH1 0x0 PUSH1 0x40
    MLOAD DUP1 DUP4 SUB DUP2 DUP6 DUP8 GAS CALL SWAP3 POP POP POP ISZERO ISZERO PUSH2
    0x149 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST PUSH1 0x0 DUP1 PUSH1 0x0 CALLER PUSH20
    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND PUSH20
    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND DUP2 MSTORE PUSH1 0x20 ADD SWAP1 DUP2
    MSTORE PUSH1 0x20 ADD PUSH1 0x0 KECCAK256 DUP2 SWAP1 SSTORE POP JUMP JUMPDEST
    CALLVALUE PUSH1 0x0 DUP1 CALLER PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND
    PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND DUP2 MSTORE PUSH1 0x20 ADD
    SWAP1 DUP2 MSTORE PUSH1 0x20 ADD PUSH1 0x0 KECCAK256 PUSH1 0x0 DUP3 DUP3 SLOAD ADD
    SWAP3 POP POP DUP2 SWAP1 SSTORE POP JUMP JUMPDEST PUSH1 0x0 DUP1 PUSH1 0x0 DUP4
    PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND PUSH20
    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND DUP2 MSTORE PUSH1 0x20 ADD SWAP1 DUP2
    MSTORE PUSH1 0x20 ADD PUSH1 0x0 KECCAK256 SLOAD SWAP1 POP SWAP2 SWAP1 POP JUMP STOP
    LOG1 PUSH6 0x627A7A723058 KECCAK256 SDIV PUSH9 0x9E15436CDF2DE17441 PUSH24
    0x293939A63C54412451EF46C2EE5BBDAD52430C7200290000
```

Figure 17: Opcodes Associated with Re-Entrancy Vulnerable Contract

with certain properties (like those that fall below the mean length of the dataset). A deeper understanding of the dataset as a whole could allow for the choice of contracts with more prominent desired features. Combining the separate datasets located into a meta-dataset and removing duplicates could improve my results as well. I think it would be worthwhile to explore both more simple and more complex models. In this project, I jumped straight to the LSTM approach.

A minor future improvement would also include re-writing the script used to remove values following PUSH operations on bytecode in a way that isn't quadratic time complexity.

8 Conclusion

The goals outlined in my project proposal were, in a condensed format, as listed:

- Familiarizing myself with vulnerabilities in general and/or vulnerabilities specific to smart contracts.
- Reviewing currently automated vulnerability analysis tools with respect to smart contracts. My goal was to find a weakness in these tools, if it existed.
- Reviewing the approaches and performance of deep learning for vulnerability detection in smart contracts.
- Finding or generating and processing a smart contract dataset.
- Implementing my own deep learning model and documenting its performance relative to the the work guiding me.

While I failed to produce an effective deep learning model for classifying smart contract vulnerabilities, I did accomplish the rest of my goals. I am now familiar with smart contract vulnerabilities, automated vulnerability detection tools related to smart contract vulnerabilities, the approaches taken to produce said tools, and believe I located most, if not all, datasets that have been made publicly available in the space. I am confident that given the knowledge I have gained thus far, and more time, I could produce a model of more value than what I did here. I additionally gained a much deeper understanding of blockchain technology, smart contract technology and other related concepts such as compilers and stack machines. I gained experience with the tools commonly used in the environment, non-exhaustively including Solidity, Remix, MAIAN, Node.js and Web3.js.

Part of the problem with my approach to this project was choosing an appropriate vulnerability. I was constrained by available data and had little technical background in the domain at the onset. I did not want to re-create the same work that had already been done so I took a leap and tried to do something moderately novel. I believe that if I was to start again, knowing what I know now, I would be better positioned to succeed in doing so.

Throughout my degree I have consistently "bit off more than I can chew." Projects that interest me seem to be projects that I am less than prepared for. This has been an advantage with respect to knowledge gained and a disadvantage with respect to the work I have been able to produce. When I begin to master a problem, it is often time to move on. In the case of this project I underestimated the amount of domain specific background that would be required for success. Much of my time was spent closing that gap. I would have liked to produce something of use to someone given the time that went into this. Now that I have the understanding I do, I intend to do so.

References

[1] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. Blockchain challenges and opportunities: A survey - henrylab.net. Available at https://www.henrylab.net/wp-content/uploads/2017/10/blockchain.pdf.

[2] Takenobu T. Ethereum evm illustrated. Available at https://takenobu-hs.github.io/downloads/.

[3] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. Available at https://nvlpubs.nist.gov/nistpubs/ir/2018/nist.ir.8202.pdf.

[4] Chavhan Yashavant, Saurabh Kumar, and Amey Karkare. Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities. Available at https://arxiv.org/pdf/2202.11409.pdf.

[5] Heidelinde Rameder. *Systematic Review of Ethereum Smart Contract Security Vulnerabilities, Analysis Methods and Tools.* PhD thesis, 2021. Available at https://web.archive.org/web/20210909005434id_/https://repositum.tuwien.at/bitstream/20.500.12708/18323/1/Rameder%20Heidelinde%20-%202021%20-%20Systematic%20Review%20of%20Ethereum%20Smart%20Contract...pdf.

[6] Jiachi Chen, Xin Xia, David Lo, John Grundy, Daniel Xiapu Luo, and Ting Chen. Defining smart contract defects on ethereum, Apr 2020. Available at https://arxiv.org/abs/1905.01467.

[7] Nami Ashizawa, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. Eth2vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts. Available at https://arxiv.org/pdf/2101.02377.pdf.

[8] Reza Parizi, Kim-Kwang Raymond Choo, Ali Dehghantanha, and Amritraj Singh. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. Available at https://arxiv.org/pdf/1809.02702.pdf.

[9] Thomas Duireux, Rui Abreu, João F. Ferreira, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart ... Available at https://arxiv.org/pdf/1910.10601.pdf.

[10] Inc. O'Reilly Media. Mastering bitcoin. Available at https://www.oreilly.com/library/view/mastering-bitcoin/9781491902639/ch07.html.

[11] Ethereum: a secure decentralised generalised transaction. Available at https://ethereum.github.io/yellowpaper/paper.pdf.

[12] Satoshi Nakomoto. Bitcoin: A peer-to-peer electronic cash system. Available at https://bitcoin.org/bitcoin.pdf.

[13] Nick Szabo. Smart contracts. Available at https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html.

[14] Alex Lipton and Stuart Levi. An introduction to smart contracts and their potential and inherent limitations, May 2018. Available at https://corpgov.law.harvard.edu/2018/05/26/an-introduction-to-smart-contracts-and-their-potential-and-inherent-limitations/.

[15] Gary Gensler and Lawrence Lessig. MIT OpenCourseWare. Available at https://ocw.mit.edu/courses/15-s12-blockchain-and-money-fall-2018/resources/session-6-smart-contracts-and-dapps/.

[16] Nick Szabo. Smart contracts 2. Available at https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[17] Wolfram Groschopf, Mario Dobrovnik, and Christian Herneth. Smart contracts for sustainable supply chain management: Conceptual frameworks for supply chain maturity evaluation and smart contract sustainability assessment, Jan 1AD. Available at https://www.frontiersin.org/articles/10.3389/fbloc.2021.506436/full.

[18] Golocorbin Kon S;Mikov M;. Counterfeit drugs as a global threat to health]. Available at https://pubmed.ncbi.nlm.nih.gov/21789919/.

[19] Ethereum virtual machine (evm) developer documentation. Available at https://ethereum.org/en/developers/docs/evm/.

[20] Gavin Wood. Ethereum: A secure decentralised generalised transaction. Available at https://ethereum.github.io/yellowpaper/paper.pdf.

[21] Ethereum virtual machine opcodes. Available at https://www.ethervm.io/.

[22] Luit Hollander. The ethereum virtual machine-how does it work?, Feb 2019. Available at https://medium.com/mycrypto/the-ethereum-virtual-machine-how-does-it-work-9abac2b7c9e.

[23] Junzhou Xu, Fan Dang, Xuan Ding, and Min Zhou. A survey on vulnerability detection tools of smart contract bytecode, Sep 2020. Available at https://dang.fan/publication/iciscae20-smart-contract/.

[24] Shelly Grossman, Yoni Zohar, Mooly Sagiv, Noam Rinetzky, Yan Michalevsky, Guy Golan-Gueta, and Ittai Abraham. 48 online detection of effectively callback free objects with applications to smart contracts. Available at https://www.cs.tau.ac.il/~shellygr/pubs/2018-popl.pdf.

[25] Available at https://docs.soliditylang.org/en/v0.8.12/contracts.html#fallback-function.

[26] ConsenSys. Reentrancy. Available at https://consensys.github.io/smart-contract-best-practices/attacks/reentrancy/.

[27] Timestamp dependency. Available at https://dasp.co/#item-8.

[28] Swc-114 · overview. Available at https://swcregistry.io/docs/SWC-114.

[29] Security considerations. Available at https://docs.soliditylang.org/en/develop/security-considerations.html#tx-origin.

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. Available at https://arxiv.org/pdf/1801.01681.pdf.

[31] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities, Jan 2021. Available at https://arxiv.org/abs/1807.06756, journal=arXiv.org.

[32] Zheng Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr/sysevr. Available at https://github.com/SySeVR/SySeVR.

[33] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. Smart contract vulnerability detection using graph neural network. Available at https://www.ijcai.org/Proceedings/2020/0454.pdf.

[34] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. Combining graph neural networks with expert knowledge for smart contract vulnerability detection, Jul 2021. Available at https://arxiv.org/abs/2107.11598.

[35] Zhenguang Liu, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. Available at https://arxiv.org/pdf/2106.09282.pdf.

[36] Peng Qian, Zhenguang Liu, Qinming He, Roger Zimmerman, and Xun Wang. Towards automated reentrancy detection for smart contracts based on sequential models. Available at https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&amp;arnumber=8970384.

[37] Wesley Joon-Wie Tann, Xing Jie Han, Sourav Sen Gupta, and Yew-Soon Ong. Towards safer smart contracts: A sequence learning approach to detecting security threats. Available at https://arxiv.org/pdf/1811.06632.pdf.

[38] Nikolic Ivica, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. Available at https://arxiv.org/pdf/1802.06038.pdf.

[39] Ajay K Gogineni1, S Swayamjyoti1, Devadatta Sahoo1, Kisor K Sahu2, Raj Kishore2, and Kisor K Sahu. Multi-class classification of vulnerabilities in smart contracts using awd-lstm, with pre-trained encoder inspired from natural language processing, Nov 2020. Available at https://iopscience.iop.org/article/10.1088/2633-1357/abcd29.

[40] Pouyan Momeni, Yu Wang, and Reza Samavi. Machine learning model for smart contracts security analysis. Available at https://ieeexplore.ieee.org/document/8949045.

[41] Hongjun Wu, Zhou Zhang, Shangwen Wang, Yan Lei, Bo Lin, Yihao Qin, Haoyu Zhang, and Xiaoguang Mao. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. Available at https://ieeexplore.ieee.org/document/9700296.

[42] Xingxin Yu, Haoyue Zhao, Zonghao Ying, and Bin Wu. Deescvhunter: A deep learning-based framework for smart contract vulnerability detection. Available at https://ieeexplore.ieee.org/abstract/document/9534324.

[43] Motjaba Eshgie, Cyrille Artho, and Dilian Gurov. Dynamic vulnerability detection on smart contracts using machine learning. Available at https://arxiv.org/pdf/2102.07420.pdf.

[44] João F. Ferreira. A dataset of vulnerable solidity smart contracts. Available at https://smartbugs.github.io/.

[45] Rob Behnke. Explained: The paraluni hack (march 2022), Mar 2022. Available at https://halborn.com/explained-the-paraluni-hack-march-2022/.

[46] Shaurya Malwa. Ola finance says attackers stole $4.7m in 're-entrancy' exploit, Apr 2022. Available at https://www.coindesk.com/tech/2022/04/01/ola-finance-says-attackers-stole-47m-in-re-entrancy-exploit/.

[47] Slowmist SlowMist. Another day, another reentrancy attack, Apr 2022. Available at https://slowmist.medium.com/another-day-another-reentrancy-attack-5cde10bbb2b4.

[48] Noama Fatima Samreen and Manar H. Alalfi. Reentrancy vulnerability identification in ethereum smart contracts, May 2021. Available at https://arxiv.org/abs/2105.02881.

[49] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Phillip Kegelmeyer. Smote: Synthetic minority over-sampling technique. Available at https://arxiv.org/pdf/1106.1813.pdf.

[50] Nick Mudge. Ethereum's maximum contract size limit is solved with the diamond standard, Oct 2021. Available at https://dev.to/mudgen/ethereum-s-maximum-contract-size-limit-is-solved-with-the-diamond-standard-2189.

[51] Ethereum. Removing contract size limit · issue 1662 · ethereum/eips. Available at https://github.com/ethereum/EIPs/issues/1662.