

Improving Reproducibility in Smart Contract Research

Sina Pilehchiha

A Thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Master of Applied Science (MAsc) at
Concordia University
Montréal, Québec, Canada

July 2022

© Sina Pilehchiha, 2022

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Sina Pilehchiha**

Entitled: **Improving Reproducibility in Smart Contract Research**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (MAsc)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Yousef R. Shayan Chair

Dr. Examiner

Dr. Examiner

Dr. Amir G. Aghdam Thesis Supervisor

Dr. Jeremy Clark Thesis Supervisor

Approved by _____
Dr. M. Zahangir Kabir, Graduate Program Director

July 31, 2022

Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Improving Reproducibility in Smart Contract Research

Sina Pilehchiha

On the Ethereum blockchain, smart contracts are used to secure billions of dollars' worth of assets. The most popular smart contract-based blockchain platform at the moment is Ethereum. Based on market value, it is the second-largest blockchain network behind Bitcoin, with a steadily increasing market share. Smart contracts' code must be examined for any potential flaws that could result in significant financial losses and harm confidence because they cannot be modified after they have been deployed. For this goal, a wide range of tools have been created, and new literature on vulnerabilities and detection techniques on the aforementioned subject is constantly being produced. The analysis, testing, and debugging of smart contracts have also been the subject of extensive research into automation. Unfortunately, it is difficult and time-consuming to replicate the findings of the majority of prior empirical studies or to contrast one's findings with those of others. When research articles offer datasets, they frequently come in the form of a sparse GitHub repository with minimal to no usage guidance. The datasets frequently get out of date very quickly on the Ethereum platform because of its rapid development. Since it takes a lot of time to complete numerous tasks including finding, extracting, cleaning, and categorising a sizable volume of high-quality, heterogeneous smart contract data, these are significant impediments to doing verifiable, reproducible research. We introduce etherbase, an extendable, queryable, and user-friendly database of smart contracts and their metrics that improves repeatability and benchmarking in smart contract research, in order to address this issue.

Acknowledgments

I would like to thank my supervisors, Profs. Jeremy Clark and Amir G. Aghdam, for their supervision, continued support, and dedication that made this thesis possible. I would not be where I am today without their help and generosity. I would like to thank Dan Guido, Josselin Feist, and Gustavo Grieco for their support during my internship at Trail of Bits, Inc. They helped me grow and thrive.

Contents

List of Figures	vii
List of Tables	viii
1 Introduction	1
1.1 Motivation	1
1.2 Thesis Statement	2
1.3 Outline and Contributions	3
2 Background	4
2.1 Ethereum	4
2.1.1 Accounts	6
2.1.2 Transactions	7
2.1.3 Blocks	7
2.1.4 Ethereum Virtual Machine	7
2.2 Smart Contracts	8
2.2.1 Vulnerabilities	8
3 Automated Vulnerability Analysis of Smart Contracts on Ethereum	12
3.1 Introductory Remarks	12
3.2 Traditional Security Analysis Methods in Smart Contracts	13

3.3	Deep Learning in Smart Contracts	15
3.4	SLITHER-SIMIL	25
3.5	Concluding Remarks	32
4	ETHERBASE: Improving Reproducibility in Smart Contract Research	33
4.1	Introductory Remarks	33
4.2	Related Work	35
4.3	Methodology	37
4.3.1	Data Acquisition	38
4.3.2	Data Generation	39
4.3.3	Data Storage	43
4.3.4	Data Application	45
4.4	Concluding Remarks	48
5	Conclusion and Future Work	50
	Bibliography	51

List of Figures

1	Ethereum blockchain structure.	5
2	An illustration of Ethereum’s GHOST protocol.	6
3	A high-level view of the process workflow of Slither-simil.	27
4	EtherBase Workflow	38
5	No. of Contracts containing vulnerabilities (log-scale)	47

List of Tables

1	Primary Metrics on Smart Contracts	41
2	Supported Vulnerabilities	46

Chapter 1

Introduction

This Chapter introduces several research questions to be answered in this thesis and motivates their importance.

1.1 Motivation

The research community has made much effort to develop automated analysis tools [25] to be able to identify vulnerabilities in smart contracts. [25] These tools and frameworks analyze smart contracts and produce vulnerability reports. In a study in 2020, researchers analyzed about one million Ethereum smart contracts and found 34,200 potentially vulnerable. [33] Another research effort showed that 8,833 (around %46) smart contracts on the Ethereum blockchain were flagged as vulnerable out of 19,366 smart contracts. [29]

Famous attacks that have caused significant financial losses and prompted the research community to work to prevent similar occurrences include The DAO hack [9] and the Parity wallet issue [35]. Comparing and reproducing such research is not an effortless process. The datasets used to test and benchmark those tools proposed in the research literature are not publicly available, making reproduction efforts immensely hard to carry out.

If a researcher intends to benchmark their new proposed methodologies, models, or

toolsets or experiment with different research ideas and compare their work with the existing work and publications, the best they can do is to directly contact the authors of the previous publications and researchers to have potentially in-time access to the same datasets used in the original research/work or make do with whatever out-of-date incomprehensive and unrepresentative dataset they have at their disposal, a very timely and inefficient process. [29]

In other cases, the researchers must start from scratch and create their datasets, a non-trivial and slow process. What makes it worse is the data bias, which can be introduced in a dataset in different phases of data acquisition and cleaning. The presence of such bias in data can quickly escalate to threaten the research’s validity. [40]

1.2 Thesis Statement

In this thesis, we present ETHERBASE, an open-source, extensible, queryable, and easy-to-use database that facilitates and enhances the verification and reproducibility of previous empirical research and lays the groundwork for faster, more rapid production of research on smart contracts. The source code for data acquisition and cleaning is not available due to private IP reasons, and only the collected data and the database is accessible to the public. Researchers, smart contract developers, and blockchain-centric teams and enterprises can use such corpus for specific use-cases.

To summarize, our contributions are as follows:

1. We propose ETHERBASE, a systemic and up-to-date database for Ethereum by exploiting its internal mechanisms.
2. We implement ETHERBASE and make its pipelines open-source. It obtains historical data and facilitates benchmarking and reproduction in research and development for new toolsets. It is more up-to-date than existing datasets and gets automatically

reviewed and renewed compared to the previous manual one-time data gathering efforts.

3. We propose the first dataset of Ethereum smart contracts, which has a mix of off-chain and on-chain data together, meaning that it contains the source code and the bytecode of the corresponding smart contracts in the same dataset.

1.3 Outline and Contributions

The rest of this dissertation is organized as follows: In Chapter 2, we go over the background material needed to understand the basic technicalities of blockchain technology and the current state of the art in developing security-enhancing tools for smart contracts.

In Chapter 3, we summarize and evaluate state of the art regarding automated vulnerability analysis practices for smart contracts on Ethereum. We discuss the motivations behind developing such tools, what they have achieved so far, and our efforts in developing and working on the tool SLITHER-SIMILAn effort in such direction.

In Chapter 4, we take a broad look at the efforts taken at improving the reproducibility in smart contract research, how we have tried to improve it by introducing ETHERBASE, and its use in getting better insights at the capabilities of some of the most frequently smart contract testing tools in research and industry.

In the final Chapter, we remark on our conclusions and reveal plans for future work and research directions.

Chapter 2

Background

This chapter reviews the necessary background knowledge for the reader to be better acquainted with the work conducted within this thesis. It highlights the foundational technical basics of Ethereum blockchain, smart contracts and the most common vulnerabilities associated with smart contracts. We cover these technicalities in the following order (based on the work of [18]): First, we go over the basics of Ethereum, its components and structure. We will go through how blocks are formed, what sort of accounts exist on the Ethereum network, and how transactions are executed. We will also go through how the Ethereum Virtual Machine functions. Afterwards, we will go over smart contracts and their most common vulnerabilities out in the wild. We will discuss Solidity-written source code and bytecode of smart contracts, and explain each vulnerability according to the DASP 10 classification. [21]

2.1 Ethereum

Ethereum is a decentralized virtual machine that was introduced as an alternative blockchain technology to Bitcoin in 2014 by [49]. A blockchain is essentially a peer-to-peer network made up of computers that act as nodes and distribute updates for a single database without



Figure 1: Ethereum blockchain structure.

necessarily having confidence in one another. It is based on a combination of combination of cryptography, networking, and incentive mechanisms. [48] The aforementioned database effectively serves as a ledger, recording each and every transaction that each node in the blockchain network makes.

As the second most popular blockchain, the Ethereum blockchain was developed as an alternative to cover for the lackings of Bitcoin. It is a transaction-based, cryptographically secure state machine, that reads a series of inputs and, based on those inputs, transitions to a new state. [18] Just like Bitcoin, Ethereum currently uses Proof-of-Work (PoW) as its consensus protocol. Proof-of-Stake (PoS), PBFT(Practical Byzantine Fault Tolerance), and DPoS (Delegated Proof of Stake) are other forms of consensus protocol, for example. The solution to a series of cryptographic puzzles are used in the PoW mechanism to prove the credibility of the data being written on the blockchain, using such mechanism as their consensus protocol. The puzzle is usually a computationally hard but easily verifiable mathematical problem. When a node creates a block, it must resolve a PoW puzzle and spend computing power to achieve so. The nodes compete o=with each other over this objective function and the node with the most computing power usually succeeds ind doing so. After the PoW puzzle is resolved, it will be broadcasted to other nodes, so as to achieve the purpose of consensus and append a new block to the blockchain. It acts as a “proof” that a node has done “work” by spending its computational resources. This process is known as

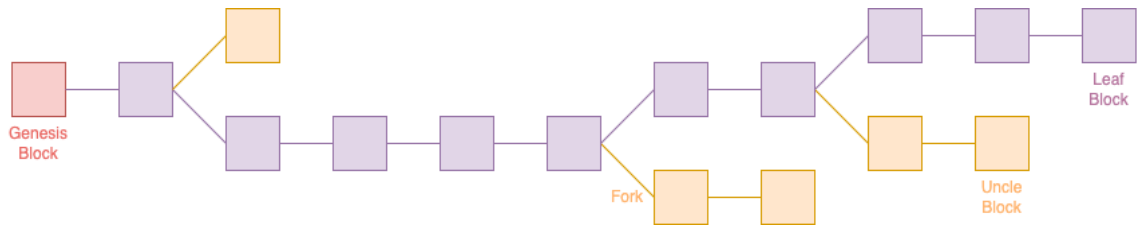


Figure 2: An illustration of Ethereum's GHOST protocol.

mining and nodes which decide to participate in this process and try to create new blocks are known as miners.

2.1.1 Accounts

The Ethereum state is consisted of many small objects naemd as accounts, where each account has a 20-byte address and state transitions to be able to interact with the other accounts on-chain. An address on the Ethereum blockchain is a 160-bit identifier that is used to identify any account. The world state is a mapping between addresses and account states. [49] Ethereum suppotrs two types of accounts: externally owned (controlled by private keys) -namely EOA's- and contract accounts (CA'S, controlled by their contract code) [13]. Inside of an Ethereum account is composed of four fields: nonce, ether balance, contract code hash, and storage root, explained as follows:

- **Nonce:** Nonce represents the number of transactions sent from particular address or the number of contract creations made by an account and is used as a guarantee that each transaction can only be processed once.
- **Balance:** Ether balance is the number of Wei owned by this address Wei is the smallest subunit of ether (1 wei is equivalent to 10^{-18} ether).
- **Storage Root:** Storage root is the 256-bit hash of the root node of a Merkle Patricia tree that represents the content of the account

- **Contract CodeHash:** Contract code hash is the Keccak-256 hash of Ethereum Virtual Machine (EVM) code of the account, which is executed if an address receives a message call.

2.1.2 Transactions

A transaction is a cryptographically signed instruction sent by an account on the network towards another. There exist only two types of transactions based on the outcomes they generate:

- Message calls, which are created by contract accounts to produce and execute a message that leads to the recipient account (an EOA or contract account) running its code. The simplest of such transaction is sending Ether from an account to another.
- Contract creation call, which results in the creation of new accounts with a code associated with it..

2.1.3 Blocks

A block is basically a collection of transactions that are executed in sequence.

2.1.4 Ethereum Virtual Machine

The formal definition of the EVM is specified in the Ethereum Yellow Paper. [49] The Ethereum Virtual Machine (EVM) at the heart of the Ethereum blockchain is a VM (virtual machine), with a stack-based architecture with 256-bit word sizes, supporting Turing-complete programming languages. EVM handles the computation side for Ethereum and comes with a set of instructions (namely, opcodes). Thus, a smart contract from a low-level point of view is a series of opcode instructions which EVM can read and compute

and execute the logic of that smart contract. The EVM is also responsible for handling the estimation and calculation of gas consumption for transactions in smart contracts.

2.2 Smart Contracts

The concept of smart contracts - programs running on the EVM - has been first introduced by Nick Szabo in one of his works in 1997. [43] They provide a framework that allow any sound program to be executed in an autonomous, distributed, and trusted manner. [32] The main programming language currently in use for the development of smart contracts is Solidity, although Vyper is gaining gradual traction as well.

2.2.1 Vulnerabilities

Solidity, like any other programming language in history, is prone to all kinds of vulnerabilities. What makes security vulnerabilities in Solidity so attractive is the fact that the programs written in Solidity are very much often used in the financial sector, handling millions of dollars in digital assets and cryptocurrencies. Attacking such contracts successfully can result in enormous financial losses. Some of these vulnerabilities like another programming language arise from the human factor involved in development of the smart contracts, and some specific to the blockchain data structures and how they and their components function and interact with each other. And these are only vulnerabilities at the scope of smart contracts we are focusing on. Vulnerabilities can arise with regards to core infrastructure of the blockchains handling smart contracts as well. In this section, we go over 9 of the more discussed vulnerabilities in Solidity and Ethereum according to [21] to get a better sense of what threat surface the developers and researchers developing analysis tools face:

Re-entrancy Often called as the most famous Ethereum vulnerability, the re-entrancy attack has been a great example at showing the risks of "*Code is Law*" and the importance of smart contract security, historically. The DAO hack [10] is one of the most famous real world examples of the re-entrancy hack. The re-entrancy attack can also be counted as a type of denial-of-service (DoS) attack, where a malicious actor can cause a program to infinitely loop and consume CPU cycles and in the case of smart contracts, drain a wallet of its ETH's. Reentrancy occurs when external contract calls are allowed to make new calls to the calling contract before the initial execution of that call is complete. For a function, this means that the contract state may change in the middle of its execution as a result of a call to an untrusted contract. [21]

Access Control The Access Control vulnerability, not exclusive to smart contract types of programs, usually occur when smart contracts use use poor visibility settings with regards to calling functions. This gives the attackers the ability to try to access the smart contract's private values, or hijack the control of the smart contract (for example, becoming the owner of a contract by initializing that contract through a statement like `owner = msg.sender()`).

Arithmetic Integer overflows and underflows can cause huge losses in smart contract-based applications [1]. Values assigned with the integer data type, if not handled carefully with regard to being signed or unsigned integers, can cause overflows and underflows and cause DoS-type attacks.

Unhandled Exception Also known as unchecked-send, this vulnerability can cause unwanted outcomes when the smart contract is executed, due to the fact that some low level calls in Solidity like `call()` and `delegatecall()` can return a boolean value set to the value `False` and let the execution flow resume if an error happens mid-execution. This

is not ideal since it means that the execution of the smart contract has not been reversed and has successfully been completed, but with wrong or undesirable outcomes. Thus, the return values of such low-level calls should always be checked and the developers need to make sure that such exceptions are handled appropriately during execution.

Frontrunning The frontrunning vulnerability is one of the more famous ones in the list, also known as Transaction Ordering Dependence (TOD). Exploitation of this vulnerability happens when malicious miners decide to alter the initial default ordering of the transactions submitted to the blockchain. Per Eskandari et al. [?], frontrunning can be generally reduced into three templates:

- Displacement attack, where an adversarial party makes a transaction in order to displace the victim user's transaction by having a higher gas price and thus, the attacker's transaction gets mined before that of victim's due to it giving having more aligned incentives with the miners' network.
- Insertion attack, in which an adversary makes two transactions, one with a higher gas price than the victim transaction and one with a lower gas price, to *sandwich* the victim transaction. [46]
- Suppression attack, where an attacker makes multiple transactions with higher gas prices than the victim transactions to prevent them from being mined in the same block.

Bad Randomness Also known as *nothing is secret*, [21] this vulnerability happens when smart contracts attempt to generate random, or to be more exact, pseudo-random numbers for any number of reasons. If the smart contract generating the pseudo-random number computes that random number using values that can be guessed by a malicious party, then the attacker can predict the next number that will be generated. Values such as block

timestamps, or block number are generally advised against to be used in such mechanisms. They are called hard-to-predict values but it is better to use an external oracle to generate the random numbers needed [7].

Time Manipulation This vulnerability is also known as *timestamp dependence* [21]. In Solidity, a block's timestamp is often used to generate pseudo-random numbers and in other times, it can be leveraged for smart contracts to conduct time-intensive operations, like unlocking funds at a specific time. A malicious miner of a block can manipulate the timestamp reported while generating the block and use this vulnerability to their own profit.

Short Address The short address vulnerability, also known as off-chain issues, results from the Ethereum Virtual Machine accepting arguments with incorrect paddings. Attackers trying to exploit this vulnerability, can craft truncated addresses which clients may encode incorrectly in transactions. Additionally, it has not been exploited in the wild as mentioned by [17].

Chapter 3

Automated Vulnerability Analysis of Smart Contracts on Ethereum

3.1 Introductory Remarks

SWith the rapid growth of blockchains, market contracts—universal and essential software applications—have attracted growing attention. On the Ethereum Mainnet, for instance, more than ten million smart contracts have been installed.

An event-driven, self-executing, state-based software known as a "smart contract" is created using high level programming languages like Vyper and Solidity. In order to facilitate quick and reliable transactions, smart contracts have been widely implemented in various business fields.

Because of their distinct features, smart contracts require more work to build than standard programmes do. First, compared to regular programmes, smart contracts are more resistant to bugs. A smart contract that has been published cannot be changed because "code is law." This is due to the fact that smart contract transactions always include cryptocurrencies, each of which is worth millions of dollars (e.g. The DAO). A smart contract bug could cause a substantial loss. Therefore, it is essential to check contracts for accuracy

before releasing. This necessitates the reuse of earlier contract development experience when creating new contracts. The creation and upkeep of smart contracts can be made much easier by using programme mining techniques for smart contracts like summarization, checking, and code search. The conventional statistical analysis tools for detecting weaknesses in smart contracts purely rely on manually defined patterns, which are likely to be error-prone and can cause them to fail in complex situations. Expert attackers can therefore quickly exploit these manual inspection patterns. To minimize the risk of the attackers, machine learning powered systems provide more secure solutions relative to hard-coded static checking tools.

Surucu et al. [42] provide the first-ever survey on machine learning methods utilized for the purposes of discovery and mitigation of vulnerabilities in smart contracts. In order to set the ground for further development of ML method on smart contract vulnerability detection, They reviewed many ML-driven intelligent detection mechanism on the following databases: Google Scholar, Engineering Village, Springer, Web of Science, Academic Search Premier, and Scholars Portal Journal. Based on their survey paper, we briefly go over the existing analysis tool first, and the the novel deep-learning-based methodologies proposed in the literature over the past few years in a chrnological order, and we add some of the works missing in [42] as well and upadte the list. Afterwards, we propose our own solution, SLITHER-SIMILand how it led us to the development of ETHERBASE.

3.2 Traditional Security Analysis Methods in Smart Contracts

Classic software testing technologies applied towards smart contract security analysis can be divided into three categories;

In the following, we will go over the tools proposed from the perspective of the technology they employ to tackle the smart contract security problem; Ren et al. [40] provide three broad categories of tools based on their utilized methodology, namely Static Analysis, Dynamic Fuzzing, and Symbolic Execution.

Using the static analysis method, we are able to analyze the program at both the source code (high-level) and bytecode (low-level) scopes, before conducting any sort of runtime execution. Static analysis-based tools can scan a whole code base, but they also generate a lot of false positives as a result of their scans. There are normally three main stages to a static analysis process:

- building an intermediate representation (IR), such as abstract syntax tree (AST) for a deeper analysis compared to analyzing the raw text / source code;
- complementing the generated IR with additional metadata with methods such as control flow and data flow analysis and symbolic execution.
- vulnerability detection w.r.t. a database of patterns and specific threshold, which define vulnerability criteria.

Tools that leverage the static analysis methods, typically convert the raw form of the input program into an intermediate representation and then perform a series of analyses on those representations based on a pre-defined database of vulnerability patterns and filter out the suspicious snippets of the input program. Slither [16], Securify [45], and SmartCheck [45] are categorised as instances of static analyzers.

Fuzzing [6] is a technique for finding software bugs that involves creating erroneous input data and watching the target program's unusual output while it runs. It allows developers to generate exploits for security-critical programs and ensure a uniform standard of quality through prepared tests, but does not narrow down the causes of detected bugs. When applied to smart contracts, a fuzzing engine will first try to generate initial seeds to form

executable transactions. With reference to the feedback of test results, it will dynamically adjust the generated data to explore as much smart contract state space as possible. Finally, it will analyze the status of each transaction based on the finite state machine to detect whether there is an attackable threat. ContractFuzzer [24], ReGuard [27], and sFuzz [32] are among the most cited smart contract fuzzers.

Symbolic execution is a technique for finding software bugs that involves creating symbolic values and watching the target program's unusual output while it runs. When using symbolic execution to analyze a program, it will use symbolic values as input instead of the specific values during the execution. Tools leveraging this technique explore a state space with a high degree of semantic awareness. [5] Symbolic execution can simultaneously explore multiple paths that the program can take under different inputs, but it also faces unavoidable problems such as path explosion. [40] The symbolic execution tools usually build a control flow graph initially which is based on the Solidity bytecode of the smart contracts being tested. Afterwards, they implement constraints based on the characteristics of smart contract vulnerabilities, and finally use the constraint solver to generate satisfying test cases. Oyente [30], Mythril [8], and Manticore [31] support symbolic execution for smart contracts.

3.3 Deep Learning in Smart Contracts

In this section, we will go over some of the literature focusing their efforts on replacing the existing tools' capabilities explained in the previous section with machine learning-based techniques. Afterwards, we will go over our own developed tool, SLITHER-SIMIL.

There have been a lot of efforts focused towards utilizing ML based techniques in the field of vulnerability discovery and mitigation with a specific focus on the programming language Solidity and its lower level representations.

While existing symbolic tools (like Oyente) for assessing vulnerabilities have shown

to be effective, Goswami et al. said in 2018 that their execution time grows noticeably with the depth of invocations in a smart contract (cite: Grech2019GigaHorse). To create a quicker and more effective replacement for symbolic analysis tools, they suggested an LSTM neural network model to find flaws in ERC-20 smart contracts. This paper's preprocessing procedures were remarkably similar to those employed by citemadmax. A dataset of 165,652 ERC-20 smart contracts was used for training and testing the model, and it contained bytecode data that Maian and Mythril had annotated (statistical code analysis tools). On the testing set, the proposed model had an F1 score of 93.26. Additionally, they have contrasted the time performance of their model with that of Maian and Mythril's symbolic analysis tools (static analysis tools). On a test set of 5,000 random tokens, their suggested model operated in 15 seconds, while Maian and Mythril needed 32,476 and 9,475 seconds, respectively. These findings show a similar advancement over symbolic analysis methods to that shown in citegrech2019gigahorse.

To identify security concerns to smart contracts in 2018, Liao et al. used a sequence learning approach (cite: madmax). The Ethereum blockchain dataset from Google Big Query was used to acquire smart contract data. Ultimately, 620,000 contracts from this source were used to train an LSTM model. Once more, one-hot vectors were used to represent the derived opcodes from the contracts. These vectors were converted into code vectors using embedding methods, resulting in decreased dimensionality and a stronger capability of capturing potential relationships between sequences because this type of representation produces highly sparse and uninformative features. The statistical characteristics of the opcode lengths of contracts that were determined to be vulnerable and safe have been compared as another stage in the preprocessing process. They made the decision to only include contracts that had a maximum opcode length of 1600 since they had noticed that the features of the two categories varied noticeably. Additionally, it was discovered that the dataset's distribution (as labelled by MAIAN) was unbalanced, with non-vulnerable cases

making up 99.03 percent of the dataset. In order to obtain a fair distribution in the training set, all vulnerable contracts were clustered together and oversampled using the Synthetic Minority Oversampling Technique (SMOTE). The outcomes showed that sequential learning methods outperformed symbolic analysis tools. The model earned an F1 score of 86.04 percent and a vulnerability detection accuracy of 99.57 percent.

To improve the vulnerability identification of smart contracts, citeetehrTrust in 2019 presented the SoliAudit concept. To maintain the structure of executions, Solidity's smart contract source code is transformed into an opcode sequence. Each contract is run via a vulnerability scanner and a dynamic fuzzer. The fuzzer (this term was introduced in an earlier paper) will parse the Application Binary Interface (ABI) of a smart contract to extract its declared function descriptions, data types of their arguments, and their signatures, whereas the vulnerability analyzer consists of a static machine learning classifier, which detects vulnerable classes. The detected vulnerable smart contract inputs and functions will then be returned. The creators of citeetehrTrust proposed the concept of a smart contract fuzzer. 13 vulnerabilities were identified by a vulnerability analyst using a set of labels produced by analytical tools like Oyente and Remix. Before utilising these labels to train the opcode sequence data, two different feature extraction techniques were examined. These included word2vec and n-gram with tf-idf. The studies were conducted using the aforementioned approach along with techniques including Gradient Boosting, Support Vector Machine, K-Nearest Neighbor, Decision Trees, Random Forests, and Logistic Regression. A matrix was produced by the latter (word2vec), and a convolutional neural network (CNN) was chosen to train it since it takes the matrix's internal structure into account. However, the results of this feature extraction and training combination were subpar. With an accuracy rate of 97.3 percent and an F1 score of 90.4 percent, Logistic Regression produced the best results for categorising vulnerabilities.

In 2019, In this research, we proposed a machine learning based model to detect security vulnerabilities of smart contracts on the Ethereum platform. We used static code analysis as the underlying technology and trained an array of machine learning models for different security vulnerabilities. Our model was able to find 16 different vulnerabilities with the average accuracy of 95improvement on computational time and resources compared to directly using static code analyzer tools. Checking a large number of smart contracts using different static code analyzers is a huge burden on developers. In addition, they need to learn how each analyzer works and combine the results for a full evaluation. Furthermore, our model can be used to identify security vulnerabilities parallel to the development process of smart contracts, thus decreasing the cost of development by preventing the security vulnerabilities to be introduced in early stages. Our proposed model is also applicable to other languages and platforms since there are no language or platform dependencies in the model. Training the model with different dataset of the attentive language and choose the corresponding static code analyzers and AST builders, new machine learning code analyzers can be generated by following the steps described in Section III.

In 2020, Xing et al. [50] proposed a feature extraction method named slicing matrix. It consists of segmenting the opcode sequences derived from smart contract bytecodes to extract opcode features from each one individually. The purpose of this segmentation is to separate useful and useless opcodes. The extracted opcode features are then combined to form the slice matrix. To carry out a comparative analysis, three models were created. These were namely Neural Network Based on opcode Feature (NNBOOF), Convolution Neural Network Based on Slice Matrix (CNNBOSM), Random Forest Based on opcode Feature (RFBOOF) [23]. These three models were each tested on three different vulnerability classification tasks: greedy contract vulnerability, arithmetic overflow/underflow vulnerability and short address vulnerability. While RFBOOF achieved the best results in all three cases based on precision, recall and F1 evaluation metrics, CNNBOSM performed

slightly better than NNBOOF in general. The authors mention that the slice matrix feature need further exploring.

In 2020, Ethereum has established itself as a popular platform for facilitating safe, Blockchain-based financial and commercial transactions. However, the security of Ethereum's smart contracts is a significant issue. Numerous discovered flaws and weaknesses in smart contracts not only complicate the upkeep of the blockchain but also result in significant financial losses. Better tools are needed to help developers verify smart contracts and ensure their dependability. In this article, we suggest SMARTEMBED, a web service tool that can assist Solidity developers in identifying repeated contract code and clone-related problems in smart contracts. Our technology is based on methods for comparing codes and code embeddings. We are able to effectively identify code clones and clone-related bugs for any solidity code provided by users, which can help to increase the users' confidence in the reliability of their code. We do this by comparing the similarities between the code embedding vectors for existing solidity code in the Ethereum blockchain and known bugs. SMARTEMBED can be used for studies of smart contracts on a large scale in addition to uses by specific developers. We discovered that solidity code has a substantially higher clone ratio than traditional software when applied to more than 22K contracts taken from the Ethereum blockchain. Based on our modest bug database, 194 clone-related defects can be efficiently and effectively diagnosed with a precision of 96

For the purpose of identifying various smart contract vulnerabilities, Liu Z. et al. designed a machine learning approach combining GNN and expert knowledge (cite:hwang2020gap). The goal of a graph neural network (GNN), a deep learning technique, is to make inference on data that is represented by graphs. A graph is a type of data structure used in computer science that consists of nodes (also known as vertices) and edges. The semantic relationships between programming elements can be preserved when written programmes are transformed to symbolic graph representation, according to research. As a result, contract

graphs can be used to represent smart contract codes. The suggested strategy, as depicted in figure 3, begins with two distinct concurrent processes (Security pattern extraction and contract graph extraction), and then uses a combining layer to combine patterns in each segment to identify vulnerabilities. The pattern feature for extracting security patterns from the contract's source code is first created by a feed-forward neural network. To extract the expert patterns from smart contract functions, they employed an open-source programme. To produce a contract graph, a GNN must be created in the second process (message propagation phase). Nodes, or programme elements, made up the GNN model, while edges, or the next function to be executed, indicated the flow of each programme element. Later, using a node elimination approach, undesirable nodes and edges are eliminated. The authors used a preprocessing technique that involved casting the source code's extensive control and data flow semantics into a contract graph. Following this, they created a node elimination stage to normalise the network and highlight important nodes. The vulnerability detection phase, where both extracted features are integrated with convolution and full-connected layer, was used to combine these two simultaneous procedures. The suggested model is tested against security detection methods that are not ML-based, including Oyente, Myhrill, Smartcheck, Securify, and Slither. Re-entrancy, timestamp dependence, and endless loop vulnerabilities of each function in the source code were all searched for by each algorithm and the proposed model. a The proposed methods (CGE) found re-entrancy and timestamp dependent type vulnerabilities with an accuracy of 89 percent and an infinite loop vulnerability (cite:hwang2020gap) with an accuracy of 83 percent.

When a piece of code is rewritten, the Eth2Vec model is suggested to address a flaw in the present vulnerability detection tools. A code rewrite in a programming language is the process of reimplementing a source code's functionality without utilising the original. Finding vulnerabilities becomes more difficult when the smart contract codes are modified. Each smart contract's source code was first transformed into EVM bytecodes by the

authors. Only useful information (such as function ids, lists of callee functions, etc.) for vulnerability detection was taken out of the bytecode by the authors. A neural network structure is utilised as the final step to find any vulnerabilities in the source code. 500 contracts were used to test the suggested model, and even though the contracts were changed, the Eth2Vec model was able to identify vulnerabilities with a 77 percent accuracy.

O. Lutz et al. cite Dolan2016lava and present another technique for identifying weaknesses in smart contracts. The authors offer a method called ESCORT, in which they employ a Deep Neural Network model to discover the semantics of the input smart contract and identify particular vulnerability types based on the discovered semantics. The ESCORT model aims to get over the limitations of existing non-DNN models in terms of scalability and generalisation. With a detection period of 0.02 seconds for each contract, the experimental results of this article produced an F1 accuracy score of 9 percent on six different vulnerability types. Scalability is easier to achieve with such rapid detection times, which meets one of the author's objectives. The ESCORT model therefore somewhat resolves the problems identified in earlier works, such as Y. Xu or N. Lesimple's models, where novel weaknesses can be realised by the model, cite:grech2019gigahorse. Unfortunately, it is relatively challenging to derive interpretability from such models, and even if new vulnerabilities are discovered, figuring out their root causes is still very challenging. Sun et al. used machine learning to try and find the following vulnerabilities: re-entrancy, arithmetic problems (integer overflow/underflow), and timestamp dependencies. In order to accommodate for differences in instructions between compilers, some stackoperating instructions were trimmed into more general versions (e.g., SWAP1, SWAP2,..., SWAPn. SWAPx). After that, as a label normalisation step, opcodes were divided into 9 categories depending on their purposes. A word2vec transformation of the opcode sequences prior to the convolutional layers was carried out, just like in "cite"etehrTrust. This article introduces an additional self-attention layer in addition to the pooling and softmax layers that

typically follow convolutional layers. The one-hot encoders that were used to encode each opcode instruction are just merely representatives and do not capture any functional similarity between them, therefore the self-attention layer's goal is to establish a connection between adjacent words in the acquired feature matrix. [20]. By using self-attention, the word embedding process has been improved as a result. CiteetehrTrust and this paper both used CNNs to find vulnerabilities, but CiteetehrTrust used a word2vec embedding, whereas this paper used an attention method, which is probably why they got superior results. gotten better outcomes. The key advantage of the newly developed model over the static analyzers that are already in use, such as Oyente and Mythril, is that it can attain comparable performance in a lot less time. In their paper, Y. Xu et al. developed two unique methods for identifying smart contracts that are vulnerable utilising both the Stochastic Gradient Descent (SGD) model and the KNearest Neighbors (KNN) model. They seek to use each of the machine learning models to discover eight of the most widely known classical vulnerability types, including arithmetic, reentrancy, denial of service, uncontrolled low level calls, access control, faulty randomization, front running, and denial of service. Similar to N.Lesimple's paper [22], their model employs an AST structure as its input, enabling it to parse the smart contract code line by line. Through the use of conventional techniques, the labels for the vulnerabilities were found. High recall, precision, and accuracy are noted for four of the eight vulnerabilities in the paper. The results for the remaining four were deemed inconclusive since there weren't enough samples in the dataset. The test set was produced from the outcomes of utilising conventional approaches, similar to the N. Lesimple study, showing that the authors were unable to demonstrate how the KNN model differed from conventional methods.

In 2021, through using bigram properties from the streamlined operation codes of smart contracts, Wang et al. [47] introduced their own approach, ContractWard, to identify vulnerabilities in smart contracts. They gathered a dataset of 49,502 smart contracts from the

Etherscan website, verified before September 2018, and found that each contract had six potential weaknesses: integer overflow/underflow, transaction ordering dependency, call stack depth attack, timestamp dependency, and re-entrancy. Each smart contract's source code is converted to opcodes; A smart contract typically has 100 different types of opcodes and 4364 opcode components. There were only 50 opcode types left after they conducted a simplifying process. As a result, the authors grouped several opcodes with related functionality into a single category, which simplified the dataset's features. Because they believe that operations have a stronger relationship with their neighbours, they later adopted the n-gram approach (a sliding window of binary-byte size) to track relationships between each opcode. Each of the contract's many labels was assigned using the Oyente [30] system. Due to the scarcity of particular vulnerabilities, the researchers ran into a class-imbalance problem after the labelling process. Extreme Gradient Boosting (XGBoost), Adaptive Boosting (AdaBoost), Random Forest (RF), Support Vector Machine (SVM), and k-Nearest Neighbour were the 5 candidate ML models used in the training procedure (KNN). By reaching above 96 percent F1, Micro-F1, and Macro-F1 score, the XGBoost model demonstrated a strong performance.

In 2022, Yuqi Fan et al. note that most of the studies on methods of vulnerability detection regarding smart contracts take rely on pre-defined manual rules from experts and auditing professionals. [15] Devising such rules and patterns are very time-intensive and labor-demanding. They discuss the previous efforts at employing deep learning methods to make up for such shortcomings, but that they fail to represent the source code / bytecode good enough both semantically and structurally. Then, they go ahead and propose a novel model of Dual Attention Graph Convolutional Network (DA-GCN) to detect to detect smart contract vulnerabilities. They extract both control flow graph and opcode sequence from smart contracts' bytecodes and feed them as input into a feature extraction pipeline and afterwards, they use a multilayer neural network to identify the vulnerable

smart contracts. They tested their proposed model on smart contracts containing one of the two vulnerabilities: reentrancy and timestamp dependency. Their experimental results demonstrated that the DA-GCN model achieved an accuracy of 91.2% and 87.5% in the two smart contract vulnerability detection tasks.

In 2022, Zhang et al. [53] propose a novel model to detect smart contract vulnerabilities: ensemble learning (EL)-based contract vulnerability prediction method. It is based on seven different neural networks using vulnerability data for vulnerability detection at the scope of smart contracts (single file-level scope). Seven neural network (NN) models were first pretrained using an information graph (IG) consisting of source datasets, which then were integrated into an ensemble model called Smart Contract Vulnerability Detection method based on Information Graph and Ensemble Learning (SCVDIE). The effectiveness of the SCVDIE model was verified using a target dataset composed of IG, and then its performances were compared with static tools and seven independent data-driven methods. The verification and comparison results show that the proposed SCVDIE method has higher accuracy and robustness than other data-driven methods in the target task of predicting smart contract vulnerabilities.

Also in 2022, Zhang et al. [52] propose another novel method for vulnerability detection: a flexible and systematic hybrid model, which they have named as the Serial-Parallel Convolutional Bidirectional Gated Recurrent Network Model, incorporating Ensemble Classifiers (SPCBIG-EC). Their new model shows noticeable improvements in performance concerning smart contract vulnerability detection. In addition, they also propose a serial-parallel convolution (SPCNN) suitable for this hybrid model and generally serial combinatorial models. It is equipped with the capability to extract features from an input sequence for multivariate combinations while retaining temporal structure and location information. The Ensemble Classifier is used in the classification phase of the model to enhance its robustness. They put their focus on six typical smart contract vulnerabilities

and constructed two datasets, CESC and UCESC, for multi-task vulnerability detection in their experiments. Numerous experiments showed that their proposal is better than most existing methods. It achieved an F1-scores of 96.74%, 91.62%, and 95.00% for reentrancy, timestamp dependency, and infinite loop vulnerability detection.

3.4 SLITHER-SIMIL

Parts of this section have been published in another piece [37] written by the same author of this dissertation and have been used here with permission.

The efforts of security auditing companies like Trail of Bits, Inc. with regard to automating smart contract security assessments has included works on an addition to an already prominent static analysis tool, Slither [16], to better help developers and researchers in their process of auditing smart contracts.

Trail of Bits, a prominent blockchain security firm has manually curated a wealth of data—years of security assessment reports—and we decided to explore how to use this data to make the smart contract auditing process more efficient with an addition to Slither—a static analysis tool—named SLITHER-SIMIL.

Based on accumulated knowledge embedded in previous audits, we set out to detect similar vulnerable code snippets in new clients’ codebases. Specifically, we explored machine learning (ML) approaches to automatically improve on the performance of Slither, our static analyzer for Solidity, and facilitate the conduct of audits for auditors and general users.

Currently, human auditors with expert knowledge of Solidity and its security nuances scan and assess Solidity source code to discover vulnerabilities and potential threats at different granularity levels. In our experiment, we explored how much we could automate security assessments to:

1. Minimize the risk of recurring human error, i.e., the chance of overlooking known, recorded vulnerabilities.
2. Help auditors sift through potential vulnerabilities faster and more easily while decreasing the rate of false positives.

SLITHER-SIMIL [38], the statistical addition to Slither, is a code similarity measurement tool that uses state-of-the-art machine learning to detect similar Solidity functions. When it began as an experiment last year under the codename *critic-pred*, it was used to vectorize Solidity source code snippets and measure the similarity between them. Last year, we took it to the next level and applied it directly to vulnerable code.

SLITHER-SIMIL currently uses its own representation of Solidity code, as introduced by [16], namely SlithIR. SlithIR (Slither Intermediate Representation), to encode Solidity snippets at the granularity level of functions. We thought function-level analysis was a good place to start our research since it's not too coarse (like the file level) and not too detailed (like the statement or line level).

As introduced by Feist et al, SlithIR was developed as an intermediate representation (IR) language with slither in mind to leverage it and represent Solidity code for further analysis. Every smart contract written in solidity can be decomposed into a control flow graph and in that graph, each node can contain up to a single Solidity expression, which is converted to a set of SlithIR instructions. This representation makes implementing analyses easier, without losing the critical semantic information contained in the Solidity source code. [16] SlithIR has a database of about 40 instruction expressions. It has no internal control flow representation and relies on Slither's control-flow graph structure (SlithIR code is associated with each node in the graph). The complete descriptions is available at [34].

In the process workflow of Slither-simil, we first manually collected vulnerabilities from the previous archived security assessments and transferred them to a vulnerability database. Note that these are the vulnerabilities auditors had to find with no automation.

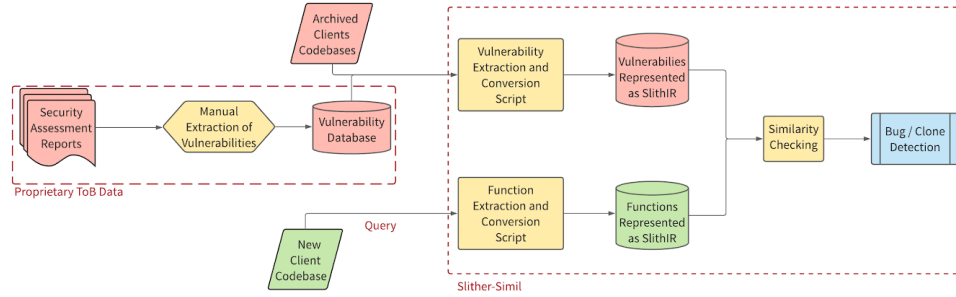


Figure 3: A high-level view of the process workflow of Slither-simil.

```

1  function transferFrom(address _from, address _to, uint256
   _value) public returns (bool success) {
2      require(_value <= allowance[_from][msg.sender]);
   // Check allowance
3      allowance[_from][msg.sender] -= _value;
4      _transfer(_from, _to, _value);
5      return true;
6  }
7

```

Listing 3.1: complete Solidity function from the contract TurtleToken.sol.

After that, we compiled previous clients' codebases and matched the functions they contained with our vulnerability database via an automated function extraction and normalization script. By the end of this process, our vulnerabilities were normalized SlithIR tokens as input to our ML system.

Here's how we used Slither to transform a Solidity function to the intermediate representation SlithIR, then further tokenized and normalized it to be an input to SLITHER-SIMIL:

First, we converted every statement or expression into its SlithIR correspondent, then tokenized the SlithIR sub-expressions and further normalized them so more similar matches

```

1 Function TurtleToken.transferFrom(address, address, uint256)
   (*)
2
3
4 Solidity Expression: require(bool) (_value <= allowance[_from
   ][msg.sender])
5 SlithIR:
6     REF_10(mapping(address => uint256)) -> allowance
   [_from]
7     REF_11(uint256) -> REF_10[msg.sender]
8     TMP_16(bool) = _value <= REF_11
9     TMP_17 = SOLIDITY_CALL require(bool) (TMP_16)
10
11
12 Solidity Expression: allowance[_from][msg.sender] -= _value
13 SlithIR:
14     REF_12(mapping(address => uint256)) -> allowance[
   _from]
15     REF_13(uint256) -> REF_12[msg.sender]
16     REF_13(-> allowance) = REF_13 - _value
17
18
19 Solidity Expression: _transfer(_from,_to,_value)
20 SlithIR:
21     INTERNAL_CALL,      TurtleToken._transfer(address,
   address,uint256) (_from,_to,_value)
22
23
24 Solidity Expression: true
25 SlithIR:
26     RETURN True
27

```

Listing 3.2: The same function with its SlithIR expressions printed out.

```

1 type_conversion(uint256)
2
3 binary(**)
4
5 binary(*)
6
7 (state_solc_variable(uint256)) := (temporary_variable(uint256)
  )
8
9 index(uint256)
10
11 (reference(uint256)) := (state_solc_variable(uint256))
12
13 (state_solc_variable(string)) := (local_solc_variable(memory,
  string))
14
15 (state_solc_variable(string)) := (local_solc_variable(memory,
  string))
16
17 ...
18

```

Listing 3.3: Normalized SlithIR tokens of the previous expressions.

would occur despite superficial differences between the tokens of this function and the vulnerability database.

After obtaining the final form of token representations for this function, we compared its structure to that of the vulnerable functions in our vulnerability database. Due to the modularity of Slither-simil, we used various ML architectures to measure the similarity between any number of functions.

Let's take a look at the function `transferFrom` from the `ETQuality.sol` smart contract to see how its structure resembled our query function:

Comparing the statements in the two functions, we can easily see that they both contain, in the same order, a binary comparison operation (`>=` and `<=`), the same type of operand comparison, and another similar assignment operation with an internal call statement and

an instance of returning a “true” value.

As the similarity score goes lower towards 0, these sorts of structural similarities are observed less often and in the other direction; the two functions become more identical, so the two functions with a similarity score of 1.0 are identical to each other.

Research on automatic vulnerability discovery in Solidity has taken off in the past two years, and tools like Vulcan [41] and SmartEmbed [?], which use ML approaches to discovering vulnerabilities in smart contracts, are showing gradually more and more promising results, with less false positives in specific vulnerability reports.

However, all the current related approaches focus on vulnerabilities already detectable by static analyzers like Slither and Mythril, while our experiment focused on the vulnerabilities these tools were not able to identify—specifically, those undetected by Slither.

Much of the academic research of the past five years has focused on taking ML concepts (usually from the field of natural language processing) and using them in a development or code analysis context, typically referred to as code intelligence. Based on previous, related work in this research area, we aim to bridge the semantic gap between the performance of a human auditor and an ML detection system to discover vulnerabilities, thus complementing the work of Trail of Bits human auditors with automated approaches (i.e., Machine Programming, or MP [19]).

We still face the challenge of data scarcity concerning the scale of smart contracts available for analysis and the frequency of interesting vulnerabilities appearing in them. We can focus on the ML model because it’s a more facilitated process but it doesn’t do much good for us in the case of Solidity where even the language itself is very young and we need to tread carefully in how we treat the amount of data we have at our disposal.

Archiving previous client data was a job in itself since we had to deal with the different solc versions to compile each project separately. For someone with limited experience in that area this was a challenge, and I learned a lot along the way. (The most important

takeaway of my summer internship is that if you're doing machine learning, you will not realize how major a bottleneck the data collection and cleaning phases are unless you have to do them.)

This past summer we resumed the development of Slither-simil and SlithIR with two goals in mind:

Research purposes, i.e., the development of end-to-end similarity systems lacking feature engineering. Practical purposes, i.e., adding specificity to increase precision and recall. We implemented the baseline text-based model with FastText to be compared with an improved model with a tangibly significant difference in results; e.g., one not working on software complexity metrics, but focusing solely on graph-based models, as they are the most promising ones right now.

For this, we have proposed a slew of techniques to try out with the Solidity language at the highest abstraction level, namely, source code.

To develop ML models, we considered both supervised and unsupervised learning methods. First, we developed a baseline unsupervised model based on tokenizing source code functions and embedding them in a Euclidean space (Figure 8) to measure and quantify the distance (i.e., dissimilarity) between different tokens. Since functions are constituted from tokens, we just added up the differences to get the (dis)similarity between any two different snippets of any size.

The diagram below shows the SlithIR tokens from a set of training Solidity data spherized in a three-dimensional Euclidean space, with similar tokens closer to each other in vector distance. Each purple dot shows one token.

We are currently developing a proprietary database consisting of our previous clients and their publicly available vulnerable smart contracts, and references in papers and other audits. Together they'll form one unified comprehensive database of Solidity vulnerabilities for queries, later training, and testing newer models.

We're also working on other unsupervised and supervised models, using data labeled by static analyzers like Slither and Mythril. We're examining deep learning models that have much more expressivity we can model source code with—specifically, graph-based models, utilizing abstract syntax trees and control flow graphs.

And we're looking forward to checking out Slither-simil's performance on new audit tasks to see how it improves our assurance team's productivity (e.g., in triaging and finding the low-hanging fruit more quickly). We're also going to test it on Mainnet when it gets a bit more mature and automatically scalable.

You can try SLITHER-SIMIL now on Github. For end users, it's the simplest CLI tool available: The user can input one or multiple smart contract files (either directory, .zip file, or a single .sol). Identify a pre-trained model, or separately train a model on a reasonable amount of smart contracts.

3.5 Concluding Remarks

In this chapter, we went over the efforts of the research community targeted at proposing machine-learned-based methods to facilitate the process of discovering and mitigating vulnerabilities in smart contracts, compared to the existing approaches used in industry. We also went over SLITHER-SIMIL; a powerful tool with potential to measure the similarity between function snippets of any size written in Solidity. We are continuing to develop it, and based on current results and recent related research, we hope to see impactful real-world results before the end of the year. But there is a lacking here and that is the bottleneck of datasets that help us train and test bigger and more comprehensive models. In the next chapter we will introduce ETHERBASE and how the development of SLITHER-SIMIL has extended into the development of ETHERBASE as a solution for us and the broader research community.

Chapter 4

ETHERBASE: Improving Reproducibility in Smart Contract Research

4.1 Introductory Remarks

Ethereum is the most widely used blockchain platform with millions of smart contracts written on it, with a market cap of over 250 billion U.S. Dollars. Simply put, a smart contract is a self-executing program stored on the Ethereum blockchain that runs when pre-defined conditions are satisfied. The research community in the past few years has put in the effort to develop automated analysis tools and frameworks [25] that locate and eliminate vulnerabilities in smart contracts for the purposes of making smart contracts more secure as their adoption is ever increasing in different sectors of decentralized finance.. In a study in 2020, researchers analyzed about one million Ethereum smart contracts and found 34,200 of them to be potentially vulnerable. [33] Another research effort showed that 8,833 (around %46) smart contracts on the Ethereum blockchain were flagged as vulnerable out

of 19,366 smart contracts. [29], [40]

Comparing and reproducing such research is not an effortless process. [40] Most of the datasets used to test and benchmark those very same tools proposed in the research literature are not publicly or easily available to the research community. This makes reproduction efforts immensely hard and time-intensive to carry out.

The current approach to compare one’s tools / methodologies of evaluating the security of smart contracts with that of another researcher / toolset, is to make do with whatever out-of-date incomprehensive and unrepresentative dataset they have at their disposal, a very timely and inefficient process. In other cases, the researchers need to start from scratch and create their own datasets, a non-trivial and slow process. What makes it worse is the data bias, which can be introduced in a dataset in different phases of data acquisition and data cleaning. This can easily escalate to become a threat to validity in the research. [40]

In this chapter, we present ETHERBASE, an extensible, and queryable database that facilitates and enhances the verification and reproducibility of previous empirical research and lays the groundwork for faster, more rapid production of research on smart contracts. ETHERBASE is open-source and publicly available. The source code for data acquisition and cleaning is not available due to private IP reasons, and only the collected data and the database is accessible to the public. Researchers, smart contract developers, and blockchain-centric teams and enterprises can also use such corpus for specific use-cases.

In summary, we make the following contributions with ETHERBASE.

1. We propose ETHERBASE, a systemic and up-to-date database for Ethereum by exploiting its internal mechanisms.
2. We implement ETHERBASE and make its pipelines open-source. It obtains historical data and facilitates benchmarking and reproduction in research and development for new toolsets. It is more up-to-date than existing datasets, gets automatically reviewed and renewed, in comparison to the previous manual one-time data gathering efforts.

3. We propose the first dataset of Ethereum smart contracts which has a mix of off-chain and on-chain data together, meaning that it contains the source code and the bytecode of the corresponding smart contracts in the same dataset.
4. We propose the first automatically up-to-date labelled dataset of Ethereum smart contracts with vulnerabilities.

4.2 Related Work

We assume the reader is familiar with blockchain technology, Ethereum blockchain, and its primary high-level programming language Solidity. Ethereum Smart contracts are developed mostly in the programming language Solidity and for execution, they get compiled to their corresponding bytecode through EVM. EVM takes bytecode as input and works in a stack-based architecture with a word size of 256 bits. [28] In this section we present and discuss references and information for some of the more prominent publicly available smart contract benchmark datasets which were identified in our studies.

SmartBugs [11] makes the top of the list as one of the most used benchmarks in the research space. In this work, Durieux et al. presented an extensible and easy-to-use execution framework for benchmarking different security analysis tools for Ethereum smart contracts. They empirically evaluated 9 analysis tools on a small (< 100) labelled dataset of vulnerabilities. Ren et al. provide a dataset consisting of 47,518 un-annotated smart contracts with the history of at least one executed transaction on the blockchain. For researchers trying to build upon such work, using the unlabelled dataset is a great improvement for both tool development and reproduction efforts, but because of the lack of some degree of ground truth in the dataset, it's not possible to easily apply, for example, supervised machine learning algorithms to that dataset. Their labeled dataset contains about a thousand smart contracts which is good for initial benchmarkings, but it only the source code of the contracts are

available and does not provide any bytecode for them.

Ren et al. [40] created a benchmark suite that integrates annotated and unlabelled raw smart contracts from a variety of sources such as Etherscan [14], SolidiFI repository, CVE Library, and Smart Contract Weakness (SWC) Registry. They collected 45,622 real-world diversified Ethereum smart contracts and proposed a systematic evaluation process and performed extensive experiments. Their labeled dataset has 350 manually generated contracts.

Many of the authors of the tools proposed in this practice leverage unlabelled datasets to evaluate the performance of their toolset. The publicly available datasets are either not appropriately annotated or a very small subset of them is labelled with the vulnerabilities's identification properties. This is because the number of the contracts -most of them being clones of other smart contracts- is so huge, they cannot be all manually annotated. To the best of our knowledge, there has been only one annotated dataset, released publicly, from Yashavant et al. [51] We strive to build upon that work and make facilitate the processes of reproduction in the smart contract research community. Our dataset has the advantage of updating regularly, and being more comprehensive with the more updated Solidity versions of different smart contracts available in the dataset.

Kalra et al. [26] published their analysis results for 1,524 smart contracts with no other information or metadata in relation to those contracts.

Luu et al. [30] collected 19,366 smart contracts from the blockchain and provided their blockchain addresses alongside analysis results on each contract on whether they contain any of their selected four vulnerabilities or not.

In 2022, Yashavant et al. [51]

The smart contract source codes collected in GitHub repositories associated with the previously published literature do not directly reference smart contracts deployed on the blockchain through an Ethereum address; [36] this makes it hard to determine if the identified smart contracts have been tested or used on the Ethereum blockchain or not. GitHub

repositories of the available datasets do not implement a search engine or query capability to filter smart contracts based on particular metrics or parameters, such as the ETH value or the number of internal transactions for a smart contract.

The GitHub repositories related to the published literature usually only provide the raw data of the smart contracts without any proper documentation, comments, or further annotations on the data connected to the collected smart contracts. None of the GitHub repositories currently available to the public or used in research papers provide smart contract ABI's or Opcodes to the best of our knowledge.

As a user of the Website Etherscan, you can easily search the Ethereum blockchain for any specific smart contract given the availability of its address but when it comes to downloading the data for that smart contract or any other batch of smart contracts, using Etherscan has its limits: [36]: Smart contracts' data is massive, based on the estimation from [36], and the daily limits on the API's provided by Etherscan make retrieving that data even harder. The API provided by Etherscan does not allow the users to obtain a list of the addresses of their desired smart contracts. The API calls currently available only allow navigation at the block-level. A researcher cannot easily explore the source code for their collected smart contracts. First, they would have to inspect any block on-chain and then search for any transactions that have a receive/send address associated with that specific smart contract.

4.3 Methodology

We specifically designed ETHERBASE to not provide the end-users with a dump of contracts and their corresponding features in a vague file hierarchy. We have a service offering the ability to filter and analyze smart contracts and a dataset of smart contracts with interesting features to conduct empirical research on, according to metrics like Pragma version, ETH value, etc. To fulfill its purpose, ETHERBASE is designed to perform four primary automatic



Figure 4: EtherBase Workflow

operations on the data:

1. **Data Acquisition:** Automatic retrieval of off-chain and on-chain data
2. **Data Generation:** Generation of bytecode and other metrics
3. **Data Storage:** Storage of the collected data in a public and accessible way
4. **Data Application:** Development of new applications and research based on the available database

The next sections walk through the above-mentioned steps of the workflow one-by one.

4.3.1 Data Acquisition

The current dataset corresponds to the contracts collected from Etherscan, the most used service for researchers trying to collect smart contracts from the Ethereum blockchain. Every contract stored on Etherscan's database is indexed by its own corresponding addresses. In order to collect the contracts, we retrieve the addresses for every contract which has more than one transaction through Google BigQuery, like the process done by [40]. Using Google BigQuery query request service, we obtain 1,712,347 distinct contract addresses that have more than one transactions associated with them.

4.3.2 Data Generation

Instead of manually writing scripts to obtain the source code, bytecode, and other metadata via services like Etherscan, we leverage a tool designed and maintained by Trail of Bits, namely Cytic-compile. All the previous works of research, work on providing either source code or bytecode of smart contracts to the researchers, but that will not be enough for the users who want to compile the smart contracts or build tools upon such data. Compiling smart contracts is also a tricky business, due to the rapid pace of changing versions of the dominant programming language, Solidity, used to write and develop smart contracts. We developed crytic-compile, a library to help the compilation of smart contracts, to help with this problem. It helps the user to be needless of maintaining an interface with solc and it automatically finds and uses the right version of solc or a better compatible version of solc to compile the input smart contracts. The way this works under the hood is that crytic-compile compiles an input smart contract and outputs a compilation unit in the standard solc output format, written in a json file, alongside the source code and other metadata. Unlike the other proposed datasets and tools discussed in Section 4.2, EtherBase targets the core problem of the lack of reproducibility in the research literature. Discussions surrounding what types of secondary metrics to retrieve from the Ethereum blockchain or third-party services will be explored further.

For the rest of the analysis of the collected contracts, we only get to work on contracts with available source-code. We adopt the method used in the work of [12] to remove the duplicates smart contracts, through checking the MD5 checksums (32-character hexadecimal numbers computed for each file) of each of the two source files in the collected dataset to see if they are the same and after removing the whitespace among the lines of code. After the process of deduplication is done, we get down to 48,622 smart contracts. For this paper and as of now, we have released 5,000 smart contracts for the tool comparison purposes. There exist many metrics that we have access to, can calculate, and

add to ETHERBASE, but not a lot of research has been conducted on the applicability of these many different types of metrics to empirical research on the Ethereum blockchain or how much it appeals to the researchers active in this field. In the following, we describe the initial set of the metrics we selected to include in EtherBase in the form of a table, to be followed by more, after more discussion and research on their applicability to research on smart contracts.

The built-in metrics relating to smart contracts are those features which depend on the internal properties of a smart contract, e.g. SLOC (Source Lines of Code), Pragma version, number of modifiers, payable, etc. Hence the title *primary metrics*.

For this table, we decided to include the core metrics of a smart contract, which would help a researcher collect a large set of smart contracts rapidly, and conduct further analysis on them, comprising of:

Table 1: Primary Metrics on Smart Contracts

Name	Description
Pragma	The <code>pragma</code> keyword is used to enable certain compiler features or checks. [2]
Contract Address	Unique 20-byte address, used as the main index to distinguish smart contracts from each other.
Creator Address	Indicates the address of the deeployer of the smart contract.
Source Code	Source code of the smart contract, specific to the programming language Solidity.
Bytecode (bin)	.
Bytcode (bin-runtime)	.
ABI	The content of the application binary interface for each contract.
Block Number	The length of the blockchain in blocks.
ETH Value	The value of each smart contract in therms f=of the ETH thay hold.
Transaction Count	Number of internal transactions from eacch smart contract.

As for the primary metrics, here are our justifications for the above selection:

- **Pragma:** Source files can (and should) be annotated with a version pragma to halt compilation with future versions of Solc due to the possibility of introduction of incompatible changes with the version of the Solc used to write the original smart contract. Filtering through contracts via Pragma helps the researcher to collect a homogenous set of contracts with a consistent syntax.

- `Contract Address`: Contract address is basically the main key in the database of ETHERBASE for distinguishing smart contracts from each other. It is usually given when a contract is deployed to the Ethereum.
- `Creator Address`: The contract address is usually given when a contract is deployed to the Ethereum Blockchain. The address comes from the creator's address, where the contract has been initially deployed from, alongside the number of transactions sent from that address (the "nonce"). [3]
- `Source Code`: The source code of each Ethereum smart contract is written in Solidity and helps researchers do all sorts of analysis on the smart contracts.
- `Bytecode (bin)`: The regular bin output is the code placed on the blockchain plus the code needed to get this code placed on the blockchain, the code of the constructor.
- `Bytecode (bin-runtime)`: bin-runtime is the code that is actually placed on the blockchain.
- `ABI`: ABI stands for Application Binary Interface. It is the standard way of interaction with smart contracts on Ethereum, both from outside the blockchain ecosystem and for contract-to-contract interaction. [4]
- `Block Number`: Block Number is the length of the blockchain in blocks, more

specifically the block on which the smart contract exists.

- `ETH Value`: The ETH every smart contract holds is an excellent filter or bar to select "interesting" contracts through for further research on the contracts that are more *active* on the blockchain.
- `Transaction Count`: Like ETH Value, transaction count is an important metric for us to be able to exclude contracts that do not participate much on the chain and hence, work on the contracts that have a higher probability of interaction with more contracts.

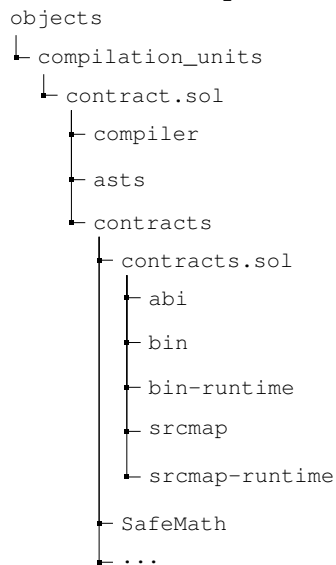
4.3.3 Data Storage

All of the smart contracts collected in the previous stage along with their corresponding metadata need to be stored somewhere and we choose a PostgreSQL database in the design, -alongside a GitHub repository- to make it easier for researchers and other users to manage, filter, and query their needed data. Afterwards and in the data application stage, users can analyse their queried data according to their specific research queries.

Figure 2 showcases the collected data in the form of a directory tree structure. The first leaf in the directory `Contracts` corresponds to the



The artifact.zip file contains



In addition to making the datasets available on GitHub, ETHERBASE also enjoy a graphical user interface (GUI) in order to allow the less technical end users access and browse through the database. We integrated ETHERBASE with Apache Superset, a powerful business intelligence tool, which lets you create charts and dashboards using the data from the database.

4.3.4 Data Application

In order to showcase an application of the empirical usage of the data from ETHERBASE, the 5,000 filtered smart contract data set is labelled using three of the most prominently used static analysis tools in Ethereum research that detect various vulnerabilities in smart contracts, using a majority voting mechanism, in order to see how they fare against each other based on an automatically labelled dataset. The criteria we used for tool selection was pretty simple; we wanted tools that had a focus on assessing Solidity source code instead of bytecode, and that they are available as open-source software and can be evaluated based on their vulnerability detection mechanisms. Based on such criteria, we selected the following three tools for our Data Application phase experiment:

- **Smartcheck:** Smartcheck [44] is an extensible static analysis tool written in Java. It detects vulnerabilities and other code issues in Ethereum smart contracts. It locates vulnerabilities by searching for pre-defined patterns in a transformed version of the Solidity source code of the contract.
- **Mythril:** Mythril is another frequently used static analyzer in the form of CLI tool developed in Python that does security analysis of Ethereum smart contracts.
- **Slither:** Slither [16] is a static analyser for analyzing Ethereum smart contracts before deploying them and evaluating them in runtime.

We select three of the highest ranked vulnerabilities according to the DASP 10 ranking by the NCC Group, to test the aforementioned tools based upon. The three vulnerabilities, as explained in Chapter 2, are as follows:

- **Re-entrancy** also known as the recursive call vulnerability, with SWCRegistry ID SWC-107.
- **Arithmetic:** concerning the integer overflows and underflow vulnerabilities in smart contracts, with SWCRegistry ID SWC-101.

Table 2: Supported Vulnerabilities

Tool Name	Vulnerability Type		
	ARTHM	RENT	UE
Slither	×	✓	✓
Mythril	✓	✓	✓
Smartcheck	✓	×	✓

- **Unchecked Ether:** also known as silent failing sends, this vulnerability can cause unexpected / undefined behavior if before executing the smart contracts, the return values are not managed properly. [21] The SWC Registry ID of this contract is SWC-104. [7]

Like the work done by [51], we also leverage the methodology given in the paper by Ren et al. [39] to detect the selected vulnerabilities in smart contracts.

When using tools like these static analyzers, we face a lot of false positive results as a necessity of the methods those tools employ. Because of that, we cannot rely on one tool only, as projects that rely on auditing their smart contracts for a certain guarantee of security also try and test with multiple tools and analysis methodologies. We use the methodology proposed by /citeyashavant2022scrawld, namely, the majority voting, that is, at least half of the tools being benchmarked should locate the very same vulnerability at the same location. For example, assume that all of the three selected static analyzers are capable of detecting a specific vulnerability. Assuming that at least two of them warn the user that that specific vulnerability is present in a smart contract, then we are allowed to report that that smart contract contains the vulnerability;

Based on the proposal from [51], the following explains the step-by-step methodology concerning the majority voting mechanism as mentioned earlier:

1. Collect the output of the selected static analysis tools for further analysis as the initial step, per vulnerability and identify the LOC in which the vulnerability happens on.

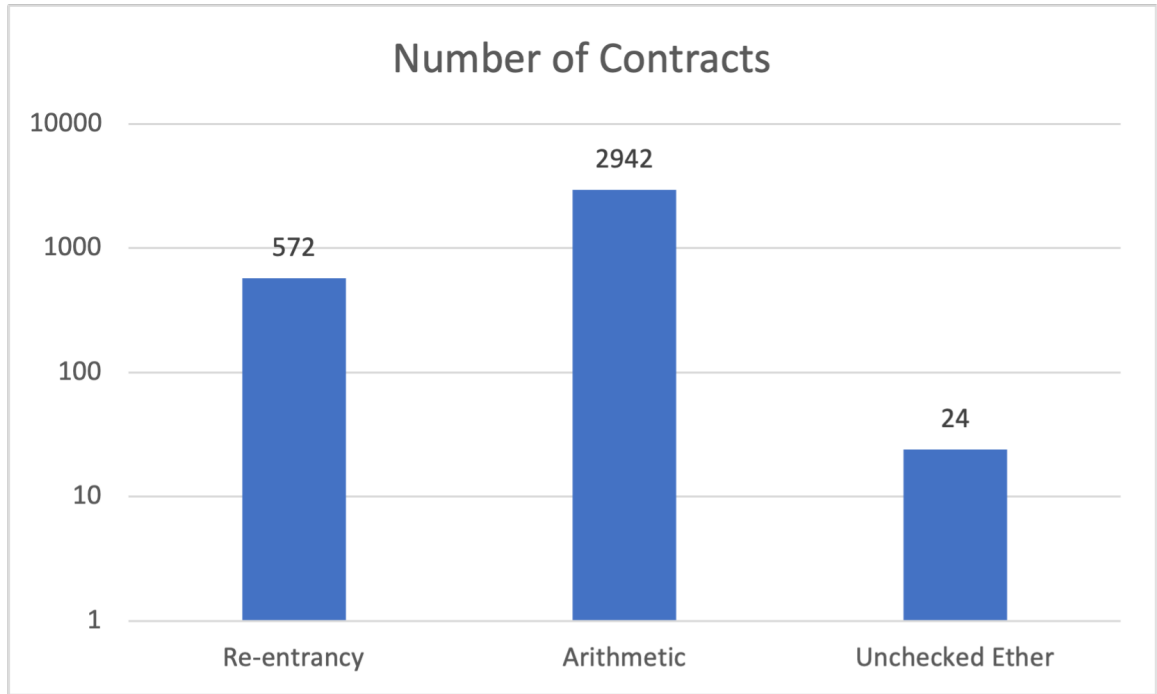


Figure 5: No. of Contracts containing vulnerabilities (log-scale)

2. With regard to each vulnerability, if different tools show different locations, we should not consider those vulnerabilities the same. We consider two warnings -of any degree of importance generated by a tool- as the same only if that vulnerability's name / ID and LOC location match for all of the different tools being benchmarked.
3. The current methodology being used determines the presence of a vulnerability on a line of code if more than 50% of the tools (2 out of 3 in this experiment scenario) confirm the presence of that vulnerability at that exact location / line of code.

Figure 5 showcases the number of smart contract files which contain at least one potential vulnerability. For instance, consider a specific targeted vulnerability. The literature tells us that all of the three static analyzers in the benchmark support the detection of this vulnerability. [51] notes that they only say the contract at hand contains that vulnerability only if an agreed upon threshold of the tools report that it is present at the same location, based on the majority voting system proposed by them. We take on the same system as

well for determining whether a vulnerability is present or not.

4.4 Concluding Remarks

This chapter introduces an up-to-date database, centred around Ethereum smart contracts, namely ETHERBASE, which includes data on the Ethereum blockchain (blocks), its smart contracts, and their metadata. Moreover, aggregate statistics and dataset exploration is presented. Furthermore, future research directions and opportunities are outlined:

During the time we were building EtherBase, we utilized Web3 APIs without taking advantage of an Ethereum full / archive node. The next version of ETHERBASE we're already working on, will take full use of an Ethereum full node and instrument it in order to add a variety of more data to EtherBase. Collecting data via invoking Web3 API's is very much slower than instrumenting an Ethereum archive node.

In addition, our current method is restricted by the rate limit imposed by the API's. For example, Etherscan has a restriction on the daily frequency of queries to its API (5 per day). [51] states this as a serious issue which we would like to solve as well.

We will also be labelling more smart contracts with more tools as we have more time and compute resources moving forward.

The Ethereum security research community can use ETHERBASE for evaluating correctness and other parameters of their proposed or other toolsets, especially those based on machine learning techniques that need comprehensive datasets for training, validation, and testing phases. ETHERBASE comprises a diverse and comprehensive set of real-world heterogeneous annotated smart contracts.

Every tool which was selected for this evaluation is not a complete / sound one, as [51] notices this as well. There are always many false positive / negative results in an audit report generated by a static analyser. Nevertheless, we utilized the mechanism of majority voting suggested by [51] to determine the presence or lack thereof a vulnerability in a

smart contract. We should, however, be wary of the generated false positive results as too many of them will lead to increasing inaccuracies in the released dataset. Such issues can be overcome by adding more tools to the benchmark process or have some auditors manually review the discovered potential vulnerabilities.

Our competitive advantage in comparison to the work done by [51] is that ETHERBASE gets updated regularly, is more comprehensive with regards to the various versions of smart contracts it contains, and that it leverages offline powerful compilation tools to retrieve more metadata about the collected smart contracts, instead of going through the time consuming process of validating each collected contract with online services separately and through manual development of data collection pipelines.

Chapter 5

Conclusion and Future Work

In this thesis, we presented ETHERBASE, an up-to-date database of Ethereum smart contracts to help researchers and developers use it as a testbed for evaluating and benchmarking various smart contracts security tools and frameworks.

Besides that, we provided an overview of the necessary background for someone with a background in electrical / computer engineering to realize the motivation behind providing such a dataset, with regards to the importance of the practice of mitigating vulnerabilities in smart contracts and the lackings in its ecosystem.

We went through the literature concerning the tools researchers have proposed to facilitate the discovery and mitigation of such vulnerabilities and explained how all of them lack the necessary benchmark and testing dataset for reproducibility for further research.

We proposed a version of SLITHER-SIMIL as the first machine-learning-based tool based on a static analysis tool and how it extended into developing ETHERBASE.

In the final chapter, we propose ETHERBASE: an up-to-date database of Ethereum smart contracts written in Solidity, to further facilitate further benchmarking and reproducibility in smart contract research.

Bibliography

- [1]
- [2] Pragmas, 2021.
- [3] Pragmas, 2021.
- [4] Pragmas, 2021.
- [5] R. S. Boyer, B. Elspas, and K. N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245, 1975.
- [6] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137, 2018.
- [7] Consensys. <https://swcregistry.io>.
- [8] ConsenSys. Mythril, 2022.
- [9] P. Daian. Analysis of the dao exploit., 2022.
- [10] V. Dhillon, D. Metcalf, and M. Hooper. The DAO hacked. In *Blockchain Enabled Applications*, pages 67–78. Springer, 2017.
- [11] T. Durieux, J. a. F. Ferreira, R. Abreu, and P. Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20*, page 530–541, New York, NY, USA, 2020. Association for Computing Machinery.
- [12] T. Durieux, J. a. F. Ferreira, R. Abreu, and P. Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20*, page 530–541, New York, NY, USA, 2020. Association for Computing Machinery.
- [13] W. Ethereum. Ethereum whitepaper. *Ethereum*. URL: <https://ethereum.org> [accessed 2020-07-07], 2014.
- [14] Etherscan. The ethereum blockchain explorer, 2022.

- [15] Y. Fan, S. Shang, and X. Ding. Smart contract vulnerability detection based on dual attention graph convolutional network. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 335–351. Springer, 2021.
- [16] J. Feist, G. Greico, and A. Groce. Slither: A static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB '19*, page 8–15. IEEE Press, 2019.
- [17] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu. Smartbugs: A framework to analyze solidity smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1349–1352, 2020.
- [18] C. Ferreira Torres. From smart to secure contracts: Automated security assessment and improvement of ethereum smart contracts. 2022.
- [19] J. Gottschlich, A. Solar-Lezama, N. Tatbul, M. Carbin, M. Rinard, R. Barzilay, S. Amarasinghe, J. B. Tenenbaum, and T. Mattson. The three pillars of machine programming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 69–80, 2018.
- [20] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186. IEEE, 2019.
- [21] N. Group. <https://www.dasp.co/>.
- [22] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 531–548, 2019.
- [23] B. Hu, Z. Zhang, J. Liu, Y. Liu, J. Yin, R. Lu, and X. Lin. A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. *Patterns*, 2(2):100179, 2021.
- [24] B. Jiang, Y. Liu, and W. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269, 2018.
- [25] B. Johnson. A study on improving static analysis tools: Why are we not using them? *Proceedings - International Conference on Software Engineering*, pages 1607–1609, 06 2012.
- [26] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018.

- [27] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68. IEEE, 2018.
- [28] H. Liu, Z. Yang, Y. Jiang, W. Zhao, and J. Sun. Enabling clone detection for ethereum via smart contract birthmarks. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 105–115. IEEE, 2019.
- [29] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS ’16*, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery.
- [30] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS ’16*, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery.
- [31] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.
- [32] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020.
- [33] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC ’18*, page 653–663, New York, NY, USA, 2018. Association for Computing Machinery.
- [34] T. of Bits: Slither documentation. <https://github.com/crytic/slither/wiki/SlitherIR>, 2018.
- [35] Parity. <https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/>.
- [36] G. A. Pierro, R. Tonelli, and M. Marchesi. An organized repository of ethereum smart contracts’ source codes and metrics. *Future internet*, 12(11):197, 2020.
- [37] S. Pilehchiha. Efficient audits with machine learning and slither-simil, Oct 2020.
- [38] S. Pilehchiha. Efficient audits with machine learning and slither-simil. <https://blog.trailofbits.com/2020/10/23/efficient-audits-with-machine-learning-and-Slither-simil/>, 2020.

- [39] M. Ren, F. Ma, Z. Yin, Y. Fu, H. Li, W. Chang, and Y. Jiang. Making smart contract development more secure and easier. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1360–1370, 2021.
- [40] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai. Empirical evaluation of smart contract testing: What is the best choice? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 566–579, New York, NY, USA, 2021. Association for Computing Machinery.
- [41] S. Srikant. *Vulcan: classifying vulnerabilities in solidity smart contracts using dependency-based deep program representations*. PhD thesis, Massachusetts Institute of Technology, 2020.
- [42] O. Sürücü, U. Yeprem, C. Wilkinson, W. Hilal, S. A. Gadsden, J. Yawney, N. Alsadi, and A. Giuliano. A survey on ethereum smart contract vulnerability detection using machine learning. *Disruptive Technologies in Information Sciences VI*, 12117:110–121, 2022.
- [43] N. Szabo. Formalizing and securing relationships on public networks. *First monday*, 1997.
- [44] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB ’18, page 9–16, New York, NY, USA, 2018. Association for Computing Machinery.
- [45] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’18, page 67–82, New York, NY, USA, 2018. Association for Computing Machinery.
- [46] M. Varun, B. Palanisamy, and S. Sural. Mitigating frontrunning attacks in ethereum. In *Proceedings of the Fourth ACM International Symposium on Blockchain and Secure Critical Infrastructure*, pages 115–124, 2022.
- [47] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, 8(2):1133–1144, 2020.
- [48] M. Wohrer and U. Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8. IEEE, 2018.
- [49] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

- [50] C. Xing, Z. Chen, L. Chen, X. Guo, Z. Zheng, and J. Li. A new scheme of vulnerability analysis in smart contract with machine learning. *Wireless Networks*, pages 1–10, 2020.
- [51] C. S. Yashavant, S. Kumar, and A. Karkare. Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities. *arXiv preprint arXiv:2202.11409*, 2022.
- [52] L. Zhang, Y. Li, T. Jin, W. Wang, Z. Jin, C. Zhao, Z. Cai, and H. Chen. Spcbig-ec: A robust serial hybrid model for smart contract vulnerability detection. *Sensors*, 22(12):4621, 2022.
- [53] L. Zhang, J. Wang, W. Wang, Z. Jin, C. Zhao, Z. Cai, and H. Chen. A novel smart contract vulnerability detection method based on information graph and ensemble learning. *Sensors*, 22(9):3581, 2022.