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 (Electrical and Computer Engineering) at
Concordia University
Montréal, Québec, Canada

August 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 (Electrical and Computer Engineering)

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

Signed by the final examining committee:

_____ Chair
Dr. Rodolfo W. L. Coutinho

_____ Examiner
Dr. Rodolfo W. L. Coutinho

_____ Examiner
Dr. Carol Fung

_____ Thesis Supervisor
Dr. Amir G. Aghdam

_____ Thesis Supervisor
Dr. Jeremy Clark

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

August 29, 2022 _____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Improving Reproducibility in Smart Contract Research

Sina Pilehchiha

The most popular smart contract-based blockchain platform at the moment is Ethereum. Based on market value, it is the second-largest blockchain platform behind Bitcoin, with a steadily increasing market share. Ethereum smart contracts are used to secure billions of dollars worth of assets. Source code for smart contracts must be examined for any potential flaws that could result in significant financial losses and damage trust because they cannot be modified after deployment. A wide range of tools have been developed for this goal, and extensive literature on vulnerabilities and detection techniques on the subject above constantly keeps emerging. The analysis, testing, and debugging of smart contracts through automated processes have also been the subject of extensive research. Researchers have worked on the development of tools that can automatically detect and fix vulnerabilities in smart contracts, especially tools that rely on less explored methodologies, such as machine learning-based tools. We provide details on our work on SLITHER-SIMIL, a statistical addition to a static analyzer, as a data-driven endeavor to complement the existing security analysis methods of smart contracts. SLITHER-SIMIL allows developers and auditors to check the similarity between the source code snippets of smart contracts written in Solidity and allows users to check smart contracts with a database of vulnerable smart contracts through the same mechanism of similarity checking in order to facilitate the discovery of security vulnerabilities in smart contracts. However, such automated analysis tools typically need datasets for their training, testing, and validation phases; collecting such data for smart contracts is time-consuming. Besides, 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 who have researched the above topics. Research studies offer datasets that frequently come in the form of sparse datasets with minimal to no usage guidance. Due to the fast-paced nature of the Ethereum ecosystem, the datasets available are often quickly outdated. These are significant barriers to performing verifiable, reproducible research, as it takes a substantial amount of time to accomplish many subtasks such as locating, extracting, cleaning, and categorizing a reasonable amount of high-quality, heterogeneous smart contract data. To address this issue, we introduce ETHERBASE, an extensible, queryable, and user-friendly database of smart contracts and their metrics that improve reproducibility and benchmarking in smart contract research.

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 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

List of Tables

Chapter 1

Introduction

This chapter introduces several research contributions in this thesis and motivates their importance.

1.1 Motivation

The research community has made much effort to develop automated analysis tools to be able to identify vulnerabilities in smart contracts [24]. 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 [24]. 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 [37].

Famous attacks that have caused significant financial losses and prompted the research community to work to prevent similar occurrences include The DAO hack [17] and the Parity wallet issue [45]. Comparing and reproducing such research is not an effortless process. The datasets used to test and benchmark the tools proposed in the research literature are not easy to access and analyze, 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 [37].

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 threaten the validity of the research [50].

We faced the same issues while working on SLITHER-SIMIL, a machine learning-based addition to the static analysis tool Slither [23] for similarity checking of smart contract source code snippets. While trying to extend SLITHER-SIMIL with supervised learning algorithms to expand its use cases (such as vulnerability prediction in source code), we faced a data scarcity problem for training the models, which led us to our work on ETHERBASE.

1.2 Contributions

In this thesis, we first present our work on SLITHER-SIMIL, a machine learning-based similarity checking tool for smart contracts that can be useful for the discovery of security vulnerabilities in smart contracts through matching the source code of a smart contract with a vulnerability pattern. Afterward, we introduce ETHERBASE, an 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 will eventually be made public, and only the collected data is accessible to the public now. 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 present our work on SLITHER-SIMIL, a lightweight similarity checking tool for smart contracts written in Solidity.

2. We propose ETHERBASE, an up-to-date database for Ethereum by exploiting its internal mechanisms.

3. We implement ETHERBASE. 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 renewed compared to the previous manual one-time data gathering efforts.

4. 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 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 work on the tool SLITHER-SIMIL, an automated analysis tool for similarity checking of smart contracts.

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 discuss our conclusions and reveal plans for future work and research directions.

Chapter 2

Background

This chapter prepares the reader by reviewing the necessary background knowledge for a better acquaintance with the contributions presented in this thesis. It briefly introduces the 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 [25]): First, we go over the basics of Ethereum, its components, and its 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. Afterward, we will go over smart contracts and their most common vulnerabilities out in the wild. We will discuss smart contracts written in Solidity and explain each vulnerability according to the DASP Top 10 classification [27].

2.1 Ethereum

Ethereum is a decentralized virtual machine introduced as alternative blockchain technology to Bitcoin in 2014 by [58]. A blockchain is a peer-to-peer network made up of computers/nodes that update for one single global database -serving as a ledger- without necessarily trusting another in a distributed fashion; the resulting ledger records every transaction

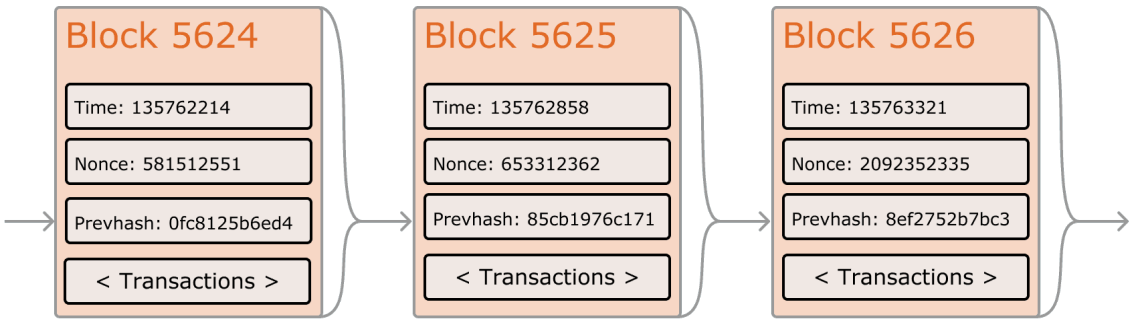


Figure 1: A visualization of Ethereum block structure [21].

that each node in the blockchain network makes [25]. It is based on a combination of cryptography, networking, and incentive mechanisms [57].

As the second most popular blockchain, the Ethereum blockchain is a transaction-based, cryptographically secure state machine [25]. It is also the most commonly used blockchain platform utilized to develop smart contracts. It takes a series of inputs transitions from its initial state to a new one according to the transition rules defined by those inputs [25]. Like Bitcoin, Ethereum currently depends on the Proof-of-Work (PoW) consensus protocol. Proof-of-Stake (PoS) [32] and PBFT (Practical Byzantine Fault Tolerance) [13] are other forms of a consensus protocol used by other blockchains per their defining characteristics. The PoW mechanism works as follows: a series of cryptographic puzzles which are computationally hard to solve are introduced to the existing nodes on the network, and the solutions to those puzzles are utilized as proof that the data being written on the blockchain are legitimate and credible. This gets verified through the consensus protocol. When a node creates a block, it must resolve a PoW puzzle and spend computing power to achieve so [35]. Trying to solve the puzzles sooner than any other party, the nodes compete with each other over this objective function and the node with the most computing power usually succeeds [25]. After a node proves its solution to a puzzle is correct, it will be relayed to every other node on the network in order for all of them can reach a consensus and agree upon appending a new block to the blockchain [35], [25]. This proves that

some node has done specific work to solve the puzzle by leveraging their computational resources. This whole process is called mining, and all the nodes participating in the process of competing with each other to create new blocks and append them to the blockchain are known as miners [25].

2.1.1 Ethereum Virtual Machine

The formal definition of the EVM is specified in the Ethereum Yellow Paper [58]. 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 that EVM can read and compute and execute the logic of that smart contract. The EVM is also responsible for estimating and calculating gas consumption for transactions in smart contracts. In short, EVM represents the state of the entire universe of Ethereum (like a global decentralized computer); every account and its balance, the code for every smart contract hosted on the Ethereum, the current state of every variable in every smart contract, and anything else Ethereum is contained within the EVM.

2.1.2 Accounts

Ethereum, at its core, is composed of many entities named accounts [25]. Each account is identified with a 20-byte address to interact with the other accounts on-chain. An address on the Ethereum blockchain is a 160-bit identifier used to identify any account [58]. Ethereum supports two types of accounts:

- externally owned (controlled by private keys), known as EOAs or "simple accounts" that are usually owned by users or devices who control these accounts, and

- contract accounts (CAs, controlled by their contract code contained in them) [21].

Both of these types of accounts are equipped with the capabilities to hold, send, and receive ETH and tokens and to interact with deployed smart contracts [21].

The key differences between these two types of accounts are as follows; Creating an EOA account costs nothing; they can initiate transactions by invoking functions on a smart contract that updates the balances of their tokens. The transaction sent/received between two accounts of the EOA type can only be transfers of ETH and/or tokens. CAs, on the other hand, have a cost associated with creating new accounts due to the fact that the creator is using up an amount of network storage; CA accounts are not capable of initiating transactions, unlike EOAs. With regards to the nature of the transactions they can send/receive, when a CA account receives a transaction from an EOA account or a CA account, that transaction can trigger a code that can execute many different actions, such as transferring tokens or the creation of another account [1].

EOA accounts can send transactions to other EOA accounts, and CA accounts can send transactions via the creation of a transaction and then signing it with a private key. A private key, alongside a public key (derived from its private key pair), constitutes what is known as an asymmetric or public key cryptography scheme [28]. This scheme leverages unique keys based on mathematical functions to secure information. From a computational standpoint, They are fairly easy to calculate, but it is very hard to compute the inverse of such functions, and the Ethereum blockchain leverages this scheme in order to create public-private key pairs. An account in Ethereum is represented then by such pair of keys. The private key controls access by being the unique piece of information needed to create sign transactions in order to authorize the spending of any funds in the account, and the address of the account is derived from the public key [6]. By signing a transaction, we mean generating a unique hash of the data being sent and encrypting it using the sender's private key.

Inside of an Ethereum account is composed of four fields: nonce, ether balance, contract codeHash, and storageRoot, explained as follows:

- **Nonce:** Nonce is the number of transactions sent from a given address. Nonce is used to make sure that every transaction is processed once and only once. This is used to prevent replay attacks in Ethereum. Nonce counts the number of transactions initiated by the account to prevent replay attacks.

- **Balance:** (Ether) Balance records the Ethereum balance of the account, which is the amount of Wei assigned to the address of that account. Wei is the smallest measurement unit of Ether (1 Wei is the equivalent amount of 10^{-18} ethers).

- **storageRoot:** Each Ethereum account has its data stored in a trie data structure, and StorageRoot is a hash of the root node of that data structure.

- **Contract codeHash:** The codeHash of a contract is the Keccak-256 hash value of the code of the account on the EVM. A hash value is the output of a hash function, a one-way function mapping its input to a fixed-size string of values [6]. Keccak-256 is such a function. By referring to this function as a one-way function, we mean that it is computationally hard to recreate the input data if we only know the output of the function [6]. The only way to determine a possible input to such function is to conduct a brute-force search, checking each candidate for a matching output, and given that the potential search space is very large, it makes the task impossible [6].

2.1.3 Transactions

A transaction is a cryptographically signed instruction sent by an account on the network towards another [25]. The runtime resources needed for transactions to get executed in Ethereum, the costs associated with conducting transactions are referred to as gas, and it is expressed in Gwei (Giga Wei or 10^{-9}). The justification for the implementation of the gas

system is for it to discourage malicious actors or accidental bugs/coding mistakes in the form of infinite loops. There exist only two types of transactions based on the outcomes they generate [25]:

- 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 transactions is sending Ether from one account to another.
- Contract creation call, which creates new accounts with a code associated with it.

2.1.4 Blocks

Blocks are a series ("chain") of batches of transactions with each block, including a cryptographic hash of its previous block in the chain they are included in [25]. This way, each block is linked to other blocks as the above-mentioned hashes are cryptographically derived from the block data. This makes the series of blocks ("blockchain") immutable and makes fraudulent conduct much harder [25]. This is because a minor mutation or change in the history of a transaction invalidates every block and its contents after the change, as all the cryptographic hashes regarding the blocks will need to change as well to reflect that minor change.

A block in Ethereum contains many sub-components of data, such as the block header and a list of transactions [25], explained as follows [2]:

- Timestamp: The stored data verifying the time when a miner mined the block.
- Block Number: The counter number of the current block, which is also the length of the blockchain in blocks up to this block.
- Base Fee Per Gas: The minimum fee per gas required for a transaction to be included in the block.

- Difficulty: The effort required to mine a block.
- MixHash: A unique identifier in the form of a cryptographic hash value for every block.
- ParentHash: The unique identifier for the block that came before the current block, expressed as a hash value.
- Transactions: The list of transactions included in the block.
- StateRoot: This is the hash value representing the root node of the trie data structure containing the state of the system, including account balances, contract code, contract storage, and account nonces.
- Nonce: a hash that, combined with the value for MixHash of a block, provides proof that this block has gone through the PoW (Proof-of-Work) process.

2.2 Smart Contracts

Smart contracts are fundamentally more (decentralized) *applications* in the form of executable code than *contracts*, containing code that can be invoked, capable of holding and sending value, and executable by a decentralized network. They provide a framework that allows any sound program to be executed in an autonomous, distributed, and trusted manner [43]. 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 Solidity

Ethereum Smart contracts are typically developed in high-level programming languages to facilitate and expedite the development process and then compiled to lower-level bytecode

to run on the EVM. Solidity is one of these languages, a statically-typed object-oriented high-level programming language for writing smart contracts [3], with a similar syntax to the Java programming language and influenced by C++ and Python as well. It is not as mature as other programming languages heavily used in the industry. Its syntax and semantics are on an ever-changing trajectory as the developers try to introduce/remove features to make it safer or facilitate the development of smart contracts due to the critical nature of the applications smart contracts typically deal with. As stated in the introductory remarks, vulnerabilities and specifically security vulnerabilities in Solidity bear huge financial risks for the users and developers of the applications utilizing smart contracts. There has been much attention focused on discovering, mitigating, and documenting these vulnerabilities, and the appropriate countermeasures for them, such as [27], [15], and [8]. In the following, we review some of the most famous security vulnerabilities discovered in smart contracts throughout the history of Ethereum.

2.2.2 Vulnerabilities

Smart contracts, like any other software in history, are prone to all kinds of vulnerabilities. What makes security vulnerabilities in languages used to write smart contracts -such as Solidity and Vyper- so attractive is that the programs written in Solidity are very 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. Unlike attacking scenarios in traditional finance, if a malicious party discovers a vulnerability, it can be very effortless to pull ETH/tokens out of the Ethereum DApps and their smart contracts quickly, in an anonymous fashion, without the oversight of banks and other financial intermediaries. Some of these vulnerabilities, like those of other programming languages, arise from the human factor involved in the development of smart contracts, and some are specific to the blockchain data structures and how they and their components function and

interact with each other. Furthermore, these are only vulnerabilities within the scope of smart contracts we focus on. Vulnerabilities can arise regarding the blockchains' core infrastructure handling smart contracts. In this section, we go over 9 of the more discussed vulnerabilities in Ethereum smart contracts according to [27] to get a better sense of what threat surface the developers and researchers developing analysis tools face:

Reentrancy. Often called the most famous Ethereum vulnerability, the reentrancy attack has been a great example of showing the risks of unsafe development of smart contracts and the importance of smart contract security historically. The DAO hack [18] is one of the most famous real-world examples of the reentrancy hack. The reentrancy attack can also be counted as a 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 ETHs. The reentrancy vulnerability is exploited when external contract calls are allowed to make new calls to the calling contract before the initial execution of that call is complete [27].

Access Control. The Access Control vulnerability, not exclusive to smart contract types of programs, usually occurs when smart contracts use poor visibility settings regarding 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 [46]. Values assigned with the integer data type, if not handled carefully concerning 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 because some low-level calls in Solidity like `call()` and `delegatecall()` can return a boolean value set to the value False and the execution flow can nevertheless resume if an error of this kind happens mid-execution. This is not ideal since it means that the execution of the smart contract has not been reversed and completed but with wrong or undesirable outcomes. Thus, the return values low-level calls like those mentioned above generate should always be investigated, and the developers must ensure 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). Exploiting this vulnerability happens when malicious miners alter the initial default ordering of the transactions submitted to the blockchain. Per Eskandari et al. [20], 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 the victim's due to it giving having more aligned incentives with he miners' network.

- Insertion attack, in which an adversarial actor makes two transactions, one with a higher gas price than that of the victim and one with a lower gas price, to *sandwich* the victim transaction [55].

- 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*, [27] 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 a malicious party can guess, then the attacker can predict the next number that will be generated. Values such as block timestamps or block numbers are generally advised against being 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 [15].

Time Manipulation. This vulnerability is also known as *timestamp dependence* [27]. In Solidity, a block's timestamp is often used to generate pseudo-random numbers. 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 for their 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 exploiting this vulnerability can craft truncated addresses that clients may encode incorrectly in transactions. Additionally, it has not been exploited in the wild, as mentioned by [24].

Chapter 3

SLITHER-SIMIL: Automated Vulnerability Analysis of Smart Contracts on Ethereum

3.1 Introductory Remarks

Smart contracts, the universal and vital programs deployed on blockchains, have gotten more and more attention with the rapid development of blockchain technology. A smart contract is an event-driven, self-executing, state-based program written using high-level programming languages like Vyper and Solidity. Because of their distinctive features, smart contracts require more careful development than the amount usually needed for traditional software programs. First, smart contracts are more prone to bugs and vulnerabilities compared to regular software. The concept that smart contracts execute themselves and cannot be stopped after deployment has been previously coined by an expression described by Lessig [34] as "code is law." Transactions of a smart contract typically involve digital assets (e.g., various cryptocurrencies or NFTs) of financial nature, which can be very

profitable if stolen (e.g., The DAO [17]). Therefore, a bug in a smart contract may lead to substantial financial losses, and ensuring the correctness of smart contracts before their deployment is critical. This requires one to reuse the experience of developed contracts in the past when developing new contracts.

Program analysis for smart contracts can greatly facilitate their development and maintenance. The conventional 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. As a result, expert adversaries can easily exploit these manual checking patterns. To minimize the risk of potential attacks, machine learning-based systems provide more secure solutions relative to hard-coded static checking tools.

Surucu et al. [52] provide the first-ever survey on machine learning methods utilized to discover and mitigate vulnerabilities in smart contracts. In order to set the ground for further development of machine learning-based methods for smart contract vulnerability detection, they reviewed many machine learning-based detection mechanisms. Based on their survey paper, we first review the existing analysis tools and then review some of the more novel machine learning-based methodologies proposed in the literature over the past few years in chronological order. We also add some of the works missing in [52]. Afterward, we propose our solution, SLITHER-SIMIL and how it led us to the development of ETHERBASE.

3.2 Traditional Security Analysis Methods in Smart Contracts

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. [50] provides three categories of tools based on their utilized methodology, namely static analysis, dynamic

fuzzing, and symbolic execution.

Using the static analysis method, we can analyze the program at both the source code (high-level) and bytecode (low-level) scopes before conducting any runtime execution. Static analysis-based tools can scan a whole codebase, 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:

1. building an intermediate representation (IR), such as an abstract syntax tree (AST) for analyzing the structure of the source code beside raw text/source code;

2. complementing the generated IR with additional metadata with methods such as control flow, data flow analysis, and symbolic execution.

3. vulnerability detection through pattern matching and referencing those patterns to a database containing (vulnerable) patterns. Setting a specific threshold defines whether a matched pattern counts as a vulnerability.

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 predefined database of vulnerability patterns and filter out the suspicious snippets of the input program. Slither [23], Securify [54], and SmartCheck [54] are categorized as instances of static analyzers.

Fuzzing [14] is a technique for finding software bugs that involve 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 tests a system with the continuous processing of test cases generated by another program. A fuzzer will initially generate a set of seeds to feed into the program and test it with. After getting initial results with regard to the initial inputs (a.k.a. seeds), the fuzzer will adjust the input space to explore as many states the program can get into as possible. ContractFuzzer [30], ReGuard [36], and

sFuzz [43] are among the most cited smart contract fuzzers.

Symbolic execution is a technique for finding software bugs that involve creating symbolic values and watching the target program's unusual output while it runs. This method utilizes symbolic values as input instead of specific/real values for variables and other parameters during the execution. Tools leveraging this technique explore a state space with a high degree of semantic awareness [12]. Given different initial seeds as input to the program, the logic of the program can transition through various states, and symbolic execution tools generally explore these multiple paths and check their outputs. This also causes issues unique to this approach, such as path explosion [9]. The symbolic execution tools usually build a control flow graph based on the Solidity bytecode of the tested smart contracts. Oyente [38], Mythril [16], and Manticore [42] 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. Afterward, we will review the tool we worked on, SLITHER-SIMIL.

There have been many efforts focused on 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 bytecode representations.

In 2019, Momeni et al. [41] proposed one of the first models to detect vulnerabilities in smart contracts based on classical machine learning methodologies. They utilized methods of static code analysis as the underlying technology and trained a series of ML-based models for various vulnerabilities. Their model of machine learning-based security vulnerability analysis took advantage of using multiple static analyzers, as mentioned above,

for the pre-training and labeling phases in order to compare the performance of their proposed ML models with those of the static analyzers. According to them, the proposed successfully discovered 16 unique vulnerabilities with an average accuracy of 95%. They state that their intention for developing such a model was to focus on reducing the time resources developers and auditors typically put into discovering and mitigating vulnerabilities in Ethereum smart contracts. The static analyzers they tested in their experiments were namely Slither [23] and Mythril Classic [16].

In 2020, Gao et al. [26] a new vulnerability detection method based on studies concerning the modeling of the rich structure of the source code as embedding vectors and discovery of vulnerabilities through checking for similarities between two embeddings. They developed this model for various tasks such as detecting clones and bugs and validating source code concerning Ethereum smart contracts. They assessed their proposed approach on more than 22,000 Solidity smart contracts in Ethereum. For the task of clone detection, their proposed tool (named SmartEmbed) can identify many instances of repetitive snippets of code written in Solidity where the clone ratio is around 90 percent, and the performance improvement compared to the classic semantic clone detection tool Deckard [31] has been noticeable. Concerning the main task of bug detection, their tool was able to identify more than 1,000 bugs based on their bug databases; This is what made SMartEmbed one of the first-ever tools to help developers and auditors to have a more efficient way of checking smart contracts for clone detection and bug detection.

In 2020, Xing et al. [59] 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) [29]. These 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. The authors mention that the slice matrix feature needs further exploring.

Ashizawa et al. [7] proposed Eth2Vec model in 2021, an ML-based static analyzer for the purpose of detecting vulnerabilities. Their emphasis in designing these tools has been implementing an automated process for extracting features from each data point (here, a smart contract) by borrowing neural network-based models from the much more mature field of natural language processing (NLP). A neural network structure is utilized as the final step to find any vulnerabilities in the source code. Five hundred contracts were used to test the suggested model, and even though the contracts were changed, the Eth2Vec model could identify vulnerabilities with a 77 percent accuracy.

In their paper published in 2021, Y. Xu et al. [60] note the use of traditional vulnerability discovery methods based on static/dynamic analysis. In their publication, they utilize two methods for vulnerability identification, namely the Stochastic Gradient Descent (SGD) and the K Nearest Neighbors (KNN) models. First, they convert source code into corresponding AST structures and extract feature vectors from the AST data structures. Then, they leverage those feature vectors to train their vulnerability detection models. Its performance has been reported as greater than 90% more for the accuracy, recall, and precision concerning some of the vulnerabilities detectable through the traditional methods, such as arithmetic, reentrancy, and access control. They report a better performance regarding their model compared to tools such as Oyente [38] and Smartcheck [53].

In 2021, by using bigram properties from the streamlined operation codes of smart contracts, Wang et al. [56] introduced their 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 reentrancy Each smart contract's source code is converted to opcodes; A smart contract typically has 100 different 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 neighbors, 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 [38] system. Due to the scarcity of particular vulnerabilities, the researchers ran into a class-imbalance problem after the labeling process. Extreme Gradient Boosting (XGBoost), Adaptive Boosting (AdaBoost), Random Forest (RF), Support Vector Machine (SVM), and k-Nearest Neighbour were the five candidate ML models used in the training procedure (KNN). By reaching above 96 percent F1, Micro-F1, and Macro-F1score, the XGBoost model demonstrated strong performance.

In 2022, Yuqi Fan et al. note that most of the studies on vulnerability detection methods regarding smart contracts rely on predefined manual rules from experts and auditing professionals [22]. 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 they fail to represent the source code/bytecode well semantically and structurally. Then, they propose a novel Dual Attention Graph Convolutional Network (DA-GCN) model 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. Afterward, 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. [63] introduce a new detection model: contract-level vulnerability prediction based on ensemble learning. It is based on seven different neural networks using vulnerability data for vulnerability detection at the scope of smart contracts (single file-level scope). Zhang et al. pre-trained seven different neural networks with an information graph (IG) consisting of the datasets of source code data. Then, they explain how they integrated each model into an ensemble model named Smart Contract Vulnerability Detection method based on Information Graph and Ensemble Learning (SCVDIE) methodologies. They tested the effectiveness of their proposal by leveraging a test dataset and benchmarked the results of the test against static analysis tools, apart from a few other data-driven methods.

Also, in 2022, Zhang et al. [62] proposed another novel method for vulnerability detection. They worked on a hybrid model named Serial-Parallel Convolutional Bidirectional Gated Recurrent Network Model. It includes an ensemble classifier which enhances the robustness of the model. They mainly focused their experimentations on six types of vulnerabilities and generated two datasets for multi-task detection purposes. They explain in the results section of their published work that their proposal has a higher performance than most of the other referenced methods as it has achieved the following F1-scores scores for three out of the six mentioned vulnerabilities: 96.74%, 91.62%, and 95.00%.

3.4 SLITHER-SIMIL

Research on automatic vulnerability discovery in Solidity has taken off in the past two years, and tools like Vulcan [51] and SmartEmbed [26], which use machine learning-based approaches towards discovering vulnerabilities in smart contracts are gradually showing promising results, with fewer false positives in reports on specific vulnerabilities.

However, all the current related work focuses on vulnerabilities already detectable by static analyzers like Slither [23] and Mythril [16], and less emphasis is put on the vulnerabilities such tools are not able to identify and human auditors need to discover.

The efforts of software security and auditing firm Trail of Bits, Inc. concerning automating smart contract security assessments have included works on an already prominent static analysis tool, Slither [23], to better help developers and researchers in their process of auditing smart contracts.

Trail of Bits has manually curated a wealth of data—years of security assessment reports —and during the author's internship at Trail of Bits- we decided to explore how to use this data to make the smart contract auditing process more efficient. Unlike the related work mentioned above and based on accumulated knowledge embedded in previous audits, we set out to detect similarly vulnerable code snippets in our clients' new codebases by focusing on the security vulnerabilities that were not detected by static analyzers or fuzzers but through human auditors' manual efforts. Specifically, we wanted to explore machine learning-based approaches for potential data-driven ways to improve Slither's performance and facilitate 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 [48], the statistical addition to Slither, is a code similarity measurement tool for detecting similar snippets of Solidity source code. When it began as an experiment last year under the codename crytic-pred, it was used to vectorize Solidity source code snippets and measure their similarity. This past year, We updated the tool and applied it directly to a dataset of vulnerable source code patterns written in Solidity. It is written entirely in 1,100 lines of Python and accepts any number of smart contracts written in Solidity in the form of .sol files.

SLITHER-SIMIL utilizes FastText [11], a library for efficient learning of word representations, to generate compact numerical representations of every Solidity function. Among the many libraries for generating word vectors, FastText was chosen as the complementing because it is well maintained by Facebook Research and enjoys a high degree of performance because it is written in C++.

We worked on the development of SLITHER-SIMIL and SlithIR for four months, focusing on data collection and development, 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 a baseline text-based model with FastText to be compared in the near future 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.

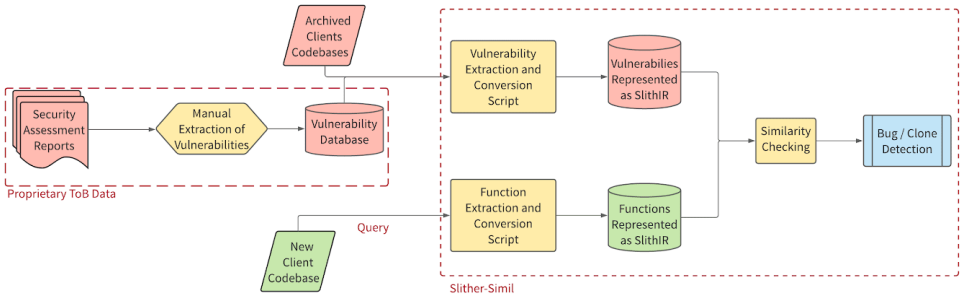


Figure 2: A high-level view of the process workflow of SLITHER-SIMIL.

In the next section, we will go over how SLITHER-SIMIL works and performs similarity checking between an input function snippet and a vulnerability database, composing of manually extracted vulnerable Solidity functions.

3.4.1 Methodology

The process workflow of SLITHER-SIMIL is illustrated in Figure 2.

First, we manually collected and extracted 89 vulnerabilities from the source code of Solidity smart contracts in projects from 10 security assessment reports conducted by Trail of Bits. Each assessment report states details on vulnerabilities discovered by human auditors and details the scope of the function or contracts they are located in. We were interested mainly in function-specific vulnerabilities, as opposed to vulnerabilities arising due to the interaction of different smart contracts, as we were trying to model Solidity source code as function-level snippets in the form of vectorized embeddings. Each extracted vulnerability had some associated information, such as the name of the function and contract it was relevant to, the line of code it was specified at, the type of the vulnerability, and a unique ID based on the name of the project it belonged to.

After that, we used Slither's parser to compile the previous clients' codebases and matched the functions they contained with our vulnerability database through an automated function extraction and normalization pipeline.

SLITHER-SIMIL currently uses its representation of Solidity code, as introduced by [23], namely SlithIR. SlithIR (Slither Intermediate Representation) is heavily used to parse and encode source code snippets written in Solidity at the granularity level of functions. We thought the function-level analysis was a good place to start our research since it is not too coarse (like the file level) and not too detailed (like the statement or line level). As introduced by Feist et al. [23], SlithIR was developed as an intermediate representation (IR) language with slither in mind to leverage it and represent Solidity code for further analysis. Solidity smart contracts can decompose into a control flow graph. Each node can contain up to a single Solidity expression in that graph, which is converted to a set of SlithIR instructions. This representation makes implementing analyses easier without losing the critical semantic information in the Solidity source code [23]. 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 are available at [44].

Here we exemplify a function from a smart contract to showcase how we used Slither to transform a Solidity function into its corresponding intermediate representation SlithIR. (Listing 3.1) First, we converted every statement or expression of the function into its corresponding SlithIR representation and then tokenized the SlithIR sub-expressions and further normalized them. (Listing 3.2)

Normalization generally helps the final performance of similarity matching because it removes the superficial and irrelevant differences -semantically speaking- between the tokens of two function snippets, such as names of variables and constants, identifiers, and numbers. (Listing 3.3)

```
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 obtaining the final form of normalized token representations for a function, we compare its structure to the vulnerable functions in our vulnerability database.

Then, SLITHER-SIMIL embeds the normalized tokens into real-value vectors via techniques adapted from word embeddings literature. FastText utilized two algorithms for learning word/token representations: Skipgram [40] and CBOW (Continuous Bag Of Words) [39]. The Skipgram model learns to predict a target word according to the context words surrounding the target word. The CBOW model, in contrast, predicts the target word according to its context. The surrounding context word(s) is represented as a bag of the words contained in a fixed-size window around the target word [5]. We preferred the use of the Skipgram model (although the user of SLITHER-SIMIL can opt to use the CBOW model by specifying an input parameter or any other model by changing a few lines of code) because it is generally known to have a better performance than CBOW on small datasets [10].

The generated normalized token sequences are then fed into the Skipgram model implemented by FastText to train the model based on the source code of the corpus containing all of the normalized tokens. After the training phase is done, each normalized SlithIR token in the training dataset gets mapped to a vectorial representation containing real values, representing the embeddings of their n-grams [11]. In order to generate embeddings for larger scopes of Solidity source code (e.g., expressions, whole functions, contracts, and files), the SlithIR token-level embeddings can be summed together to make function-level

```
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.

```
type_conversion(uint256)

binary(**)

binary(*)

(state_solc_variable(uint256)):=(temporary_variable(uint256))

index(uint256)

(reference(uint256)):=(state_solc_variable(uint256))

(state_solc_variable(string)):=(local_solc_variable(memory, string))

(state_solc_variable(string)):=(local_solc_variable(memory, string))

...

```

Listing 3.3: Normalized SlithIR tokens of the previous expressions.

```
$ slither-simil test etherscan_verified_contracts.bin --filename TurtleToken.sol --fname TurtleToken.transferFrom --input cache.npz --ntop 5

Output:
Reviewed 825062 functions, listing the five most similar ones:

filename contract function score
...
TokenERC20.sol TokenERC20 freeze 0.991
...
ETQuality.sol StandardToken transferFrom 0.936
...
NHST.sol NHST approve 0.889

```

Listing 3.4: Normalized SlithIR tokens of the previous expressions.

embeddings.

At the similarity checking stage, and after obtaining the final form of token representations for an input function, we compare its structure to that of the vulnerable functions in our vulnerability database, based on an input threshold percentage of similarity given by the user. For this process to happen, we need to define a metric for measuring the similarity between two function snippets. We used the cosine similarity metric for measurement, measuring the similarity between two vectors using the cosine of the angle between them in an embedding space, defined as:

$$\text{Similarity}(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\| \, \|B\|} \tag{1}$$

where $A, B \in \mathbb{R}^n$. A and B are two real-valued vectors selected for measuring their (dis)similarity, and $\|A\|$ is the Euclidean norm of vector A, which is defined as follows:

$$\|A\| = \sqrt{\sum_{k=1}^{n} |A_k|^2} \tag{2}$$

where A_k is the kth element of A. As the similarity score gets lower and lower and tends towards 0, these 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.

3.4.2 Evaluation

For evaluation, we manually extracted vulnerabilities from previously archived audit reports and transferred them to a vulnerability database. Afterward, we tried to compare the similarity of some of the new codebases set for auditing with the vulnerable function snippets in our vulnerability database, composed of 89 vulnerabilities extracted from 10

security assessment reports. (the vulnerability database cannot be shared publicly due to IP limitations). The results of this process were not meaningfully helpful. Comparing function snippets from our test codebases to any vulnerable function snippet from the vulnerability database generates a very high similarity percentage. This is, in part, due to the high clone ratio in Ethereum smart contracts. This is also noted by other work such as [33] and [26], reporting clone ratios of 79.2% and 90% among the smart contracts they studied, respectively. Besides, the high clone ratio and the lack of enough data made it hard for us to find any significant vulnerabilities in our similarity checking attempts between the new codebases and the vulnerability database. This is because although the function snippets were vectorized in embedding space, their similarities in the text were not significant enough to show after the embedding process. SLITHER-SIMIL proved useful for another use case we did not plan for, and that was when an auditor was suspicious of the safety of a function snippet. They could leverage SLITHER-SIMIL to quickly generate a vectorized embedding of the suspicious function snippet and check the similarity percentage to the other function snippets in the vulnerability database. They would check the function again more carefully if it were above the stated threshold (usually 95%). 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 is a more facilitated process, but it does not do much good for us in the case of Solidity, where even the language itself is very young, and we need to be careful of how we treat the amount of data we have at our disposal.

SLITHER-SIMIL is now available on Github and ready to use. For end users, it is available as a CLI tool; 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 research community's efforts to propose machine-learning-based methods to discover and mitigate vulnerabilities in smart contracts compared to the existing approaches used in the industry. We also went over SLITHER-SIMIL, a powerful tool with the potential to measure the similarity between function snippets of any size written in Solidity. Its inputs are batches of smart contract files written in Solidity, and its output is vectorized embeddings for function snippets in each smart contract file. Nevertheless, there is a lacking here, and that is the bottleneck of datasets that help us train and test bigger and more comprehensive models. It took the author about two months to extract and collect the 89 vulnerabilities from the ten projects as mentioned above since we needed to handle the different solc versions to compile each project separately, retrieve the lost subtle knowledge required to run each project, each with their own set of unique dependencies. This shows that data collection and cleaning is a bottleneck in further reproducible research on smart contracts. 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 the broader research community and us. For future work concerning SLITHER-SIMIL, we plan to utilize ETHERBASE and other data sources to train and test unsupervised models that are more general and more accurate than the ones we have developed and also to train and test supervised models for tasks such as vulnerability prediction. We want to experiment with recently developed deep learning models with deep expressivity capabilities, specifically graph-based models, utilizing abstract syntax trees and control flow graphs to extract more structural information from Solidity source code than just the raw text.

Chapter 4

ETHERBASE: Improving Reproducibility in Smart Contract Research

4.1 Introductory Remarks

In the previous chapter, we reviewed some of the work done on the development of automated analysis tools for discovering and mitigating security vulnerabilities in Ethereum smart contracts. We briefly reported on the traditional methodologies leveraged by tool developers and researchers to make smart contracts more secure, such as static analysis and fuzzing. Moreover, some recent research on developing automated analysis tools for smart contracts, especially those leveraging machine learning-based methods, was surveyed. Afterward, the author's work on SLITHER-SIMIL as a machine learning-based similarity checking tool which can be leveraged for checking the similarity of smart contract source code snippets was presented. In the end, we noted the data scarcity issue for further development of tools such as SLITHER-SIMIL, as machine learning-based tools usually

need to collect troves of data for their models to be suitably trained, tested, and validated. Besides that, data scarcity is also a serious bottleneck for reproducibility in smart contract research and benchmarking of the tools developed for their analysis. These issues led to the development of our work on ETHERBASE.

ETHERBASE is 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.[1] The source code for data acquisition and cleaning will be gradually available due to private IP reasons, and only the collected data will be accessible to the public. Researchers, smart contract developers, and blockchain-centric teams and enterprises can use such corpus for specific use-cases.

4.2 Related Work

a

4.3 Methodology

We designed ETHERBASE in order to have an up-to-date dataset to utilize for research. We have a pipeline 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 the Pragma version, ETH value, and other metrics. To fulfill its purpose, ETHERBASE is designed to perform four primary automatic 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

[1] https://github.com/spilehchiha/etherbase

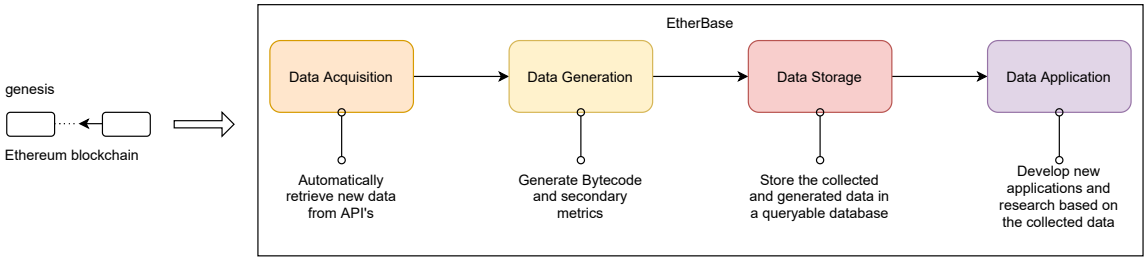


Figure 3: EtherBase Workflow

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

Figure 3 showcases the workflow explained above.

The next sections walk through the steps of the workflow as mentioned earlier one by one.

4.3.1 Data Acquisition

As a user of the website Etherscan, one 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, using Etherscan has its limits [47]. Smart contracts' data is massive, based on the estimation from [47], and the daily limits on the APIs 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. Researchers cannot easily explore the source code for their collected smart contracts. First, they would have to inspect any block on-chain and search for transactions with a receive/send address associated with that specific smart contract.

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 corresponding addresses. In order to collect the contracts, we retrieve the addresses for every contract with more than one transaction through Google BigQuery, similar to the process done by [50]. Using Google BigQuery query request service, we obtain 1,712,347 separate contract addresses with more than one associated transaction.

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, Crytic-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 tricky due to the rapid pace of changing versions of the dominant programming language, Solidity, used to write and develop smart contracts. We utilized Crytic-compile, a library to help compile smart contracts to help with this problem. It helps the user avoid maintaining an interface with solc and automatically finds and uses the correct version of solc or a better compatible version of solc to compile the input smart contracts. This works under the hood because 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.

4.3.2 Data Generation

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 [19] to remove the duplicated smart contracts by 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 tool comparison purposes. Many metrics exist that we have access to, can calculate, and add to ETHERBASE. However, not much research has been conducted on the applicability of these many different 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 that depend on the internal properties of a smart contract, such as SLOC (Source Lines of Code), Pragma version, number of modifiers, and payable. 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 `pragma` keyword is used to enable certain compiler features or checks [4].
`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 deployer of the smart contract.
`Source Code`	Source code of the smart contract, specific to Solidity programming language.
`Bytecode (bin)`	The EVM bytecode of the smart contract corresponding to its source code.
`Bytcode (bin-runtime)`	The code that is actually placed on the blockchain.
`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 terms of the ETH they hold.
`Transaction Count`	Number of internal transactions from each 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 pragma version 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 homogeneous set of contracts with consistent syntax.

- `Contract Address`: Contract address is 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 after the deployment of a contract on the Ethereum Blockchain. The address comes from the creator's address, where the contract has been initially deployed, alongside the number of transactions sent from that address (the "nonce") [2].

- `Source Code`: The source code of each Ethereum smart contract is written in Solidity and helps researchers do all sorts of analyses 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 interacting with smart contracts on Ethereum, both from outside the blockchain ecosystem and for contract-to-contract interaction [3].

- `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 quantity of ETH every smart contract holds is an excellent filter or bar to select "interesting" contracts for further research on the contracts that are more *active* on the blockchain.

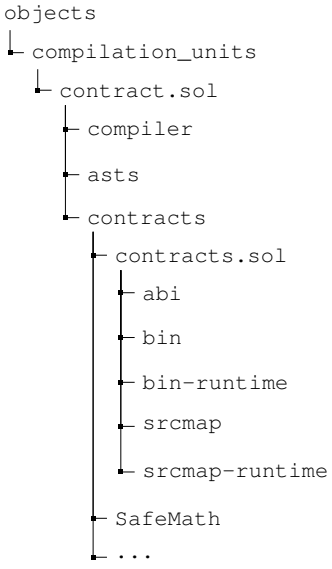
- `Transaction Count`: Like ETH Value, transaction count is an important metric for researchers to exclude contracts that do not participate much on the chain and hence, work on the smart contracts with a higher probability of interaction with more contracts.

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. Afterward, users can analyze their queried data according to their specific research queries in the data application stage.

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 enjoys a graphical user interface (GUI) to allow the less technical end users to access and browse through the database. We integrated ETHERBASE with Apache Superset, a powerful business intelligence tool that allows one to 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 labeled 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 to see how they fare against each other based on an automatically labeled dataset. The criteria we used for tool selection were pretty simple; we wanted tools that had a focus on assessing Solidity source code instead of bytecode and that 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 [53] 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 a CLI tool developed in Python that does security analysis of Ethereum smart contracts.

- **Slither:** Slither [23] is a static analyzer 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 tools mentioned above 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.

- **Arithemtic:** 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 the return values are not managed properly before executing the smart contract [27]. The SWC Registry ID of this contract is SWC-104 [15].

Like the work done by [61], we also leverage the methodology given in the paper by Ren et al. [49] to detect the selected vulnerabilities in smart contracts.

When using tools like these static analyzers, we face many false positive results because of the methods those tools employ. Because of that, we cannot rely on one tool only, as projects that rely upon 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 [61], 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 about the presence of that specific vulnerability in a smart contract, we are allowed to report it as vulnerable.

Based on the proposal from [61], 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 on which the vulnerability happens.

2. Concerning each vulnerability, if different tools show different locations, we should

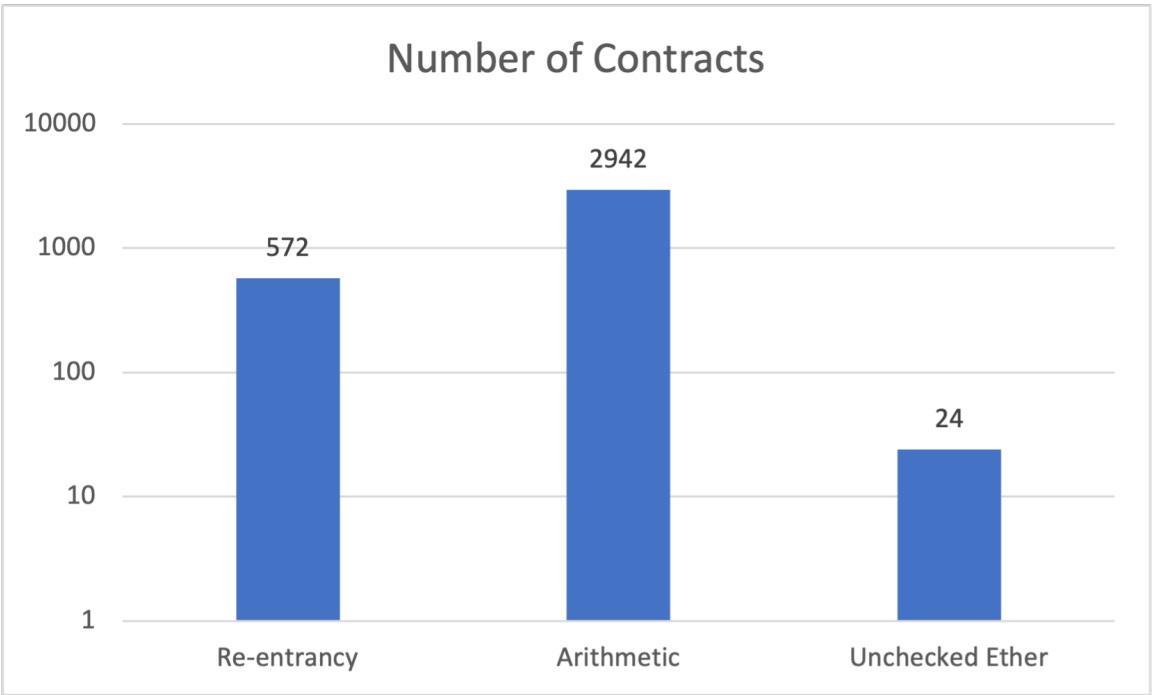


Figure 4: No. of Contracts containing vulnerabilities (log-scale)

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 4 showcases the number of smart contract files which contain at least one potential vulnerability. For instance, consider a specifically targeted vulnerability. The literature tells us that all three static analyzers in the benchmark support the detection of this vulnerability. That means that a majority threshold of the experimented tools states the same vulnerability for the same file. [61] notes that based on the methodology used, we only say the contract at hand contains that vulnerability if an agreed-upon threshold of the tools

reports that it is present at the same location, based on the majority voting system proposed by them. We also take on the same system for determining whether a vulnerability is present.

4.4 Concluding Remarks

This chapter introduces an up-to-date database centered around Ethereum smart contracts, namely ETHERBASE, containing data from the Ethereum blockchain (blocks), its smart contracts and their metadata. Furthermore, future research directions and opportunities are outlined:

While building ETHERBASE, we utilized Web3 APIs without taking advantage of an Ethereum full / archive node. The next version of ETHERBASE we are already working on will use an Ethereum full node and instrument it to add a variety of more data to ETHERBASE. Collecting data via invoking Web3 APIs is very much slower than instrumenting an Ethereum archive node.

In addition, our current method is restricted by the rate limit imposed by the APIs of different services like those of Etherscan and Infura. For example, Etherescan restricts the daily frequency of queries to its API (5 per day). [61] states this as a serious issue that we would like to solve in future work.

The Ethereum security research community can use ETHERBASE for evaluating the correctness and other parameters of their proposed tools 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 [61] notices this as well. There are always many false positive/negative results in an audit report generated by a static analyzer. Nevertheless, we utilized the mechanism of majority voting

suggested by [61] to determine a vulnerability's presence or lack thereof 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 having some auditors manually review the discovered potential vulnerabilities.

Our competitive advantage in comparison to the work done by [61] 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 contract security tools and frameworks.

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

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

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

In the final chapter, we proposed ETHERBASE, an up-to-date database of smart contracts developed with Solidity to facilitate further benchmarking and reproducibility in smart contract research.

For future work, we intend to use an Ethereum full node to instrument it and add a

variety of more smart contract-related data to ETHERBASE. We also plan to utilize more static analysis tools for benchmarking purposes in order to have more expressive ground truth datasets and paint a more accurate picture of their detection capabilities.

Bibliography

[1] Ethereum accounts.

[2] Ethereum documentation, 2021.

[3] Solidity documentation, 2021.

[4] Solidity documentation, pargmas, 2021.

[5] Fasttext documentation, word representations, 2022.

[6] A. M. Antonopoulos and G. Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[7] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura. Eth2vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, pages 47–59, 2021.

[8] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*, pages 164–186. Springer, 2017.

[9] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[10] S. Bhoir, T. Ghorpade, and V. Mane. Comparative analysis of different word embedding models. In *2017 International conference on advances in computing, communication and Control (ICAC3)*, pages 1–4. IEEE, 2017.

[11] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *Transactions of the association for computational linguistics*, 5:135–146, 2017.

[12] 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.

[13] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[14] 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.

[15] Consensys. https://swcregistry.io.

[16] ConsenSys. Mythril, 2022.

[17] P. Daian. Analysis of the dao exploit., 2022.

[18] V. Dhillon, D. Metcalf, and M. Hooper. The DAO hacked. In *Blockchain Enabled Applications*, pages 67–78. Springer, 2017.

[19] 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, pages 530–541, New York, NY, USA, 2020. Association for Computing Machinery.

[20] S. Eskandari, S. Moosavi, and J. Clark. Sok: Transparent dishonesty: front-running attacks on blockchain. In *International Conference on Financial Cryptography and Data Security*, pages 170–189. Springer, 2019.

[21] W. Ethereum. Ethereum whitepaper. *Ethereum. URL: https://ethereum. org [accessed 2020-07-07]*, 2014.

[22] 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.

[23] 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.

[24] 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.

[25] C. Ferreira Torres. From smart to secure contracts: Automated security assessment and improvement of ethereum smart contracts. 2022.

[26] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy. Checking smart contracts with structural code embedding. *IEEE Transactions on Software Engineering*, 2020.

[27] N. Group. Dasp top 10. https://www.dasp.co/, 2019.

[28] M. E. Hellman. An overview of public key cryptography. *IEEE Communications Magazine*, 40(5):42–49, 2002.

[29] 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.

[30] 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.

[31] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[32] S. King and S. Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19(1), 2012.

[33] M. Kondo, G. A. Oliva, Z. M. J. Jiang, A. E. Hassan, and O. Mizuno. Code cloning in smart contracts: a case study on verified contracts from the ethereum blockchain platform. *Empirical Software Engineering*, 25(6):4617–4675, 2020.

[34] L. Lessig. *Code: And other laws of cyberspace*. ReadHowYouWant. com, 2009.

[35] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen. A survey on the security of blockchain systems. *Future Generation Computer Systems*, 107:841–853, 2020.

[36] 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.

[37] 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.

[38] 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.

[39] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[40] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[41] P. Momeni, Y. Wang, and R. Samavi. Machine learning model for smart contracts security analysis. In *2019 17th International Conference on Privacy, Security and Trust (PST)*, pages 1–6. IEEE, 2019.

[42] 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.

[43] 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.

[44] T. of Bits: Slithir documentation. https://github.com/crytic/slither/wiki/SlithIR, 2018.

[45] Parity. https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/.

[46] Peckshield. Arithmetic vulnerabilities, 2018.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] S. Srikant. *Vulcan: classifying vulnerabilities in solidity smart contracts using dependency-based deep program representations*. PhD thesis, Massachusetts Institute of Technology, 2020.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[59] 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.

[60] Y. Xu, G. Hu, L. You, and C. Cao. A novel machine learning-based analysis model for smart contract vulnerability. *Security and Communication Networks*, 2021, 2021.

[61] 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.

[62] L. Zhang, Y. Li, T. Jin, W. Wang, Z. Jin, C. Zhao, Z. Cai, and H. Chen. Spcbigec: A robust serial hybrid model for smart contract vulnerability detection. *Sensors*, 22(12):4621, 2022.

[63] 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.