

Improving Reproducibility in Smart Contract Research

Sina Pilehchiha

A Thesis
in
The Department
of
Electrical and Computer Engineering

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

July 2022

© Sina Pilehchiha, 2022

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Sina Pilehchiha**

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

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

Master of Applied Science (MAsc)

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

Signed by the final examining committee:

Dr. Yousef R. Shayan Chair

Dr. Examiner

Dr. Examiner

Dr. Amir G. Aghdam Thesis Supervisor

Dr. Jeremy Clark Thesis Supervisor

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

July 31, 2022

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

Abstract

Improving Reproducibility in Smart Contract Research

Sina Pilehchiha

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 The datasets frequently get out of date very quickly on the Ethereum platform because of its rapid development.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

Acknowledgments

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

Contents

List of Figures	vii
List of Tables	viii
1 Introduction	1
1.1 Motivation	1
1.2 Thesis Statement	2
1.3 Outline and Contributions	3
2 Background	4
2.1 Ethereum	4
2.1.1 Ether	7
2.1.2 Accounts	8
2.1.3 Transactions	8
2.1.4 Blocks	10
2.1.5 Ethereum Virtual Machine	12
2.2 Smart Contracts	12
2.2.1 Vulnerabilities	12
3 Automated Vulnerability Analysis of Smart Contracts on Ethereum	17
3.1 Introductory Remarks	17

3.2	Traditional Security Analysis Methods in Smart Contracts	18
3.3	Deep Learning in Smart Contracts	20
3.4	SLITHER-SIMIL	30
3.5	Concluding Remarks	37
4	ETHERBASE: Improving Reproducibility in Smart Contract Research	38
4.1	Introductory Remarks	38
4.2	Related Work	40
4.3	Methodology	42
4.3.1	Data Acquisition	43
4.3.2	Data Generation	44
4.3.3	Data Storage	48
4.3.4	Data Application	50
4.4	Concluding Remarks	53
5	Conclusion and Future Work	55
	Bibliography	56

List of Figures

1	Ethereum blockchain structure.	5
2	An illustration of Ethereum’s GHOST protocol.	7
3	A high-level view of the process workflow of Slither-simil.	32
4	EtherBase Workflow	43
5	No. of Contracts containing vulnerabilities (log-scale)	52

List of Tables

1	Primary Metrics on Smart Contracts	46
2	Supported Vulnerabilities	51

Chapter 1

Introduction

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

1.1 Motivation

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

Famous attacks that have caused significant financial losses and prompted the research community to work to prevent similar occurrences include The DAO hack [6] and the Parity wallet issue [35]. Comparing and reproducing such research is not an effortless process. The datasets used to test and benchmark those very same tools proposed in the research literature are not publicly available and this makes reproduction efforts immensely hard to

carry out.

If the developer of a new tool or a researcher intends to compare their new tool with the existing work and projects, the current approach is to contact the authors of those alternative tools and hope for access to the same datasets used in the original research / work or make do with whatever out-of-date incomprehensive and unrepresentative dataset they have at their disposal, a very timely and inefficient process. [29]

In other cases, the researchers need to start from scratch and create their own datasets, a non-trivial and slow process. What makes it worse is the data bias, which can be introduced in a dataset in different phases of data acquisition and data cleaning. This can easily escalate to become a threat to validity in the research. [41]

1.2 Thesis Statement

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

In summary, we make the following contributions.

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

and renewed, in comparison to the previous manual one-time data gathering efforts.

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

1.3 Outline and Contributions

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

In Chapter 3, we summarize and evaluate the 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 own efforts in developing and working on the tool Slither-similar as an effort in such direction.

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

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

Chapter 2

Background

This chapter reviews the necessary background knowledge for the reader to be better acquainted with the work conducted within this thesis. It highlights the foundational technical basics of Ethereum blockchain, smart contracts and the most common vulnerabilities associated with smart contracts. We cover these technicalities in the following order (based on the work of [15]): First, we go over the basics of Ethereum, its components and structure. We will go through how blocks are formed, what sort of accounts exist on the Ethereum network, and how transactions are executed. We will also go through how the Ethereum Virtual Machine functions. Afterwards, we will go over smart contracts and their most common vulnerabilities out in the wild. We will discuss Solidity-written source code and bytecode of smart contracts, and explain each vulnerability according to the DASP 10 classification. [20]

2.1 Ethereum

Ethereum is a decentralized virtual machine that was introduced as an alternative blockchain technology to Bitcoin in 2014 by [51]. A blockchain is essentially a peer-to-peer network made up of computers that act as nodes and distribute updates for a single database without



Figure 1: Ethereum blockchain structure.

necessarily having confidence in one another. It is based on a combination of combination of cryptography, networking, and incentive mechanisms. [50] The aforementioned database effectively serves as a ledger, recording each and every transaction that each node in the blockchain network makes.

As the second most popular blockchain, the Ethereum blockchain was developed as an alternative to cover for the lackings of Bitcoin. It is a transaction-based, cryptographically secure state machine, that reads a series of inputs and, based on those inputs, transitions to a new state. [15] Just like Bitcoin, Ethereum currently uses Proof-of-Work (PoW) as its consensus protocol. Proof-of-Stake (PoS), PBFT(Practical Byzantine Fault Tolerance), and DPoS (Delegated Proof of Stake) are other forms of consensus protocol, for example. The solution to a series of cryptographic puzzles are used in the PoW mechanism to prove the credibility of the data being written on the blockchain, using such mechanism as their consensus protocol. The puzzle is usually a computationally hard but easily verifiable mathematical problem. When a node creates a block, it must resolve a PoW puzzle and spend computing power to achieve so. The nodes compete o=with each other over this objective function and the node with the most computing power usually succeeds ind doing so. After the PoW puzzle is resolved, it will be broadcasted to other nodes, so as to achieve the purpose of consensus and append a new block to the blockchain. It acts as a “proof” that a node has done “work” by spending its computational resources. This process is known as

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

As participating nodes are allowed to propose new blocks simultaneously and at nearly the same time, it can be the case that two or more blocks are proposed at once with a valid hash while referencing the same parent block. This is called a "fork" in the blockchain nomenclature. Blockchain forks pose a serious issue to Ethereum. They result in multiple concurrent states and make it difficult for the nodes to come to an agreement concerning which state should be the correct state. For example, if the states were to diverge, a user might own 100 coins on one chain, 200 on another, and 300 on another. Forks can happen intentionally or be deliberately orchestrated. To prevent multiple states and help determine which fork is the most valid one, Ethereum blockchain uses a technique known as the Greedy Heaviest Observed Subtree (GHOST) protocol. The GHOST protocol declares that of all the states, one must select the state that contains the most computation. One way to determine the heaviest state (computation-wise), is to look at the block number of the most recent block (i.e., leaf block) of a state, which amounts to the total number of blocks in the current state (not including the genesis block). The higher the block number, the longer the chain of that state and the greater the mining computation that must have gone into arriving at that current leaf block. This allows the blockchain nodes to agree on an accepted version of the current state of the blockchain. Those blocks that are not included in the canonical chain are often referred to as orphans or uncle blocks. Unlike other blockchains, such as Bitcoin, Ethereum also adds uncle blocks to the aforementioned calculation to figure out the longest and computationally heaviest chain of blocks. This allows for the inclusion of more transactions and attributes a reward to the creators of uncle blocks as well as miners for declaring concurrent blocks as uncles and thereby keeping forked chains as short as possible.



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

2.1.1 Ether

Ether is the native cryptocurrency of Ethereum. A cryptocurrency is a digital currency that is secured by means of cryptographic primitives. The purpose of ether is not only to allow users to exchange value between one another, but also to provide an economic incentive for users of its underlying platform to provide computational resources to the Ethereum network. Any network participant who sends a transactions must offer some amount of ether to the Ethereum network as a fee. This remuneration will be awarded to whoever gets to mine the block that includes the transaction, as a result of doing the work of verifying, executing, and broadcasting the transaction to the rest of the blockchain network. The amount of ether offered in such transaction as the corresponding fee, must correspond to the time and effort spent in executing the transaction. These costs prevent malicious participants from intentionally blocking up the network by requesting the execution of infinite computation or other resource-intensive actions, as these ill-intentioned participants must pay for the computation.

Ethereum provides a metric system of denominations to describe different units of ether. Each denomination has its own unique name. Wei and Gwei are the most popular denominations. Wei is the smallest possible denomination of ether also known as the base unit, and as a result, many technical implementations base all their calculations in wei. Gwei (short for gigawei) is often used to describe costs related to “gas”.

2.1.2 Accounts

The Ethereum state is consisted of many small objects naemd as accounts, where each account has a 20-byte address and state transitions to be able to interact with the other accounts on-chain. An address on the Ethereum blockchain is a 160-bit identifier that is used to identify any account. The world state is a mapping between addresses and account states. [51] Ethereum suppotrs two types of accounts: externally owned (controlled by private keys) -namely EOA's- and contract accounts (CA'S, controlled by their contract code) [11]. Inside of an Ethereum account is composed of four fields: nonce, ether balance, contract code hash, and storage root, explained as follows:

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

2.1.3 Transactions

Transactions are essentially cryptographically signed instructions from EOAs to update the state of the Ethereum blockchain. EOAs sign their transactions using their private key in order to cryptographically prove that the transaction could only have come from them

and not from someone else. Two types of transactions exist: message calls and contract creations. The latter are transactions with an empty recipient field that result in creating new CAs (i.e., smart contracts). The code, to be associated with the CA, is placed inside the data field of the transaction. Regardless of their type, all transactions contain the following fields:

- A count on the number of transactions sent by the sender. This number is incremented by one every time a transaction is sent by the sender.
- The amount of wei that the sender is willing to pay for each unit of gas that is used during the execution of the transaction.
- The maximum amount of gas units that the sender is willing to spent for the execution of this transaction. This amount is set and paid upfront, before any computation is performed.
- The address of the recipient. If the recipient is an EOA, the transaction will transfer value, if the recipient is a CA, the transaction will transfer value as well as execute the contract's code. A transaction with an empty recipient address is used to trigger the creation of a new CA.
- The amount of wei to be sent from the sender account to the recipient account. Interestingly, this value may be used to set the starting balance of the newly created CA in a contract-creating transaction.
- These values represent the digital signature (R, S) which can be used to recover the public key (V). These values identify the sender of the transaction and confirms that the sender has authorized the transaction.
- This field is only part of contract-creating transactions and consists of an unlimited length byte array that includes the code to be used during the initialization process

and the code to be permanently associated with the newly created CA.

- This is an optional field that is only part of message calls and consists of a byte array of unlimited size that specifies the input data (e.g., , function name, function parameters) of the message call.

As previously mentioned, contract-creating transactions and message calls are always initiated by EOAs. However, this does not mean that CAs cannot communicate with other CAs. CAs can send messages or so-called “internal transactions” to other CAs. Internal transactions are similar to normal transactions, with the major difference being that they are not initiated by EOAs, but instead they are initiated by CAs. Moreover, internal transactions merely exist as virtual objects that, unlike transactions, are not persisted in the Ethereum blockchain and only exist at execution time. When a CA sends an internal transaction to another CA, the code that is associated with the recipient CA is executed. In contrast to normal transaction, internal transactions do not contain a gas limit by default. They are only limited by the gas limit that was determined by the normal transaction that triggered them. Thus, the gas limit that the EOA provides within its normal transaction, must be large enough to perform the execution of the normal transaction, including any sub-executions that occur as a result of that transaction, such as any internal transactions. If the execution of an internal transaction runs out of gas, then its execution will be will be reverted, along with any subsequent internal transactions triggered by the execution. However, the parent execution is not reverted.

2.1.4 Blocks

Every Blockchain, including Ethereum, consists of a sequence of blocks, which are bound and "chained" together through cryptographic hashes, grouping transactions altogether. A block on the Ethereum blockchain consists of a body and a header. The block header contains metadata about its own specific block. The first block of the Blockchain is called

genesis which has no parent block. An ommer is a block whose parent is equal to the current block's parent's parent. A block header is a portion of the block consisting of the following apart from the transactions listed inside:

- **ParentHash:** This is the Keccak-256 hash of the parent block's header.
- **OmmersHash:** This is the Keccak-256 hash of the ommer's list portion of this block.
- **Beneficiary:** The account address that receives the fees for mining this block
- **LogsBloom:** A Bloom filter (i.e., a data structure) that allows efficient querying of information contained in the logs.
- **Difficulty:** the difficulty level of this block, i.e. the required effort for mine a block.
- **Number:** the count of current block (the genesis block has a block number of zero; the block number increases by 1 for each each subsequent block)
- **GasLimit:** The current gas limit per block.
- **GasUsed:** the sum of the total gas used by transactions in this block.
- **ExtraData:** Arbitrary data that can set by the miner. This data is limited to 32-byte and usually refers to the name of the miner or the client version that was used to mine the block.
- **StateRoot:** the hash of the root node of the state trie (recall how we learned that the state trie is stored in the header and makes it easy for light clients to verify anything about the state)
- **Timestamp:** The UNIX timestamp when the block was mined.
- **Nonce:** A hash that, when combined with the mixHash, proves that this block has carried out enough computation

- **MixHash:** A hash that, when combined with the nonce, proves that this block has carried out enough computation
- **TransactionsRoot:** The hash of the root node of the Merkle Patricia trie that stores all transactions listed in this block.
- **ReceiptsRoot:** The hash of the root node of the Merkle Patricia trie that stores the receipts of all transactions listed in this block. Transaction receipts are generated after the execution of a transaction contain information such as logs or the actual gas that has been used during execution.

2.1.5 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) at the heart of the Ethereum blockchain is a VM (virtual machine), with a stack-based architecture, supporting Turing-complete programming languages.

2.2 Smart Contracts

The concept of smart contracts - programs running on the EVM - has been first introduced by Nick Szabo in one of his works in 1997. [45]

2.2.1 Vulnerabilities

Solidity, like any other programming language in history, is prone to all kinds of vulnerabilities. What makes security vulnerabilities in Solidity so attractive is the fact that the programs written in Solidity are very much often used in the financial sector, handling millions of dollars in digital assets and cryptocurrencies. Attacking such contracts successfully

can result in enormous financial losses. Some of these vulnerabilities like another programming language arise from the human factor involved in development of the smart contracts, and some specific to the blockchain data structures and how they and their components function and interact with each other. And these are only vulnerabilities at the scope of smart contracts we are focusing on. Vulnerabilities can arise with regards to core infrastructure of the blockchains handling smart contracts as well. In this section, we go over 9 of the more discussed vulnerabilities in Solidity and Ethereum, to get a better sense of what threat surface the developers and researchers developing analysis tools face:

Re-entrancy The Reentrancy attack, probably the most famous Ethereum vulnerability, surprised everyone when discovered for the first time. It was first unveiled during a multi-million dollar heist which led to a hard fork of Ethereum. Reentrancy occurs when external contract calls are allowed to make new calls to the calling contract before the initial execution is complete. For a function, this means that the contract state may change in the middle of its execution as a result of a call to an untrusted contract or the use of a low level function with an external address.

Access Control Access Control issues are common in all programs, not just smart contracts. In fact, it's number 5 on the OWASP top 10. One usually accesses a contract's functionality through its public or external functions. While insecure visibility settings give attackers straightforward ways to access a contract's private values or logic, access control bypasses are sometimes more subtle. These vulnerabilities can occur when contracts use the deprecated `tx.origin` to validate callers, handle large authorization logic with lengthy require and make reckless use of `delegatecall` in proxy libraries or proxy contracts.

Arithmetic Integer overflows and underflows are not a new class of vulnerability, but they are especially dangerous in smart contracts, where unsigned integers are prevalent

and most developers are used to simple int types (which are often just signed integers). If overflows occur, many benign-seeming codepaths become vectors for theft or denial of service.

Unhandled Exception One of the deeper features of Solidity are the low level functions `call()`, `callcode()`, `delegatecall()` and `send()`. Their behavior in accounting for errors is quite different from other Solidity functions, as they will not propagate (or bubble up) and will not lead to a total reversion of the current execution. Instead, they will return a boolean value set to false, and the code will continue to run. This can surprise developers and, if the return value of such low-level calls are not checked, can lead to fail-opens and other unwanted outcomes. Remember, `send` can fail!

Denial of Service Denial of service is deadly in the world of Ethereum: while other types of applications can eventually recover, smart contracts can be taken offline forever by just one of these attacks. Many ways lead to denials of service, including maliciously behaving when being the recipient of a transaction, artificially increasing the gas necessary to compute a function, abusing access controls to access private components of smart contracts, taking advantage of mixups and negligence, etc. This class of attack includes many different variants and will probably see a lot of development in the years to come.

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

- Displacement attack, where an adversarial party makes a transaction in order to displace the victim user's transaction by having a higher gas price and thus, the attacker's transaction gets mined before that of victim's due to it giving having more aligned incentives with the miners' network.
- Insertion attack, in which an adversary makes two transactions, one with a higher gas price than the victim transaction and one with a lower gas price, to *sandwich* the victim transaction. [48]
- 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*, [20] this vulnerability happens when smart contracts attempt to generate random, or to be more exact, pseudo-random numbers for any number of reasons. If the smart contract generating the pseudo-random number computes that random number using values that can be guessed by a malicious party, then the attacker can predict the next number that will be generated. Values such as block timestamps, or block number are generally advised against to be used in such mechanisms. They are called hard-to-predict values but it is better to use an external oracle to generate the random numbers needed [4].

Time Manipulation This vulnerability is also known as *timestamp dependence* [20]. In Solidity, a block's timestamp is often used to generate pseudo-random numbers and in other times, it can be leveraged for smart contracts to conduct time-intensive operations, like unlocking funds at a specific time. A malicious miner of a block can manipulate the timestamp reported while generating the block and use this vulnerability to their own profit.

Short Address The short address vulnerability, also known as off-chain issues, results from the Ethereum Virtual Machine accepting arguments with incorrect paddings. Attackers trying to exploit this vulnerability, can craft truncated addresses which clients may encode incorrectly in transactions. Additioally, it has not been exploited in the wild as mentioned by [14].

Chapter 3

Automated Vulnerability Analysis of Smart Contracts on Ethereum

3.1 Introductory Remarks

Smart contracts, the universal and vital programs that are deployed on blockchains, have gained increasing attention with the rapid development of blockchains. For example, more than 10 million smart contracts have been deployed on the Ethereum Mainnet.

smart contract is an event-driven, self-executing, state-based program that is written in high level languages such as Vyper and Solidity. Smart contracts have been widely used in many business domains to enable efficient and trustful transactions.

Unlike general programs, the development of smart contracts requires special effort due to their unique characteristics. First, smart contracts are more bug intolerant compared with general programs. “Code is law”, a smart contract can not be modified once it has been released. This is because transactions of a smart contract always involve cryptocurrencies which are worthy of millions of dollars (e.g. The DAO). A bug in a smart contract may lead to a substantial loss. Therefore, ensuring the correctness of contracts before releasing is critical. This requires us to reuse experience of developed contracts in the past when

developing new contracts. Program mining for smart contracts such as summarization, checking, and code search can greatly facilitate the development and maintenance of smart contracts. The conventional statistical analysis tools for detecting weaknesses in smart contracts purely rely on manually defined patterns, which are likely to be error-prone and can cause them to fail in complex situations. As a result, expert attackers can easily exploit these manual checking patterns. To minimize the risk of the attackers, machine learning powered systems provide more secure solutions relative to hard-coded static checking tools.

Surucu et al. [44] provide the first-ever survey on machine learning methods utilized for the purposes of discovery and mitigation of vulnerabilities in smart contracts. In order to set the ground for further development of ML method on smart contract vulnerability detection, They reviewed many ML-driven intelligent detection mechanism on the following databases: Google Scholar, Engineering Village, Springer, Web of Science, Academic Search Premier, and Scholars Portal Journal. Based on their survey paper, we briefly go over the existing analysis tool first, and the the novel deep-learning-based methodologies proposed in the literature over the past few years in a chronological order, and we add some of the works missing in [44] as well. Afterwards, we propose our own solution, SLITHER-SIMILand how it led us to the development of ETHERBASE.

3.2 Traditional Security Analysis Methods in Smart Contracts

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

In the following, we will go over the tools proposed from the perspective of the technology they employ to tackle the smart contract security problem; Ren et al. [41] provide three broad categories of tools based on their utilized methodology, namely Static Analysis,

Dynamic Fuzzing, and Symbolic Execution.

With static analysis, you can analyse the program at both the source code and bytecode levels, before runtime execution. Static analysis-based tools can scan a whole code base, but they also generate a lot of false positives as a result of their scans. There are normally three main stages to a static analysis process:

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

Normally, other tools will first obtain binary bytecode, and then use it to construct a custom intermediate representation, which have a series of forms, like SSA used in Slither [13], Datalog used in Securify [47] and MadMax [18], XML parsing tree used in SmartCheck [46] and XCFG used in Clairvoyance [53]. Based on this representation, vulnerability pattern definition and matching are performed to screen out suspected code snippets. Formal verification, a static analysis technique based on mathematical proof, is frequently used to check the logical integrity of smart contracts like Zeus [26], VeriSmart [42], and EtherTrust [19]. One critical issue with using static analysis to look for vulnerabilities in smart contracts is watching out for false positives and false negatives in their generated reports and not mistaking them for pure ground truth values for further analysis or benchmarking purposes.

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

to generate initial seeds to form executable transactions. With reference to the feedback of test results, it will dynamically adjust the generated data to explore as much smart contract state space as possible. Finally, it will analyze the status of each transaction based on the finite state machine to detect whether there is an attackable threat. ContractFuzzer [24] is the first to apply fuzz testing to smart contracts. Later, other researchers start to study improvements to different parts of fuzzing. ReGuard [27] and Harvey [52] are dedicated to generating diverse inputs and transactions that are more likely to reveal vulnerabilities, ILF [21] and sFuzz [32] target at designing more effective generation or mutation strategy.

When using symbolic execution to analyze a program, it will use symbolic values as input instead of the specific values during the execution. When a fork is reached, the analyzer will collect the corresponding path constraints, and then use a constraint solver to obtain specific values that can trigger each branch. Symbolic execution can simultaneously explore multiple paths that the program can take under different inputs, but it also faces unavoidable problems such as path explosion. In most cases, the symbolic executor will first build a control flow graph based on Ethereum bytecode, then design corresponding constraints based on the characteristics of smart contract vulnerabilities, and finally use the constraint solver to generate satisfying test cases; for example, Oyente [30], Mythril [5]. In recent years, there has been continuous research to optimize the process of symbolic execution. Manticore [31] adds the support of exotic execution environments, DefectChecker [3] extracts defect related features to help improve efficiency, sCompile [2] identifies critical paths which involve monetary transaction and VerX [36] focuses on verifying effectively external callback free contracts.

3.3 Deep Learning in Smart Contracts

In this section, we will go over some of the literature focusing their efforts on replacing the existing tools' capabilities explained in the previous section with machine learning-based

techniques. Afterwards, we will go over our own developed tool, SLITHER-SIMIL.

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

Goswami et al. mentioned that while existing symbolic tools (e.g., Oyente) for analyzing vulnerabilities have proven to be efficient, their execution time increases significantly with depth of invocations in a smart contract [17]. They proposed an LSTM neural network model to detect vulnerabilities in ERC-20 smart contracts in an effort to produce a less time consuming and efficient alternative to symbolic analysis tools. The preprocessing steps followed in this paper were very similar to the methods used by [18]. The model was trained and tested on a dataset of 165,652 ERC-20 smart contracts, which consisted of bytecode data labeled by Maian and Mythril (statistical code analysis tools). The proposed model achieved 93.26% accuracy, 92% recall and an F1 score of 93% on the testing set. Further they have compared the time performance of their model to those of the symbolic analysis tools Maian and Mythril (static analysis tools). While their proposed model had a runtime of 15 seconds on a testing set of 5,000 random tokens, Maian and Mythril took 32,476 and 9,475 seconds respectively. These results indicate the same type of improvement achieved over symbolic analysis tools as in [17].

Liao et al. have adopted a sequence learning approach to detect smart contract security threats [18]. Smart contract data was obtained from the Google Big Query Ethereum blockchain dataset. Ultimately, an LSTM model was trained on 620,000 contracts from this source. Once again, the derived opcodes from the contracts were represented as one-hot vectors. As this type of representation results in highly sparse and uninformative features, these vectors were transformed into code vectors using embedding algorithms, resulting in lower dimensionality and a higher capability of capturing potential relationship between sequences. As another preprocessing step, they have compared the statistical properties

of the opcode lengths of contracts that were identified as vulnerable and safe. Having observed that the properties of the two categories differ significantly, they have limited the input data to the LSTM to only include contracts that had a maximum opcode length of 1600, as a design choice. Further, the distribution of the dataset (labeled by MAIAN) was realized to be imbalanced with non-vulnerable instances making up 99.03% of the dataset. Therefore, all vulnerable contracts were grouped together and oversampled to achieve a balanced distribution in the training set using the Synthetic Minority Oversampling Technique (SMOTE). The results indicated the superiority of a sequential learning approach over symbolic analysis tools. The model achieved a vulnerability detection accuracy of 99.57% and F1 score of 86.04%.

SoliAudit model was proposed to enhance the vulnerability detection of smart contracts [19]. Smart contract source code in Solidity is converted into an opcode sequence to preserve the structure of executions. Each contract goes through both a dynamic fuzzer and a vulnerability analyzer. The vulnerability analyzer consists of a static machine learning classifier, which detects vulnerable classes, whereas the fuzzer (this term was introduced in an earlier paper) will parse the Application Binary Interface (ABI) of a smart contract to extract its declared function descriptions, data types of their arguments and their signatures. It will then return the smart contract inputs and functions that are identified as vulnerable. The idea of a smart contract fuzzer was introduced by the authors of [19]. Vulnerability analyzer used a set of labels (13 vulnerabilities) determined by analysis tools such as Oyente and Remix. Before training the opcode sequence data using these labels, two types of feature extraction methods were tested. These were namely, n-gram with tf-idf and word2vec. The experiments were carried out by applying the former method together with algorithms such as Logistic Regression, Support Vector Machine, K-Nearest Neighbor, Decision Trees, Random Forests and Gradient Boosting. The output from the latter (word2vec) was a matrix and a Convolutional Neural Network (CNN) was preferred to

train it as it considers the inner structure of the matrix. However, this combination of feature extraction and training did not yield good results. The best results for the classification of vulnerabilities were obtained using Logistic Regression with an accuracy of 97.3% and F1 score of 90.4%.

Xing et al. [22] developed a new feature extraction method called slicing matrix, which 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) [22]. These three models were each tested on three different vulnerability classification tasks: greedy contract vulnerability, arithmetic overflow/underflow vulnerability and short address vulnerability. While RFBOOF achieved the best results in all three cases based on precision, recall and F1 evaluation metrics, CNNBOSM performed slightly better than NNBOOF in general. The authors mention that the slice matrix feature need further exploring.

Momeni et al.

SmartEmbed, Ethereum has become a widely used platform to enable secure, Blockchain-based financial and business transactions. However, a major concern in Ethereum is the security of its smart contracts. Many identified bugs and vulnerabilities in smart contracts not only present challenges to maintenance of blockchain, but also lead to serious financial losses. There is a significant need to better assist developers in checking smart contracts and ensuring their reliability. In this paper, we propose a web service tool, named SMARTEMBED, which can help Solidity developers to find repetitive contract code and clone-related bugs in smart contracts. Our tool is based on code embeddings and similarity checking

techniques. By comparing the similarities among the code embedding vectors for existing solidity code in the Ethereum blockchain and known bugs, we are able to efficiently identify code clones and clone-related bugs for any solidity code given by users, which can help to improve the users' confidence in the reliability of their code. In addition to the uses by individual developers, SMARTEMBED can also be applied to studies of smart contracts in a large scale. When applied to more than 22K solidity contracts collected from the Ethereum blockchain, we found that the clone ratio of solidity code is close to 90%, much higher than traditional software, and 194 clone-related bugs can be identified efficiently and accurately based on our small bug database with a precision of 96%.

In N. Lesimple et al.'s paper [7], the authors study the effect of deep learning models when used to identify vulnerabilities in Smart Contracts. It specifically highlights the vulnerabilities relating to Domain Specific Languages (DSL), which is defined as a language engineered to work solely on a single program. This is highly relevant for blockchain, as Solidity was specifically designed for Ethereum, and therefore is a DSL. The authors then identify some common vulnerabilities in traditional smart contract code, and examine issues with traditional vulnerability checking techniques. Of these, one of the most important issues with traditional techniques is that the subset of bugs found are due to the strict predefined inputs that are used. The paper proposes that, through the use of Deep Learning, the input can be varied significantly to identify faults that the predefined static tests would otherwise not. The authors then propose a novel approach, which analysis the line level code and trains a Deep Learning Neural Network to understand the control paths and data transformations occurring in the code [7]. As an input to the model, to allow for the model to understand the code on a line level, the authors used an Abstract Syntax Tree (AST) structure, which relates variables to one another, marking their dependencies and transformations throughout the code. The author analyzed several Natural Language Processing techniques, and Recurrent Neural Networks, and eventually landed on using an

LSTM network to train their model. They found that LSTM's outperformed most RNN models, and due to the vast variety in code syntax, the NLP techniques were unable to interpret many situations, since the code and inputs were inconsistently structured. Their results were quite accurate, but it is important to note that the results were tested against results from a traditional model that they were actually attempting to replace. If this paper could acquire a test set of vulnerabilities that were not acquired through the use of a traditional method, the results would be more poignant.

Liu Z. et al. proposed a combining GNN and expert knowledge based machine learning model for detecting various smart contract vulnerabilities [23]. A graph neural network (GNN) is a deep learning method, where the principle is to perform inference on data described by graphs. In computer science, a graph is a data structure consisting of two components: nodes (vertices) and edges. Researches have proven that written programs can be converted to symbolic graph representation, without disrupting semantic relationship between programming elements. Thus, smart contract codes can be represented as contract graphs. The proposed model consists of two different parallel processes (Security pattern extraction and contract graph extraction) at the beginning, and the combining layer merged patterns in each section to find vulnerabilities, as shown in figure 3. First, a feed-forward neural network generates the pattern feature for extracting security patterns from the contract's source code. They have used an opensourced tool to extract the expert patterns from smart contract functions. The second process (message propagation phase) is to create a GNN to achieve a contract graph. Inside the GNN model, nodes were the program elements (i.e., function), where edges represented the flow (i.e., next function to be executed) of each program elements. Later, unwanted nodes and edges are removed based on a node elimination strategy. As a preprocessing method, the authors casted rich control and data flow semantics of the source code into a contract graph. After

this step, they designed a node elimination stage to highlight critical nodes by normalizing the graph. These two parallel processes were combined using vulnerability detection phase, where both extracted features are combined convolution and full-connected layer. In experiment, the proposed model is compared with non-ML-based security detection algorithms, namely Oyente, Myhrill, Smartcheck, Securify, and Slither. Each algorithm and the proposed model performed a search of several vulnerabilities (re-entrancy, timestamp dependence, and infinite loop vulnerabilities) of each function in the source code. The proposed algorithms (CGE) achieved 89% accuracy on finding re-entrancy and timestamp dependence type of vulnerabilities, and 83% accuracy on detecting infinite loop vulnerability [23].

Eth2Vec model is proposed to deficiency in current vulnerability detection tools when a code is rewritten. In programming languages, a code rewrite is reimplementing a source code's functionality without reusing it. When the smart contract codes are rewritten, detecting vulnerabilities become harder. The authors first converted each smart contract source code into EVM bytecodes. From the bytecode, the authors extracted only valuable information (i.e., function id, list of callee functions etc.) for vulnerability detection. As the last process, a neural network structure is used to catch any vulnerabilities in the source code. After testing the proposed model on 500 contracts, the Eth2Vec model was able to detect vulnerabilities with a 77% precision even though the contracts are rewritten.

O. Lutz et al. [8] introduce yet another method of detecting vulnerabilities within smart contracts. The authors propose a solution entitled ESCORT, wherein they use a Deep Neural Network model to learn the semantics of the input smart contract, and learn specific vulnerability types based on the found semantics. The goal of the ESCORT model is to overcome the scalability and generalization limitations of traditional non-DNN models. Experimental results of this paper yielded an F1 accuracy score of 9% on six found vulnerability types, with a detection time of 0.02 seconds per contract. With such quick

detection times, scalability is more easily achieved, satisfying one of the author's goals. Then, through the use of transfer learning, the ESCORT model slightly overcomes the issues found in other papers, such as Y. Xu or N. Lesimple's models [17], where newfound vulnerabilities can be realized by the model. Unfortunately, it is rather difficult to obtain interpretability from such models, and though new vulnerabilities may be found, understanding their cause remains to be exceedingly difficult.

Sun et al. have attempted to detect the following vulnerabilities: re-entrancy, arithmetic issues (integer overflow/underflow) and timestamp dependence using machine learning [17]. As a common prerequisite step, some stackoperating instructions were truncated into more general forms (e.g., SWAP1, SWAP2, ..., SWAPn. \rightarrow SWAPx) to account for variations in instructions among different compilers. Following this, opcodes were separated into 9 categories based on their functions, as a label normalization step. As in [19] a word2vec transformation of the opcode sequences, preceding the convolutional layers, was performed. In addition to the pooling and softmax layers that commonly follow convolutional layers, this paper introduces an additional self-attention layer. The purpose of the self-attention layer is to create a connection between adjacent words in the obtained feature matrix since one-hot encoders that were used to encode each opcode instruction are just mere representatives and do not capture any functional similarity between them [17]. As a result, the word embedding process has been enhanced through the use of self-attention. When compared to the vulnerability detection performance of [19], they have both used a CNN but [19] used a word2vec embedding whereas this paper employed an attention mechanism, which is the likely reason that they obtained better results. obtained better results. The main improvement of the created model over the existing static analyzers such as Oyente and Mythril is that it can achieve comparable performance in much less time.

A vulnerability and transaction behaviour-based detection is proposed [24] In this

work, the authors built a model that correlates malicious activities, and the vulnerabilities present in smart contracts. In respect to strength of the correlation unsupervised ML models (K-means and HDBSCAN) assign a severity score to each smart contract. The model was trained to detect suspects among benign smart contracts. The aim of the research was to test their hypothesis, which was “the transaction behavior is a more critical factor in identifying malicious smart contracts than vulnerabilities in the smart contract.” Thus, they brought a different perspective to the literature of smart contracts vulnerability detection.

Y. Xu et al.’s paper introduced two novel smart contract vulnerability detecting approach using both a KNearest Neighbors (KNN) model and a Stochastic Gradient Descent (SGD) model [13]. Identifying some common vulnerabilities identified by traditional methods today, they attempt to use each of the machine learning models to identify eight of the most prominently recognized traditional vulnerability types, including arithmetic, reentrancy, denial of service, unchecked low level calls, access control, bad randomness, front running, and denial of service. As with N.Lesimple’s paper [21], the input to their model uses an AST structure, allowing the model to gather line by line information about the smart contract code. The labels for the vulnerabilities were identified using traditional methods. The paper notes high accuracy, precision and recall, for four of the eight vulnerabilities. The other four did not have enough samples in the dataset, and the corresponding results were recognized as inconclusive. As with the N. Lesimple paper, the test set was created from results from using traditional methods, indicating that the authors were unable to illustrate how the KNN model differed from traditional techniques.

Wang et al. [49] proposed their own model, ContractWard, to detect vulnerabilities in smart contracts utilizing bigram features from simplified operation codes of smart contracts. They collected a dataset of 49,502 smart contracts from Etherscan Website, where each contract -verified before September 2018- contained six possible vulnerabilities. call

stack depth attack, timestamp dependency, and re-entrancy vulnerability. Wang et al. convert the source code of each smart contract to opcodes. On average, a smart contract contains 4364 opcode elements with 100 types of opcodes in total. After the simplification process, there were only 50 opcode types left. Due to that reason, the authors wrapped opcodes with similar functionalities in a same category, and ultimately simplified features in the dataset. Later, they used n-gram technique (sliding window of binary-byte size) to track relations of each opcodes, since they assume that the operations have higher relation with its neighbors. Oyente [30] was used to assign multi label to each contract. After the labeling process, the researchers encountered class-imbalance problem, due to rarity of some vulnerabilities. The training process adopted 5 candidate ML models: eXtreme Gradient Boosting (XGBoost), Adaptive Boosting (AdaBoost), Random Forest (RF), Support Vector Machine (SVM), and k-Nearest Neighbour (KNN). The XGBoost model showed a robust performance by achieving over 96% F1, Micro-F1, and Macro-F1score.

In 2022, Yuqi Fan et al Smart contracts on blockchains have received increasing attention due to the decentralized, transparent, and immutable characteristics of blockchain. However, smart contracts are prone to security problems caused by critical vulnerabilities, which can lead to huge economic losses. Therefore, it is urgent to provide strong and robust security assurance for smart contracts. Most existing studies on smart contract vulnerability detection methods take heavy reliance on experts-defined rules, which are extremely time-consuming and labor-demanding. Moreover, the manually-set rules are limited to specific tasks and subject to errors. Although some studies explore the use of deep learning methods, they fail to represent both semantics and structural information. In this paper, we propose a novel model, Dual Attention Graph Convolutional Network (DA-GCN), to detect vulnerabilities in smart contracts on blockchains. Both control flow graph and opcode sequence extracted from smart contract bytecodes are fed into the feature extractor based on graph convolutional network and self-attention mechanism. Model

DA-GCN then uses control flow level attention to focus on the more important nodes in the control flow graph and suppress useless information. Finally, a multi layer perceptron is used to identify whether the smart contract is vulnerable. Experimental results on the real-world smart contract data set containing two vulnerabilities of reentrancy and timestamp dependency demonstrate that our proposed model DA-GCN can effectively improve the performance of smart contract vulnerability detection.

3.4 SLITHER-SIMIL

Parts of this section have been published in another piece [38] written by the same author of this dissertation and have been used here with permission.

The efforts of security auditing companies like Trail of Bits, Inc. with regard to automating smart contract security assessments has included works on an addition to an already prominent static analysis tool, Slither [13], to better help developers and researchers in their process of auditing smart contracts.

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

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

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

security assessments to:

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

SLITHER-SIMIL [39], the statistical addition to Slither, is a code similarity measurement tool that uses state-of-the-art machine learning to detect similar Solidity functions. When it began as an experiment last year under the codename *critic-pred*, it was used to vectorize Solidity source code snippets and measure the similarity between them. Last year, we took it to the next level and applied it directly to vulnerable code.

SLITHER-SIMIL currently uses its own representation of Solidity code, as introduced by [13], namely SlithIR. SlithIR (Slither Intermediate Representation), to encode Solidity snippets at the granularity level of functions. We thought function-level analysis was a good place to start our research since it's not too coarse (like the file level) and not too detailed (like the statement or line level).

As introduced by Feist et al, SlithIR was developed as an intermediate representation (IR) language with slither in mind to leverage it and represent Solidity code for further analysis. Every smart contract written in solidity can be decomposed into a control flow graph and in that graph, each node can contain up to a single Solidity expression, which is converted to a set of SlithIR instructions. This representation makes implementing analyses easier, without losing the critical semantic information contained in the Solidity source code. [13] SlithIR has a database of about 40 instruction expressions. It has no internal control flow representation and relies on Slither's control-flow graph structure (SlithIR code is associated with each node in the graph). The complete descriptions is available at [34].

In the process workflow of Slither-simil, we first manually collected vulnerabilities from the previous archived security assessments and transferred them to a vulnerability

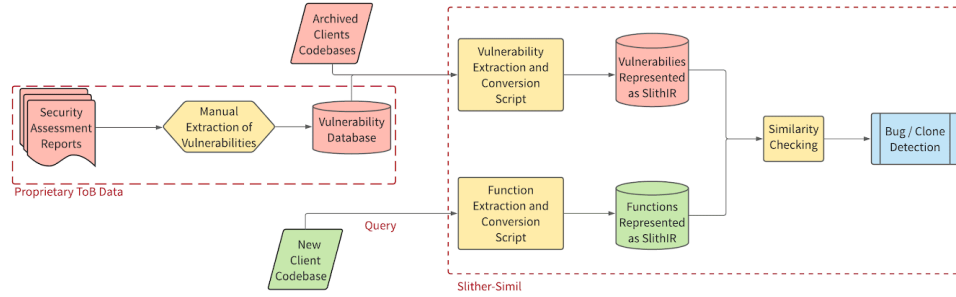


Figure 3: A high-level view of the process workflow of Slither-simil.

```

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

```

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

database. Note that these are the vulnerabilities auditors had to find with no automation.

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

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

First, we converted every statement or expression into its SlithIR correspondent, then


```

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

```

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

```

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

```

Listing 3.3: Normalized SlithIR tokens of the previous expressions.

tokenized the SlithIR sub-expressions and further normalized them so more similar matches would occur despite superficial differences between the tokens of this function and the vulnerability database.

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

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

Comparing the statements in the two functions, we can easily see that they both contain, in the same order, a binary comparison operation (`>=` and `<=`), the same type of operand

comparison, and another similar assignment operation with an internal call statement and an instance of returning a “true” value.

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

Research on automatic vulnerability discovery in Solidity has taken off in the past two years, and tools like Vulcan [43] and SmartEmbed [?], which use ML approaches to discovering vulnerabilities in smart contracts, are showing gradually more and more promising results, with less false positives in specific vulnerability reports.

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

Much of the academic research of the past five years has focused on taking ML concepts (usually from the field of natural language processing) and using them in a development or code analysis context, typically referred to as code intelligence. Based on previous, related work in this research area, we aim to bridge the semantic gap between the performance of a human auditor and an ML detection system to discover vulnerabilities, thus complementing the work of Trail of Bits human auditors with automated approaches (i.e., Machine Programming, or MP [16]).

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

Archiving previous client data was a job in itself since we had to deal with the different solc versions to compile each project separately. For someone with limited experience in

that area this was a challenge, and I learned a lot along the way. (The most important takeaway of my summer internship is that if you're doing machine learning, you will not realize how major a bottleneck the data collection and cleaning phases are unless you have to do them.)

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

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

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

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

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

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

for queries, later training, and testing newer models.

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

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

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

3.5 Concluding Remarks

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

Chapter 4

ETHERBASE: Improving Reproducibility in Smart Contract Research

4.1 Introductory Remarks

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

of 19,366 smart contracts. [29], [41]

Comparing and reproducing such research is not an effortless process. [41] Most of the datasets used to test and benchmark those very same tools proposed in the research literature are not publicly or easily available to the research community. This makes reproduction efforts immensely hard and time-intensive to carry out.

The current approach to compare one's tools / methodologies of evaluating the security of smart contracts with that of another researcher / toolset, is to make do with whatever out-of-date incomprehensive and unrepresentative dataset they have at their disposal, a very timely and inefficient process. In other cases, the researchers need to start from scratch and create their own datasets, a non-trivial and slow process. What makes it worse is the data bias, which can be introduced in a dataset in different phases of data acquisition and data cleaning. This can easily escalate to become a threat to validity in the research. [41]

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

In summary, we make the following contributions with ETHERBASE.

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

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

4.2 Related Work

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

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

which is good for initial benchmarkings, but it only the source code of the contracts are available and does not provide any bytecode for them.

Ren et al. [41] created a benchmark suite that integrates annotated and unlabelled raw smart contracts from a variety of sources such as Etherscan [12], SolidiFI repository, CVE Library, and Smart Contract Weakness (SWC) Registry. They collected 45,622 real-world diversified Ethereum smart contracts and proposed a systematic evaluation process and performed extensive experiments. Their labeled dataset has 350 manually generated contracts.

Many of the authors of the tools proposed in this practice leverage unlabelled datasets to evaluate the performance of their toolset. The publicly available datasets are either not appropriately annotated or a very small subset of them is labelled with the vulnerabilities's identification properties. This is because the number of the contracts -most of them being clones of other smart contracts- is so huge, they cannot be all manually annotated. To the best of our knowledge, there has been only one annotated dataset, realeased publicly, from Yashavant et al. [54] We strive to build upon that work and make facilitate the processes of reproduction in the smart contract research community. Our dataset has the advantage of updating regularly, and being more comprehensive with the more updated Solidity versions of different smart contracts available in the dataset.

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

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

In 2022, Yashavant et al. [54]

The smart contract source codes collected in GitHub repositories associated with the previously published literature do not directly reference smart contracts deployed on the

blockchain through an Ethereum address; [37] this makes it hard to determine if the identified smart contracts have been tested or used on the Ethereum blockchain or not. GitHub repositories of the available datasets do not implement a search engine or query capability to filter smart contracts based on particular metrics or parameters, such as the ETH value or the number of internal transactions for a smart contract.

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

As a user of the Website Etherscan, you can easily search the Ethereum blockchain for any specific smart contract given the availability of its address but when it comes to downloading the data for that smart contract or any other batch of smart contracts, using Etherscan has its limits: [37]: Smart contracts' data is massive, based on the estimation from [37], and the daily limits on the API's provided by Etherscan make retrieving that data even harder. The API provided by Etherscan does not allow the users to obtain a list of the addresses of their desired smart contracts. The API calls currently available only allow navigation at the block-level. A researcher cannot easily explore the source code for their collected smart contracts. First, they would have to inspect any block on-chain and then search for any transactions that have a receive/send address associated with that specific smart contract.

4.3 Methodology

We specifically designed ETHERBASE to not provide the end-users with a dump of contracts and their corresponding features in a vague file hierarchy. We have a service offering the ability to filter and analyze smart contracts and a dataset of smart contracts with interesting

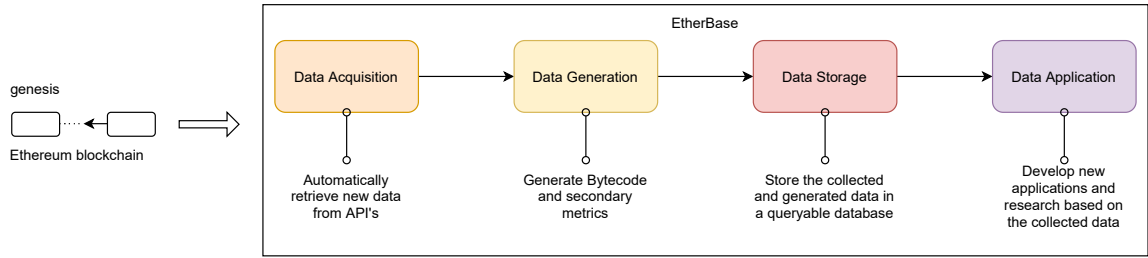


Figure 4: EtherBase Workflow

features to conduct empirical research on, according to metrics like Pragma version, ETH value, etc. To fulfill its purpose, ETHERBASE is designed to perform four primary automatic operations on the data:

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

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

4.3.1 Data Acquisition

The current dataset corresponds to the contracts collected from Etherscan, the most used service for researchers trying to collect smart contracts from the Ethereum blockchain. Every contract stored on Etherscan's database is indexed by its own corresponding addresses. In order to collect the contracts, we retrieve the addresses for every contract which has more than one transaction through Google BigQuery, like the process done by [41]. Using Google BigQuery query request service, we obtain 1,712,347 distinct contract addresses that have more than one transactions associated with them.

4.3.2 Data Generation

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

For the rest of the analysis of the collected contracts, we only get to work on contracts with available source-code. We adopt the method used in the work of [10] to remove the duplicates smart contracts, that is checking the MD5 checksums of each of the two source files in the collected dataset to see if they are the same and after removing the whitespace among the lines of code. After the process of deduplication is done, we get down to 48,622 smart contracts. For this paper and as of now, we have released 5,000 smart contracts for the tool comparison purposes. There exist many metrics that we have access to, can calculate, and add to ETHERBASE, but not a lot of research has been conducted

on the applicability of these many different types of metrics to empirical research on the Ethereum blockchain or how much it appeals to the researchers active in this field. In the following, we describe the initial set of the metrics we selected to include in EtherBase in the form of a table, to be followed by more, after more discussion and research on their applicability to research on smart contracts.

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

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

Table 1: Primary Metrics on Smart Contracts

Name	Description
Pragma	The <code>pragma</code> keyword is used to enable certain compiler features or checks. [1]
Contract Address	Unique 20-byte address, used as the main index to distinguish smart contracts from each other.
Creator Address	Indicates the address of the deeployer of the smart contract.
Source Code	Source code of the smart contract, specific to the programming language Solidity.
Bytecode (bin)	.
Bytcode (bin-runtime)	.
ABI	The content of the application binary interface for each contract.
Block Number	The length of the blockchain in blocks.
ETH Value	The value of each smart contract in therms f=of the ETH thay hold.
Transaction Count	Number of internal transactions from eacch smart contract.

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

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

- `Contract Address`: In Ethereum, the state is made up of objects called "accounts", with each account having a 20-byte address. Contract address is the main key in EtherBase for distinguishing contracts from each other.
- `Creator Address`: The contract address is usually given when a contract is deployed to the Ethereum Blockchain. The address comes from the creator's address, where the contract has been initially deployed from, alongside the number of transactions sent from that address (the "nonce"). A creator's address can be helpful in analyzing the clone ratio and
- `Source Code`: The source code of each Ethereum smart contract is written in Solidity and helps researchers do all sorts of analysis on the smart contracts.
- `Bytecode (bin)`: The regular `bin` output is the code placed on the blockchain plus the code needed to get this code placed on the blockchain, the code of the constructor.
- `Bytecode (bin-runtime)`: `bin-runtime` is the code that is actually placed on the blockchain.
- `ABI`: ABI stands for application binary interface. It's basically how you can encode Solidity contract calls for the EVM and, backwards, how to read the data out of transactions.

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

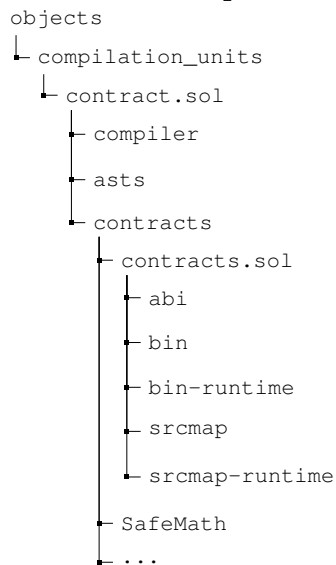
4.3.3 Data Storage

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

Figure 2 shows the directory structure of the collected data. The first leaf in the directory `Contracts` corresponds to the



The `artifact.zip` file contains



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

4.3.4 Data Application

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

- **Smartcheck:** Smartcheck [46] is an extensible static analysis tool written in Java. It detects vulnerabilities and other code issues in Ethereum smart contracts. It locates vulnerabilities by searching for pre-defined patterns in a transformed version of the Solidity source code of the contract.
- **Mythril:** Mythril is another frequently used static analyzer in the form of CLI tool developed in Python that does security analysis of Ethereum smart contracts.
- **Slither:** Slither [13] is a static analyser for analyzing Ethereum smart contracts before deploying them and evaluating them in runtime.

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

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

Table 2: Supported Vulnerabilities

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

- **Unchecked Ether:** also known as silent failing sends, which can lead to unexpected behavior if return values are not handled properly, with SWCRegistry ID SWC-104. [20]

Like the work done by [54], we also leverage the methodology given in the paper by Ren et al. [40] to detect the selected vulnerabilities in smart contracts.

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

Based on the proposal from [54], the following explains the step-by-step methodology concerning the majority voting mechanism as mentioned earlier:

1. Collect the output of the selected static analysis tools for further analysis as the initial step, per vulnerability and identify the LOC in which the vulnerability happens on.
2. With regard to each vulnerability, if different tools show different locations, we

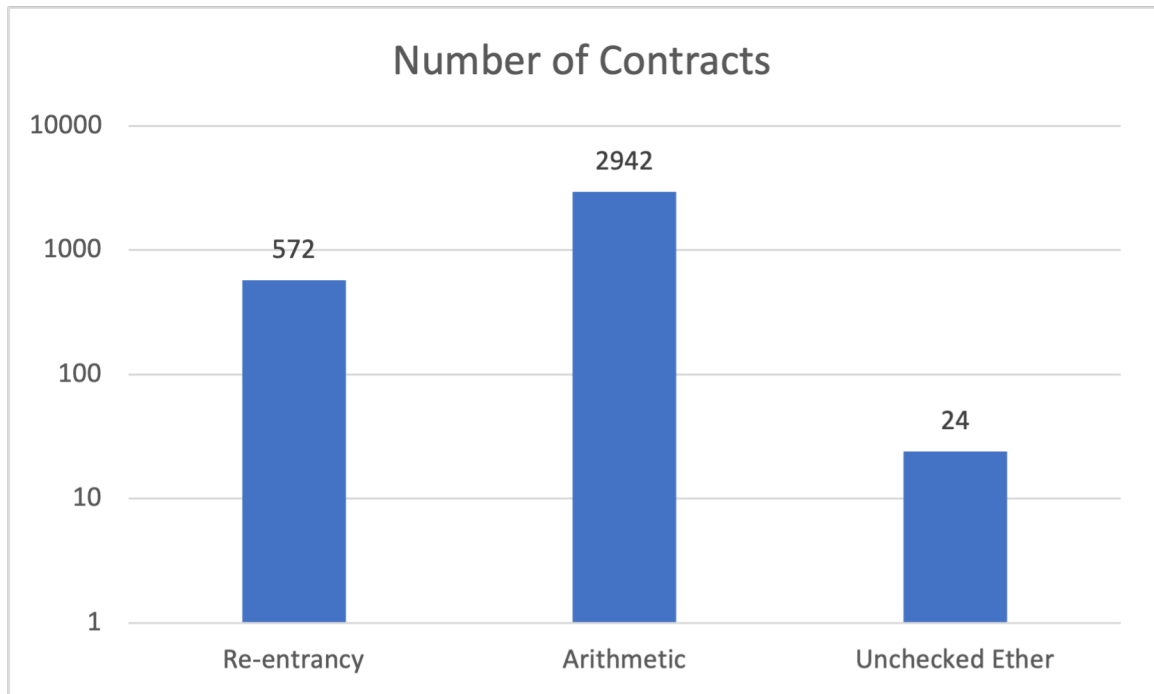


Figure 5: No. of Contracts containing vulnerabilities (log-scale)

should not consider those vulnerabilities the same. We consider two warnings -of any degree of importance generated by a tool- as the same only if that vulnerability's name / ID and LOC location match for all of the different tools being benchmarked.

3. The current methodology being used determines the presence of a vulnerability on a line of code if more than 50% of the tools (2 out of 3 in this experiment scenario) confirm the presence of that vulnerability at that exact location / line of code.

Figure 5 shows the number of smart contracts that contain at least one vulnerability. For instance, consider a specific targeted vulnerability. The literature tells us that all of the three static analyzers in the benchmark support the detection of this vulnerability. [54] notes that they only say the contract at hand contains that vulnerability only if an agreed upon threshold of the tools report that it is present at the same location, based on the majority voting system proposed by them. We take on the same system as well for determining whether a vulnerability is present or not.

4.4 Concluding Remarks

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

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

In addition, our current method is restricted by the rate limit imposed by the API's. For example, Etherscan has a restriction on the daily frequency of queries to its API (5 per day). [54] states this as a serious issue which we would like to solve as well.

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

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

Every tool which was selected for this evaluation is not a complete / sound one, as [54] notices this as well. There are always many false positive / negatives results in an audit report generated by a static analyser. Nevertheless, we utilized the mechanism of majority voting suggested by [54] to determine the presence or lack thereof a vulnerability in a smart contract. We should, however, be wary of the generated false positive results as too many of them will lead to increasing inaccuracies in the released dataset. Such issues can be

overcome by adding more tools to the benchmark process or have some auditors manually review the discovered potential vulnerabilities.

Our competitive advantage in comparison to the work done by [54] is that ETHERBASE gets updated regularly, is more comprehensive with regards to the various versions of smart contracts it contains, and that it leverages offline powerful compilation tools to retrieve more metadata about the collected smart contracts, instead of going through the time consuming process of validating each collected contract with online services separately and through manual development of data collection pipelines.

Chapter 5

Conclusion and Future Work

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

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

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

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

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

Bibliography

- [1] Pragmas, 2021.
- [2] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang. scompile: Critical path identification and analysis for smart contracts. In *International Conference on Formal Engineering Methods*, pages 286–304. Springer, 2019.
- [3] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering*, 2021.
- [4] Consensys. <https://swcregistry.io>.
- [5] ConsenSys. Mythril, 2022.
- [6] P. Daian. Analysis of the dao exploit., 2022.
- [7] A. Day and E. Medvedev. Ethereum in bigquery: a public dataset for smart contract analytics. *Google Cloud Blog*, 2019.
- [8] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.
- [9] T. Durieux, J. a. F. Ferreira, R. Abreu, and P. Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20*, page 530–541, New York, NY, USA, 2020. Association for Computing Machinery.
- [10] T. Durieux, J. a. F. Ferreira, R. Abreu, and P. Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20*, page 530–541, New York, NY, USA, 2020. Association for Computing Machinery.
- [11] W. Ethereum. Ethereum whitepaper. *Ethereum*. URL: <https://ethereum.org> [accessed 2020-07-07], 2014.
- [12] Etherscan. The ethereum blockchain explorer, 2022.

- [13] 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.
- [14] 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.
- [15] C. Ferreira Torres. From smart to secure contracts: Automated security assessment and improvement of ethereum smart contracts. 2022.
- [16] J. Gottschlich, A. Solar-Lezama, N. Tatbul, M. Carbin, M. Rinard, R. Barzilay, S. Amarasinghe, J. B. Tenenbaum, and T. Mattson. The three pillars of machine programming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 69–80, 2018.
- [17] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186. IEEE, 2019.
- [18] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.
- [19] I. Grishchenko, M. Maffei, and C. Schneidewind. *A Semantic Framework for the Security Analysis of Ethereum Smart Contracts*, pages 243–269. 04 2018.
- [20] N. Group. <https://www.dasp.co/>.
- [21] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 531–548, 2019.
- [22] 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.
- [23] S. Hwang and S. Ryu. Gap between theory and practice: An empirical study of security patches in solidity. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 542–553, 2020.
- [24] B. Jiang, Y. Liu, and W. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269, 2018.

- [25] B. Johnson. A study on improving static analysis tools: Why are we not using them? *Proceedings - International Conference on Software Engineering*, pages 1607–1609, 06 2012.
- [26] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018.
- [27] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68. IEEE, 2018.
- [28] H. Liu, Z. Yang, Y. Jiang, W. Zhao, and J. Sun. Enabling clone detection for ethereum via smart contract birthmarks. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 105–115. IEEE, 2019.
- [29] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery.
- [30] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery.
- [31] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.
- [32] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020.
- [33] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC '18*, page 653–663, New York, NY, USA, 2018. Association for Computing Machinery.
- [34] T. of Bits: Slithir documentation. <https://github.com/crytic/slither/wiki/SlithIR>, 2018.
- [35] Parity. <https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/>.
- [36] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*, pages 1661–1677. IEEE, 2020.

- [37] 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.
- [38] S. Pilehchiha. Efficient audits with machine learning and slither-simil, Oct 2020.
- [39] 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.
- [40] 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.
- [41] 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.
- [42] S. So, M. Lee, J. Park, H. Lee, and H. Oh. Verismart: A highly precise safety verifier for ethereum smart contracts. pages 1678–1694, 05 2020.
- [43] S. Srikant. *Vulcan: classifying vulnerabilities in solidity smart contracts using dependency-based deep program representations*. PhD thesis, Massachusetts Institute of Technology, 2020.
- [44] 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.
- [45] N. Szabo. Formalizing and securing relationships on public networks. *First monday*, 1997.
- [46] 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.
- [47] 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.
- [48] 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.

- [49] Y. Wang. Automated market makers for decentralized finance (defi). *arXiv preprint arXiv:2009.01676*, 2020.
- [50] 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.
- [51] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.
- [52] V. Wüstholtz and M. Christakis. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409, 2020.
- [53] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1029–1040, 2020.
- [54] 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.