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 lof of effort to develop automated analysis tools [27] to be able to identify vulnerabilities vulnerabilities in smart contracts. [27] 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. [35] 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. [31]

Famous attacks that have caused significant financial losses and prompted the research community to work to prevent similar occurrences include The DAO hack [7] and the Parity wallet issue [37]. 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 reprodyction 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tools and hope for access to the same datasets used in the original ressearch / work or make do with whatever out-of-date incomprehensive and unrepresentative dataset they have at their disposal, a very timely and inefficient process. [31]

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. [42]

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 acquisiton and cleaning is not available due to private IP reasons, and only the collected data and the database is accessibel 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 intetrnal mechanisms.

2. We implement ETHERBASEand make its pipelines open-source. It obtains historical data and facilitates benchamrking and reproduction in research and development for new toolsets. It is more up-to-date than existing datasets, gets sutomatically 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 mtoivations behind developing such tools, what they have achieved so far, and our own efforts in developing and working on the tool Slither-similas 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 acquanited 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 [17]): 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 vulnreabilities out in the wild. We will discuss Solidity-written source code and bytecode of smart contracts, and explain each vulnrability according to the DASP 10 classification. [22]

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

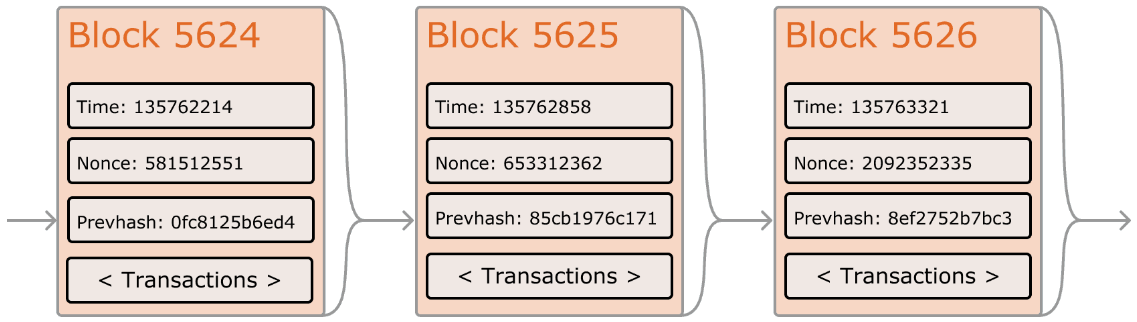

Figure 1: Ethereum blockchain structure.

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. [17] 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 concensus protocol, for example. The solution to a series of cryptographic puzzles are used in the PoW mechanism to prove the credibility of the data being written on the blockchain, using such mechanism as their consensus protocol. The puzzle is usually a computationally hard but easily verifiable mathematical problem. When a node creates a block, it must resolve a PoW puzzle and spend computing power to achieve so. The nodes compete o=with each other over this objective function and the node with the most computing power usually succeeds ind doing so. After the PoW puzzle is resolved, it will be broadcasted to other nodes, so as to achieve the purpose of consensus and append a new block to the blockchain. It acts as a “proof” that a node has done “work” by spending its computational resources. This process is known as

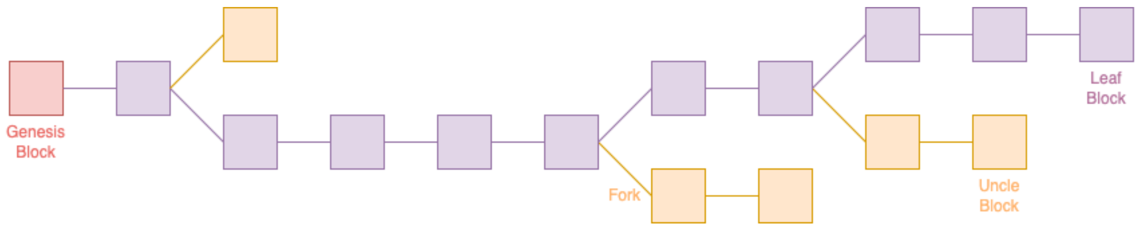

Figure 2: An illustration of Ethereum's GHOST protocol.

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

2.1.1 Accounts

The Ethereum state is consisted of many small objects naemd as accounts, where each account has a 20-byte address and state transitions to be able to interact with the other accounts on-chain. An address on the Ethereum blockchain is a 160-bit identifier that is used to identify any account. The world state is a mapping between addresses and account states. [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) [13]. Inside of an Ethereum account is composed of four fields: nonce, ether balance, contract code hash, and storage root, explained as follows:

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

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

2.1.2 Transactions

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

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

2.1.3 Blocks

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

2.1.4 Ethereum Virtual Machine

The formal definition of the EVM is specified in the Ethereum Yellow Paper. [51] The Ethereum Virtual Machine (EVM) at the heart of the Ethereum blockchain is a VM (virtual machine), with a stack-based architecture with 256-bit word sizes, supporting Turing-complete programming languages. EVM handles the computation side for Ethereum and comes with a set of instructions (namely, opcodes). Thus, a smart contract from a low-level point of view is a series of opcode instructions which EVM can read and compute and execute the logic of that smart contract. The EVM is also responsible for handling the estimation and calculation of gas consumption for transactions in smart contracts.

2.2 Smart Contracts

The concept of smart contracts - programs running on the EVM - has been first introduced by Nick Szabo in one of his works in 1997. [45] They provide a framework that allow any sound program to be executed in an autonomous, distributed, and trusted manner. [34] The main programming langauge currently in use for the development of smart contracts is Solidity, although Vyper is gaining gradual traction as well.

2.2.1 Vulnerabilities

Solidity, like any other programming langauge in history, is prone to all kinds of vulnre-abilities. 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 aassets and cryptocurrencies. Attcking such contracts successfully can result in enormous financial losses. Some of these vulnerabilities like another programming language arise from the human factor invlved in development of the smart contracts, and some specific to the blockchain data structuresa and how they and their components function and interact with each other. And these are only vulnerabilities at the scope of smart contracts we are focusing on. Vulnerabilities can arise with regards to core infrastructure of the blockchains handling smart contracts as well. In this section, we go over 9 of the more discussed vulnerabilities in Solidity and Ethereum according to [22] to get a better sense of what threat surface the deveopers and researchers developing analysis tools face:

Re-entrancy Often called as the most famous Ethereum vulnerability, the re-entrancy attack has been a great example at showing the risks of *"Code is Law"* and the importance of smart contract security, historically. The DAO hack [9] is one of the most famous real worl examples of the re-entrancy hack. The re-entrancy attack can also be counted as a type of denial-of-service (DoS) attack, where a malicious actor can cause a program to infinitely loop and consume CPU cycles and in the case of smart contracts, drain a wallet of its ETH's. Reentrancy occurs when external contract calls are allowed to make new calls to the calling contract before the initial execution of that call is complete. For a function, this means that the contract state may change in the middle of its execution as a result of a call to an untrusted contract. [22]

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

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

Unhandled Exception Also known as unchecked-send, this vulnerability can cause unwanted outcomes when the smart contract is executed, due to the fact that some low level calls in Solidity like `call()` and `delegatecall()` can return a boolean value set to the value False and lt the execution flow resume if an error happens mid-execution. This is not ideal since it means that the execution of the smart contract has not been reversed and has successfully been completed, but with wrong or undesirable outcomes. Thus, the return values of such low-level calls should always be checked and the develoeprs need to make sure that such exceptions are handled appropriately during execution.

Frontrunning The frontrunning vulnerability is one of the more famous ones in the list, also known as Transaction Ordering Dependance (TOD). Explotation of this vulnerability happens when malaicious 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 tarnsaction gets mined before that of victim's due to it giving having more aligned incentives with he 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]
- Supperession attack, where an attacker makes multiple transactions with higehr gas prices that the victim transactions to prevent them from being mined in the same block.

Bad Randomness Also known as *nothing is secret*, [22] 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 [5].

Time Manipulation This vulnerability is also known as *timestamp dependence* [22]. In Solidity, a block's timestamp is often used to generate psuedi-random numebrs and in other times, it can be leveraged for smart contracts to conduct time-intesive oeprations, like unlocking funds at a specific time. A malicious miner of a block can manipulate the timestamp reported while generating the block and sue this vulenrability 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 [16].

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 chrnological 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. [42] provide three broad categories of tools based on their utilized methodology, namely Static Analysis, Dynamic Fuzzing, and Symbolic Execution.

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

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

Tools that leverage the static analysis methods, typicallu conevrt the raw form of the input program into an intermediate representation and then perform a series of analyses on those representations based on a pre-defined database of vulnerability patterns and filter out the sucpicious snippets of the input program. Slither [15], Securify [47], and SmartCheck [47] are categorised as instances of static analyzers.

Fuzzing [4] is a technique for finding software bugs that involves creating erroneous input data and watching the target program's unusual output while it runs. It allows developers to generate exploits for security-critical programs and ensure a uniform standard of quality through prepared tests, but does not narrow down the causes of detected bugs. When applied to smart contracts, a fuzzing engine will first try to generate initial seeds to form executable transactions. With reference to the feedback of test results, it will dynamically adjust the generated data to explore as much smart contract state space as possible. Finally, it will analyze the status of each transaction based on the finite state machine to detect whether there is an attackable threat. ContractFuzzer [26], ReGuard [29], and sFuzz [34] are among the modt cited smart contract fuzzers.

Symbolic execution is a technique for finding software bugs that involves creating symbolic values and watching the target program's unusual output while it runs. When using symbolic execution to analyze a program, it will use symbolic values as input instead of the specific values during the execution. Tools leveraging this technique explore a state space with a high degree of semantic awareness. [3] 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. [42] The symbolic execution tools usually build a control flow graph initially which is based on the Solidity bytecode of the smart contracts being tested. Afterwards, tehy implement constraints based on the characteristics of smart contract vulnerabilities, and finally use the constraint solver to generate satisfying test cases. Oyente [32], Mythril [6], and Manticore [33] support symbolic execution for smart contracts.

3.3 Deep Learning in Smart Contracts

In this section, we will go over some of the literature focusing their efforts on replacing the existing tools' capabilities explained in the previous section with amchine 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 [19]. 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 [20]. 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 [19].

Liao et al. have adopted a sequence learning approach to detect smart contract security threats [20]. 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 [21]. 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 [21]. 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. [24] 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) [24]. 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 transac- tions. 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 loses. 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 [8], 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 [8]. 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 [25]. 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. aThe proposed algorithms (CGE) achieved 89% accuracy on finding re-entrancy and timestamp dependence type of vulnerabilities, and 83% accuracy on detecting infinite loop vulnerability [25].

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. [10] 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 [19], 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 [19]. 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 [21] 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 [19]. As a result, the word embedding process has been enhanced through the use of self-attention. When compared to the vulnerability detection performance of [21], they have both used a CNN but [21] 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 [26] 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 [15]. 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 [23], 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 [32] 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 [39] 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 [15], to better help developers and researchers in their process of auditing dmart 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 [40], 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 crytic-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-SIMILcurrently uses its own representation of Solidity code, as introduced by [15], 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. [15] 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 [36].

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

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.

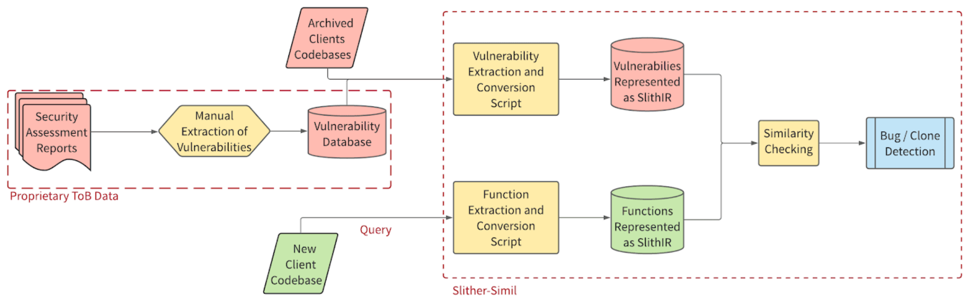
Archived Clients Codebases
Security Assessment Reports
Manual Extraction of Vulnerabilities
Vulnerability Database
Proprietary ToB Data
Vulnerability Extraction and Conversion Script
Vulnerabilies Represented as SlithIR
Similarity Checking
Bug / Clone Detection
Function Extraction and Conversion Script
Functions Represented as SlithIR
Query
New Client Codebase
Slither-Simil

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

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

```

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

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

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

```
Function TurtleToken.transferFrom(address,address,uint256)
    (*)


Solidity Expression: require(bool)(_value <= allowance[_from
    ][msg.sender])
SlithIR:
         REF_10(mapping(address => uint256)) ->    allowance
    [_from]
         REF_11(uint256) -> REF_10[msg.sender]
         TMP_16(bool) = _value <= REF_11
         TMP_17 = SOLIDITY_CALL require(bool)(TMP_16)


Solidity Expression: allowance[_from][msg.sender] -= _value
SlithIR:
         REF_12(mapping(address => uint256)) -> allowance[
   _from]
         REF_13(uint256) -> REF_12[msg.sender]
         REF_13(-> allowance) = REF_13 - _value


Solidity Expression: _transfer(_from,_to,_value)
SlithIR:
         INTERNAL_CALL,      TurtleToken._transfer(address,
   address,uint256)(_from,_to,_value)


Solidity Expression: true
SlithIR:
         RETURN True

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

```
type_conversion(uint256)

binary(**)

binary(*)

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

index(uint256)

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

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

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

...

```
Listing 3.3: Normalized SlithIR tokens of the previous expressions.

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 [18]).

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 facilitied 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-learnined-based methods to facilitate the process of discovering and mitigating vulnerabilities in smart contracts, compraed 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 modelsl. In the next chapter we will introduce ETHERBASEand how the development of SLITHER-SIMIL has extended into the development of ETHERBASEas 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-exeucting 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 frameorks [27] 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. [35] 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. [31], [42]

Comparing and reproducing such research is not an effortless process. [42] 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 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. [42]

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 acquisiton and cleaning is not available due to private IP reasons, and only the collected data and the database is accessibel 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 intetrnal mechanisms.

2. We implement ETHERBASEand make its pipelines open-source. It obtains historical data and facilitates benchamrking and reproduction in research and development for new toolsets. It is more up-to-date than existing datasets, gets sutomatically 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 laballed dataset of Ethereum smart contracts with vulnrabilities.

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. [30] In this section we present and discuss references and information for some of the more prominent publicly available smart contract benchamrk datasets which were identified in our studies.

SmartBugs [11] makes the top of the list as one of the most used benchmarks in the research space. In this work, Durieux et al. presented an extensible and easy-to-use execution framework for benchmarking different security analysis tools for Ethereum smart contracts. They empirically evaluated 9 analysis tools on a small (< 100)labelled dataset of vulnerabilities. Ren et al. provide a dataset consisting of 47,518 un-annotated smart contracts with the history of at least one executed transaction on the blockchain. For researchers trying to build upon such work, using the unlabelled dataset is a great improvement for both tool development and reproduction efforts, but because of the lack of some degree of ground truth in the dataset, it's not possible to easily apply, for example, supervised amchine learnign 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. [42] created a benchmark suite that integrates annotated and unlabelled raw smart contracts from a variety of sources such as Etherscan [14], SolidiFI repository, CVE Library, and Smart Contract Weakness (SWC) Registry. They collected 45,622 real-world diversified Ethereum smart contracts and proposed a systematic evaluation process and performed extensive experiments. Their labeled dataset has 350 manually generated contracts.

Many of the authors of the tools proposed in this practice leverage unlabelled datasets to evaluate the performance of their toolset. The publicly available datasets are either not appropriately annotated or a very small subset of them is labelled with the vulnerabilities's identification properties. This is because the number of the contracts -most of them being clones of other smart contracts- is so huge, they cannot be all manually annotated. To the best of our knowledge, there has been only one annotated dataset, realeased publicly, from Yashavant et al. [52] 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. [28] published their analysis results for 1,524 smart contracts with no other information or metadata in relation to those contracts.

Luu et al. [32] 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. [52]

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; [38] 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 datat 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: [38]: Smart contracts' data is massive, based on the estimation from [38], 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 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 ETHERBASEto 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

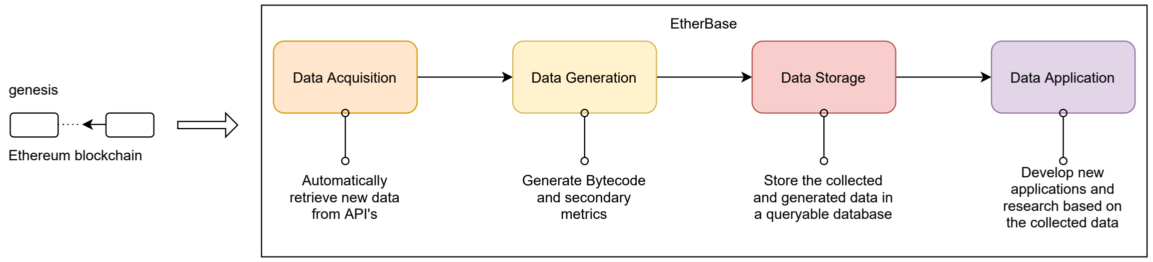
EtherBase
genesis
Ethereum blockchain
Data Acquisition
Data Generation
Data Storage
Data Application
Automatically retrieve new data from API's
Generate Bytecode and secondary metrics
Store the collected and generated data in a queryable database
Develop new applications and research based on the collected data

Figure 4: EtherBase Worflow

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

1. **Data Acquisiton**: 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 collected 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 [42]. 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 Cyitic-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 busines, 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 compatiable version of solc to compile the input smart contracts. They way this works under the hood is that crytic-compile compiles an input smart contract and outputs a compilation untit in the standard solc output format, written in a json file, alongside the source code and other metadata. Unlike the other proposed datasets and tools discussed in Section 4.2, EtherBase targets the core problem of the lack of reproducibility in the research literature. Discussions surrounding what types of secondary metrics to retrieve from the Ethereum blockchain or third-party services will be explored further.

For the rest of the analysis of the collected contracts, we only get to work on contracts with available source-code. We adopt the method used in the work of [12] to remove the duplicates smart contracts, 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 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 `pragma` keyword is used to enable certain compiler features or checks. [2]
`Contract Address`	Unique 20-byte address, used as the main index to distinguish smart contracts from each other.
`Creator Address`	Indicates the address of the deeployer of the smart contract.
`Source Code`	Source code of the smart contract, specific to the programming language Solidity.
`Bytecode (bin)`	.
`Bytcode (bin-runtime)`	.
`ABI`	The content of the application binary interface for each contract.
`Block Number`	The length of the blockchain in blocks.
`ETH Value`	The value of each smart contract in therms f=of the ETH thay hold.
`Transaction Count`	Number of internal transactions from eacch smart contract.

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

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

- `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 sortds 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 smartt contract exists.

- `ETH Value`: The ETH every smart contract hols 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 thast 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 their specific research queries.

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

```
contracts
├─ 0
├─ 1
│  ├─ 0
│  │  ├─ 0
│  │  │  ├─ 0
│  │  │  │  ├─ 0
│  │  │  │  │  └─ 5
│  │  │  │  │     └─ 0x100005bc082d49eefffdc720864984bd7f3f7e5e
│  │  │  │  │        ├─ 0x100005bc082d49eefffdc720864984bd7f3f7e5e-SudEX.sol
│  │  │  │  │        ├─ artifact.zip
│  │  │  │  │        ├─ slither-findings.json
│  │  │  │  │        ├─ slither-findings.md
│  │  │  │  │        └─ slither-findings.txt
│  │  │  │  ├─ ...
│  │  │  │  └─ f
│  │  │  ├─ ...
│  │  │  └─ f
│  │  ├─ ...
│  │  └─ f
│  ├─ ...
│  └─ f
├─ ...
└─ f
```

The `artifact.zip` file contains

```
objects
└─ compilation_units
   └─ contract.sol
      ├─ compiler
      ├─ asts
      └─ contracts
         ├─ contracts.sol
         │  ├─ abi
         │  ├─ bin
         │  ├─ bin-runtime
         │  ├─ srcmap
         │  └─ srcmap-runtime
         ├─ SafeMath
         └─ ...
```

In addition to making the datasets available on GitHub, ETHERBASEalso enjoy a graphical user interface (GUI) in order to allow the less technical end users access and browse through the database. We integrated ETHERBASEwith Apache Superset, a powerful business intelligence tool, which lets you creata 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 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 vulnreability 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 thereum 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 [15] 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 thre vulnerabilities, as explained in Chapter 2, are as follows:

- **Re-entrancy** also known as the recursive call vulnerability, with SWCRegistry ID SWC-107.

- **Arithemtic:** concerning the integer overflows and underflow vulnerabilities in smart contracts, with SWCRegistry ID SWC-101.

Table 2: Supported Vulnerabilities

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

- **Unchecked Ether:** also known as silent failing sends, which can lead to unexpected behavior if return values are not handled properly, with SWCRegistry ID SWC-104. [22]

Like the work done by [52], we also leverage the methodology given in the paper by Ren et al. [41] to detect the selected vulnerabilities in smart contracts.

When using tools like these static analyzers, we face a lot of false positive resultsas a necessaity of the methods those tools employ. Because of that, we cannot rely on one tool only, as projects that rely n=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 [52], 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 vulnreability and identify the LOC in which the vulnerability happens on.

2. With regard to each vulnerability, if different tools show different locations, we

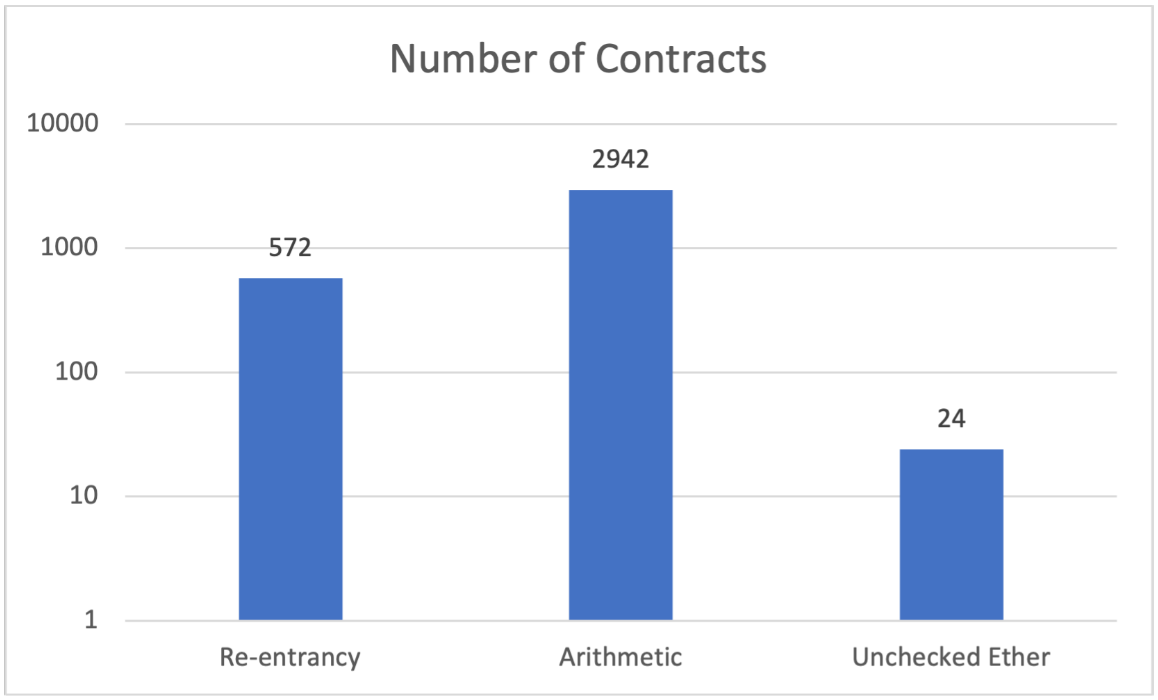

Figure 5: No. of Contracts containing vulnerabilities (log-scale)

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 fo the the three static analyzers in the benchmark support the detection of this vulnerability. [52] 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 wether 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 ETHERBASEwe're already working on, will take full use of an Ethereum full node and instrment 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, Etherescan has a restriction on the daily frequency of queries to its API (5 per day). [52] states this as aserious 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 ETHERBASEfor evaluating correctness and other parameters of their proposed or other toolsets, especially those based on machine learning techniques that need comprhensiver datasets for training, validation, and testing phases. ETHERBASE comprises a diverse and comprehensive set of real-world heterogenuous annotated smart contracts.

Every tool which was selected for this evaluation is not a complete / sound one, as [52] 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 [52] to determine the presence or lacke thereof a vulnerability in a smart contract. We should, however, be wary of the generated false positive rsults 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 benchamrk process or have some auditors manually review the discovered potential vulnerabilities.

Our competitive advantage in comparison to the work done by [52] is that ETHERBASEgets updatedt regularly, is more comprehensive with regards to the various versions of smart contracts it contains, and that it leverages offline powerful compialtion 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.