Chapter 1

Introduction

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

1.1 Motivation

The research community has made much effort to develop automated analysis tools [27] to be able to identify 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 potentially vulnerable. [37] 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. [33]

Famous attacks that have caused significant financial losses and prompted the research community to work to prevent similar occurrences include The DAO hack [9] and the Parity wallet issue [39]. Comparing and reproducing such research is not an effortless process. The datasets used to test and benchmark those tools proposed in the research literature are not publicly available, making reproduction efforts immensely hard to carry out.

If a researcher intends to benchmark their new proposed methodologies, models, or toolsets or experiment with different research ideas and compare their work with the existing work and publications, the best they can do is to directly contact the authors of the previous publications and researchers to have potentially in-time access to the same datasets used in the original research/work or make do with whatever out-of-date incomprehensive and unrepresentative dataset they have at their disposal, a very timely and inefficient process. [33]

In other cases, the researchers must start from scratch and create their datasets, a non-trivial and slow process. What makes it worse is the data bias, which can be introduced in a dataset in different phases of data acquisition and cleaning. The presence of such bias in data can quickly escalate to threaten the research's validity. [44]

1.2 Thesis Statement

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

To summarize, our contributions are as follows:

1. We propose ETHERBASE, a systemic and up-to-date database for Ethereum by exploiting its internal mechanisms.

2. We implement ETHERBASEand make its pipelines open-source. It obtains historical data and facilitates benchmarking and reproduction in research and development for new toolsets. It is more up-to-date than existing datasets and gets automatically

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

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

1.3 Outline and Contributions

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

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

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

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

Chapter 2

Background

This chapter reviews the necessary background knowledge for the reader to be better acquainted with the work conducted within this thesis. It highlights the foundational technical basics of Ethereum blockchain, smart contracts, and the most common vulnerabilities associated with smart contracts. We cover these technicalities in the following order (based on the work of [19]): 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. Afterward, 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. [23]

2.1 Ethereum

Ethereum is a decentralized virtual machine introduced as alternative blockchain technology to Bitcoin in 2014 by [53]. Blockchain are peer-to-peer networks made up of computers/nodes that update for one single global database without necessarily trusting in one

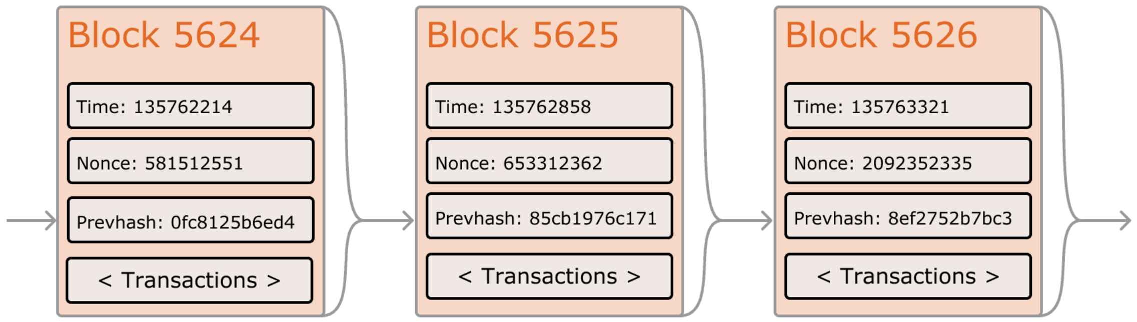

Figure 1: Ethereum blockchain structure.

another in a distributed fashion. It is based on a combination of cryptography, networking, and incentive mechanisms. [52] The database mentioned above effectively serves as a ledger, recording every transaction that each node in the blockchain network makes.

As the second most popular blockchain, the Ethereum blockchain is a transaction-based, cryptographically secure state machine. It takes a series of inputs transitions from its initial state to a new one according to the transition rules defined by those inputs. [19] Like Bitcoin, Ethereum currently depends on the Proof-of-Work (PoW) consensus protocol. Proof-of-Stake (PoS) and PBFT(Practical Byzantine Fault Tolerance) are other forms of a consensus protocol, used by other blockchains per their defining characteristics. The PoW mechanism works as follows: A series of cryptographic puzzles aer intoroduced to the existing nodes on the network and the solutions to those puzzles are utilized as proof that the data being written on the blockchain are legitimate and credible. This gets verified through the concensus 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. Trying to solve the puzzles sooner than any other party, the nodes compete with each other over this objective function, and the node with the most computing power usually succeeds. After a node proves that their solution to a puzzle is correct, it will be broadcasted to other nodes so that all of them can reach concensus and agree upon appending a new block to the blockchain. [30] This acts as a

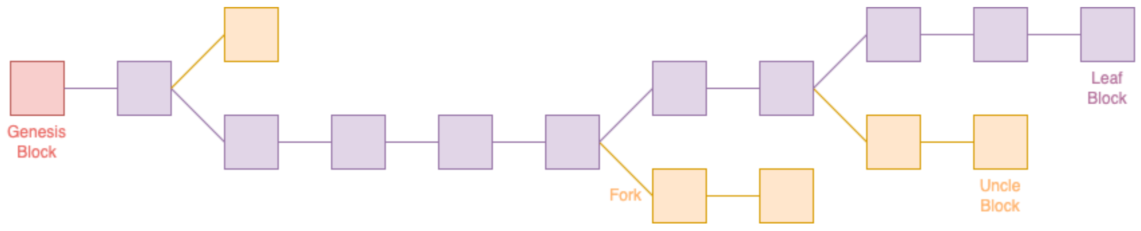

Figure 2: Visualization of GHOST protocol in Ethereum.

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

2.1.1 Accounts

The Ethereum state consists of many small objects named accounts. Each of these accounts are identified with a 20-byte address to interact with the other accounts on-chain. An address on the Ethereum blockchain is a 160-bit identifier used to identify any account. Ethereum supports two types of accounts:

- externally owned (controlled by private keys), known as EOAs, and
- contract accounts (CAs, controlled by their contract code). [14]

To Be Added

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

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

- **Balance:** (Ether) Balance basically records the Ethereum balance of the account, which is the amount of Wei assigned to the address of that account. Wei is the smallest measurement unit of Ether (1 Wei is the equivalent amount of 10-18 ethers).
- **storageRoot:** Each account has its own storage trie as explained above and StorageRoot is the 256-bit hash of the root node of a of that trie.
- **Contract codeHash:** The codeHash of a contract is the Keccak-256 hash value of the code of the account on the EVM.

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

2.1.3 Ethereum Virtual Machine

The formal definition of the EVM is specified in the Ethereum Yellow Paper. [53] The Ethereum Virtual Machine (EVM) at the heart of the Ethereum blockchain is a VM (virtual machine) with a stack-based architecture with 256-bit word sizes, supporting Turing-complete programming languages. EVM handles the computation side for Ethereum and comes with a set of instructions (namely, opcodes). Thus, a smart contract, from a low-level point of view, is a series of opcode instructions that EVM can read and compute and execute the logic of that smart contract. The EVM is also responsible for estimating and calculating gas consumption for transactions in smart contracts.

2.2 Smart Contracts

Nick Szabo introduced smart contracts as a concept - programs running on the EVM - in one of his works in 1997. [47] They provide a framework that allows any sound program to be executed in an autonomous, distributed, and trusted manner. [36] The main programming language currently in use for the development of smart contracts is Solidity, although Vyper is gaining gradual traction as well.

2.2.1 Vulnerabilities

Solidity, like any other programming language in history, is prone to all kinds of vulnerabilities. What makes security vulnerabilities in Solidity so attractive is the fact that the programs written in Solidity are very 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, arising from the human factor involved in the development of the smart contracts, and some are specific to the blockchain data structures and how they and their components function and interact with each other. Furthermore, these are only vulnerabilities within the scope of smart contracts we focus on. Vulnerabilities can arise regarding the blockchains' core infrastructure handling smart contracts. In this section, we go over 9 of the more discussed vulnerabilities in Solidity and Ethereum according to [23] to get a better sense of what threat surface the developers and researchers developing analysis tools face:

Reentrancy Often called the most famous Ethereum vulnerability, the reentrancy attack has been a great example of showing the risks of *"Code is Law"* and the importance of smart contract security historically. The DAO hack [10] is one of the most famous real-world examples of the reentrancy hack. The reentrancy attack can also be counted as a denial-of-service (DoS) attack, where a malicious actor can cause a program to infinitely loop and consume CPU cycles and, in the case of smart contracts, drain a wallet of its ETHs. The reentrancy vulnerability is exploited when external contract calls are allowed to make new calls to the calling contract before the initial execution of that call is complete. [23]

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

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

Unhandled Exception Also known as unchecked-send, this vulnerability can cause unwanted outcomes when the smart contract is executed because some low-level calls in Solidity like `call()` and `delegatecall()` can return a boolean value set to the value False and 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 successfully completed but with wrong or undesirable outcomes. Thus, the return values of such low-level calls should always be checked, and the developers must ensure that such exceptions are handled appropriately during execution.

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

- Displacement attack, where an adversarial party makes a transaction in order to displace the victim user's transaction by having a higher gas price, and thus, the attacker's transaction gets mined before that of the victim's due to it giving having more aligned incentives with he miners' network.
- Insertion attack, in which an adversarial actor makes two transactions, one with a higher gas price than that of the victim and one with a lower gas price, to *sandwich* the victim transaction. [50]
- 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*, [23] this vulnerability happens when smart contracts attempt to generate random, or to be more exact, pseudo-random numbers for any number of reasons. If the smart contract generating the pseudo-random number computes that random number using values that a malicious party can guess, then the attacker can predict the next number that will be generated. Values such as block timestamps or block numbers are generally advised against being used in such mechanisms. They are called hard-to-predict values, but it is better to use an external oracle to generate the random numbers needed [7].

Time Manipulation This vulnerability is also known as *timestamp dependence* [23]. In Solidity, a block's timestamp is often used to generate pseudo-random numbers. In other times, it can be leveraged for smart contracts to conduct time-intensive operations, like unlocking funds at a specific time. A malicious miner of a block can manipulate the timestamp reported while generating the block and use this vulnerability for their profit.

Short Address The short address vulnerability, also known as off-chain issues, results from the Ethereum Virtual Machine accepting arguments with incorrect paddings. Attackers exploiting this vulnerability can craft truncated addresses that clients may encode incorrectly in transactions. Additionally, it has not been exploited in the wild, as mentioned by [18].

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. A smart contract is an event-driven, self-executing, state-based program created using high-level programming languages like Vyper and Solidity. Because of their distinctive features, smart contracts require more careful development than the amount traditional software programs do. First, compared to regular software, smart contracts are more prone to bugs and vulnerabilities. The concept that the smart contracts execute themselves and cannot be stopped after deployment hs been previously coined by an expression described by Lessig [29] as "code is law". This is because transactions of a smart contract always involve digital assets (e.g. various cryptocurrencies or NFTs) of financial nature which can be very profitable if stolen(e.g. The DAO [9]). Therefore, a bug in a smart contract may lead to substantial financial losses. 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. [46] provide the first-ever survey on machine learning methods utilized to discover and mitigate vulnerabilities in smart contracts. In order to set the ground for further development of ML methods on smart contract vulnerability detection, They reviewed many ML-driven intelligent detection mechanisms 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 then the novel deep-learning-based methodologies proposed in the literature over the past few years in chronological order, and we add some of the works missing in [46] as well and update the list. Afterward, we propose our solution, SLITHER-SIMIL, and how it led us to the development of ETHERBASE.

3.2 Traditional Security Analysis Methods in Smart Contracts

Classic software testing technologies applied toward 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. [44] provides three broad categories of tools based on their utilized methodology, namely Static Analysis, Dynamic Fuzzing, and Symbolic Execution.

Using the static analysis method, we can analyze the program at both the source code (high-level) and bytecode (low-level) scopes before conducting any runtime execution. Static analysis-based tools can scan a whole 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 analyzing the structure of the source code beside raw text/source code;
- complementing the generated IR with additional metadata with methods such as control flow, data flow analysis, and symbolic execution.
- vulnerability detection through pattern matching and referencing those patterns to a database containing (vulnerable) patterns. Setting a specific threshold defines wether matched pattern counts as a vulnerability or not.

Tools that leverage the static analysis methods typically convert the raw form of the input program into an intermediate representation and then perform a series of analyses on those representations based on a pre-defined database of vulnerability patterns and filter out the suspicious snippets of the input program. Slither [17], Securify [49], and SmartCheck [49] are categorised as instances of static analyzers.

Fuzzing [6] is a technique for finding software bugs that involve creating erroneous input data and watching the target program's unusual output while it runs. It allows developers to generate exploits for security-critical programs and ensure a uniform standard of quality through prepared tests but does not narrow down the causes of detected bugs. A fuzzing engine will first try to generate initial seeds to form executable transactions when applied to smart contracts. Regarding the feedback on 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 [31], and sFuzz [36] are among the modt cited smart contract fuzzers.

Symbolic execution is a technique for finding software bugs that involve creating symbolic values and watching the target program's unusual output while it runs. When using symbolic execution to analyze a program, it will use symbolic values as input instead of the specific values during the execution. Tools leveraging this technique explore a state space with a high degree of semantic awareness. [5] Symbolic execution can simultaneously explore multiple paths the program can take under different inputs, but it also faces unavoidable problems such as path explosion. [44] The symbolic execution tools usually build a control flow graph based on the Solidity bytecode of the smart contracts being tested. Afterward, they implement constraints based on the characteristics of smart contract vulnerabilities and finally use the constraint solver to generate satisfying test cases. Oyente [34], Mythril [8], and Manticore [35] support symbolic execution for smart contracts.

3.3 Deep Learning in Smart Contracts

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

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

While existing symbolic tools (like Oyente) for assessing vulnerabilities have shown to be effective, Goswami et al. said in 2018 that their execution time grows noticeably with the depth of invocations in a smart contract (cite: Grech2019GigaHorse). They suggested an LSTM neural network model find flaws in ERC-20 smart contracts to create a quicker and more effective replacement for symbolic analysis tools. This paper's preprocessing procedures were remarkably similar to those employed by [22]. A dataset of 165,652 ERC-20 smart contracts was used for training and testing the model, and it contained bytecode data that Maian and Mythril had annotated (statistical code analysis tools). On the testing set, the proposed model had an F1 score of 93.26Additionally, they have contrasted the time performance of their model with that of Maian and Mythril's symbolic analysis tools (static analysis tools). Their suggested model operated on a test set of 5,000 random tokens in 15 seconds, while Maian and Mythril needed 32,476 and 9,475 seconds, respectively. These findings show a similar advancement over symbolic analysis methods to that shown in citegrech2019gigahorse.

To identify security concerns to smart contracts in 2018, Liao et al. used a sequence learning approach (cite: madmax). The Ethereum blockchain dataset from Google Big Query was used to acquire smart contract data. Ultimately, 620,000 contracts from this source were used to train an LSTM model. Once more, one-hot vectors were used to represent the derived opcodes from the contracts. These vectors were converted into code vectors using embedding methods, resulting in decreased dimensionality and a stronger capability to capture potential relationships between sequences because this representation produces highly sparse and uninformative features. The statistical characteristics of the opcode lengths of contracts determined to be vulnerable and safe have been compared as another stage in the preprocessing process. They decided to only include contracts with a maximum opcode length of 1600 since they noticed that the features of the two categories varied noticeably. Additionally, it was discovered that the dataset's distribution (as labeled by MAIAN) was unbalanced, with non-vulnerable cases making up 99.03 percent of the dataset. In order to obtain a fair distribution in the training set, all vulnerable contracts were clustered together and oversampled using the Synthetic Minority Oversampling Technique (SMOTE). The outcomes showed that sequential learning methods outperformed symbolic analysis tools. The model earned an F1 score of 86.04 percent and a vulnerability detection accuracy of 99.57 percent.

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

In 2019, In this research, we proposed a machine learning-based model to detect security vulnerabilities of smart contracts on the Ethereum platform. We used static code analysis as the underlying technology and trained various machine learning models for security vulnerabilities. Our model found 16 different vulnerabilities with an average accuracy of 95to directly using static code analysis tools. Checking many smart contracts using different static code analyzers is a huge burden on developers. In addition, they need to learn how each analyzer works and combine the results for a full evaluation. Furthermore, our model can be used to identify security vulnerabilities parallel to the development process of smart contracts, thus decreasing the cost of development by preventing the security vulnerabilities from being introduced in the early stages. Our proposed model also applies to other languages and platforms since the model has no language or platform dependencies. Training the model with different attentive language datasets and choosing the corresponding static code analyzers and AST builders, new machine learning code analyzers can be generated by following the steps described in Section III.

In 2020, Xing et al. [54] proposed a feature extraction method named slicing matrix. It consists of segmenting the opcode sequences derived from smart contract bytecodes to extract opcode features from each one individually. The purpose of this segmentation is to separate useful and useless opcodes. The extracted opcode features are then combined to form the slice matrix. To carry out a comparative analysis, three models were created. These were namely Neural Network Based on opcode Feature (NNBOOF), Convolution Neural Network Based on Slice Matrix (CNNBOSM), Random Forest Based on opcode Feature (RFBOOF) [25]. These models were each tested on three different vulnerability classification tasks: greedy contract vulnerability, arithmetic overflow/underflow vulnerability, and short address vulnerability. While RFBOOF achieved the best results in all three cases based on precision, recall, and F1 evaluation metrics, CNNBOSM performed slightly better than NNBOOF. The authors mention that the slice matrix feature needs further exploring.

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

To identify various smart contract vulnerabilities, Liu Z. et al. designed a machine learning approach combining GNN and expert knowledge (cite:hwang2020gap). The goal of a graph neural network (GNN), a deep learning technique, is to make inferences from data represented by graphs. A graph is a data structure used in computer science that consists of nodes (also known as vertices) and edges. According to research, the semantic relationships between programming elements can be preserved when written programs are transformed into symbolic graph representations. As a result, contract graphs can be used to represent smart contract codes. The suggested strategy, as depicted in figure 3, begins with two distinct concurrent processes (Security pattern extraction and contract graph extraction) and then uses a combining layer to combine patterns in each segment to identify vulnerabilities. A feed-forward neural network first creates the pattern feature for extracting security patterns from the contract's source code. To extract the expert patterns from smart contract functions, they employed an open-source program. A GNN must be created in the second process (message propagation phase) to produce a contract graph. Nodes, or program elements, made up the GNN model, while edges, or the next function to be executed, indicated the flow of each program element. Later, using a node elimination approach, undesirable nodes and edges are eliminated. The authors used a preprocessing technique that involved casting the source code's extensive control and data flow semantics into a contract graph. Following this, they created a node elimination stage to normalize the network and highlight important nodes. The vulnerability detection phase, where both extracted features are integrated with convolution and a fully connected layer, was used to combine these two simultaneous procedures. The suggested model is tested against security detection methods that are not ML-based, including Oyente, Mythril, Smartcheck, Securify, and Slither. Re-entrancy, timestamp dependence, and endless loop vulnerabilities of each function in the source code were all searched for by each algorithm and the proposed model. a The proposed methods (CGE) found reentrancy and timestamp-dependent type vulnerabilities with an accuracy of 89 percent and an infinite loop vulnerability (cite:hwang2020gap) with an accuracy of 83 percent.

When a piece of code is rewritten, the Eth2Vec model is suggested to address a flaw in the present vulnerability detection tools. A code rewrite in a programming language is reimplementing a source code's functionality without utilizing the original. Finding vulnerabilities becomes more difficult when the smart contract codes are modified. The authors first transformed each smart contract's source code into EVM bytecodes. Only useful information (such as function ids and lists of callee functions) for vulnerability detection was taken out of the bytecode by the authors. A neural network structure is utilized as the final step to find any vulnerabilities in the source code. Five hundred contracts were used to test the suggested model, and even though the contracts were changed, the Eth2Vec model could identify vulnerabilities with a 77 percent accuracy.

O. Lutz et al. [11] and present another technique for identifying weaknesses in smart contracts. The authors offer a method called ESCORT, which employs a Deep Neural Network model to discover the semantics of the input smart contract and identify particular vulnerability types based on the discovered semantics. The ESCORT model aims to get over the limitations of existing non-DNN models in terms of scalability and generalization. With a detection period of 0.02 seconds for each contract, the experimental results of this article produced an F1 accuracy score of 9 percent on six different vulnerability types. Scalability is easier to achieve with rapid detection times, which meets one of the author's objectives. The ESCORT model, therefore, somewhat resolves the problems identified in earlier works, such as Y. Xu or N. Lesimple's models, where the model can realize novel weaknesses. [21] Unfortunately, it is relatively challenging to derive interpretability from such models, and even if new vulnerabilities are discovered, figuring out their root causes is still very challenging. Sun et al. used machine learning to try and find the following vulnerabilities: reentrancy, arithmetic problems (integer overflow/underflow), and timestamp dependencies. In order to accommodate for differences in instructions between compilers, some stack operating instructions were trimmed into more general versions (e.g., SWAP1, SWAP2,..., SWAPn. SWAPx). After that, as a label normalization step, opcodes were divided into nine categories depending on their purposes. A word2vec transformation of the opcode sequences prior to the convolutional layers was carried out, just like in "cite" etehrTrust. This article introduces an additional self-attention layer in addition to the pooling and softmax layers that typically follow convolutional layers. The one-hot encoders used to encode each opcode instruction are merely representatives and do not capture any functional similarity between them; therefore, the self-attention layer's goal is to establish a connection between adjacent words in the acquired feature matrix. [21]. By using self-attention, the word embedding process has been improved as a result. CiteetehrTrust and this paper used CNNs to find vulnerabilities, but CiteetehrTrust used a word2vec embedding, whereas this paper used an attention method, which is why they got superior results. The key advantage of the newly developed model over the static analyzers that are already in use, such as Oyente and Mythril, is that it can attain comparable performance in a lot less time. In their paper, Y. Xu et al. developed two unique methods for identifying vulnerable smart contracts: the Stochastic Gradient Descent (SGD) model and the K Nearest Neighbors (KNN) model. They seek to use each machine learning model to discover eight of the most widely known classical vulnerability types, including arithmetic, reentrancy, denial of service, uncontrolled low-level calls, access control, faulty randomization, front running, and denial of service. Similar to N.Lesimple's paper [24], their model employs an AST structure as its input, enabling it to parse the smart contract code line by line. Through the use of conventional techniques, the labels for the vulnerabilities were found. High recall, precision, and accuracy are noted for four of the eight vulnerabilities in the paper. The results for the remaining four were deemed inconclusive since there were not enough samples in the dataset. The test set was produced from the outcomes of utilizing conventional approaches, similar to the N. Lesimple study, showing that the authors could not demonstrate how the KNN model differed from conventional methods.

In 2021, by using bigram properties from the streamlined operation codes of smart contracts, Wang et al. [51] introduced their approach, ContractWard, to identify vulnerabilities in smart contracts. They gathered a dataset of 49,502 smart contracts from the Etherscan website, verified before September 2018, and found that each contract had six potential weaknesses: Integer overflow/underflow, transaction ordering dependency, call stack depth attack, timestamp dependency, and re-entrancy Each smart contract's source code is converted to opcodes; A smart contract typically has 100 different opcodes and 4364 opcode components. There were only 50 opcode types left after they conducted a simplifying process. As a result, the authors grouped several opcodes with related functionality into a single category, which simplified the dataset's features. Because they believe that operations have a stronger relationship with their neighbors, they later adopted the n-gram approach (a sliding window of binary-byte size) to track relationships between each opcode. Each of the contract's many labels was assigned using the Oyente [34] system. Due to the scarcity of particular vulnerabilities, the researchers ran into a class-imbalance problem after the labeling process. Extreme Gradient Boosting (XGBoost), Adaptive Boosting (AdaBoost), Random Forest (RF), Support Vector Machine (SVM), and k-Nearest Neighbour were the five candidate ML models used in the training procedure (KNN). By reaching above 96 percent F1, Micro-F1, and Macro-F1score, the XGBoost model demonstrated strong performance.

In 2022, Yuqi Fan et al. note that most of the studies on methods of vulnerability detection regarding smart contracts take rely on pre-defined manual rules from experts and auditing professionals. [16] Devising such rules and patterns are very time-intensive and labor-demanding. They discuss the previous efforts at employing deep learning methods to make up for such shortcomings, but they fail to represent the source code/bytecode well semantically and structurally. Then, they propose a novel model of Dual Attention Graph Convolutional Network (DA-GCN) to detect smart contract vulnerabilities. They extract both control flow graph and opcode sequence from smart contracts' bytecodes and feed them as input into a feature extraction pipeline. Afterward, they use a multilayer neural network to identify the vulnerable smart contracts. They tested their proposed model on smart contracts containing one of the two vulnerabilities: reentrancy and timestamp dependency. Their experimental results demonstrated that the DA-GCN model achieved an accuracy of 91.2% and 87.5% in the two smart contract vulnerability detection tasks.

In 2022, Zhang et al. [57] propose a novel model to detect smart contract vulnerabilities: ensemble learning (EL)-based contract vulnerability prediction method. It is based on seven different neural networks using vulnerability data for vulnerability detection at the scope of smart contracts (single file-level scope). Seven neural network (NN) models were first pre-trained using an information graph (IG) consisting of source datasets, which then were integrated into an ensemble model called the Smart Contract Vulnerability Detection method based on Information Graph and Ensemble Learning (SCVDIE). The effectiveness of the SCVDIE model was verified using a target dataset composed of IG, and then its performances were compared with static tools and seven independent data-driven methods. The verification and comparison results show that the proposed SCVDIE method has higher accuracy and robustness than other data-driven methods in predicting smart contract vulnerabilities.

Also, in 2022, Zhang et al. [56] propose another novel method for vulnerability detection: A flexible and systematic hybrid model, which they have named the Serial-Parallel Convolutional Bidirectional Gated Recurrent Network Model, incorporating Ensemble Classifiers (SPCBIG-EC). Their new model shows noticeable improvements in performance concerning smart contract vulnerability detection. In addition, they also propose a serial-parallel convolution (SPCNN) suitable for this hybrid model and generally serial combinatorial models. It is equipped with the capability to extract features from an input sequence for multivariate combinations while retaining temporal structure and location information. The Ensemble Classifier is used in the classification phase of the model to enhance its robustness. In their experiments, they focused on six typical smart contract vulnerabilities and constructed two datasets, CESC and UCESC, for multi-task vulnerability detection.

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

3.4 SLITHER-SIMIL

Parts of this section have been published in another piece [41] 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. concerning automating smart contract security assessments have included works on an addition to an already prominent static analysis tool, Slither [17], 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 addition to Slither -a static analysis tool named SLITHER-SIMIL.

Based on accumulated knowledge embedded in previous audits, we set out to detect similarly vulnerable code snippets in new clients' codebases. Specifically, we explored machine learning (ML) approaches to automatically improve 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 [42], 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 their similarity. Last year, we took it to the next level and applied it directly to vulnerable code.

SLITHER-SIMILcurrently uses its representation of Solidity code, as introduced by [17], namely SlithIR. SlithIR (Slither Intermediate Representation), to encode Solidity snippets at the granularity level of functions. We thought the function-level analysis was a good place to start our research since it is not too coarse (like the file level) and not too detailed (like the statement or line level).

As introduced by Feist et al., 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. Each node can contain up to a single Solidity expression in that graph, which is converted to a set of SlithIR instructions. This representation makes implementing analyses easier without losing the critical semantic information contained in the Solidity source code. [17] SlithIR has a database of about 40 instruction expressions. It has no internal control flow representation and relies on Slither's control-flow graph structure (SlithIR code is associated with each node in the graph). The complete descriptions are available at [38].

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

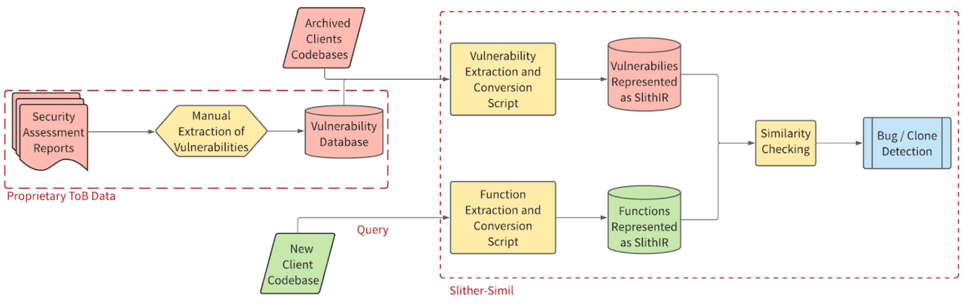
Archived Clients Codebases
Security Assessment Reports
Manual Extraction of Vulnerabilities
Vulnerability Database
Proprietary ToB Data
Vulnerability Extraction and Conversion Script
Vulnerabilies Represented as SlithIR
Similarity Checking
Bug / Clone Detection
Function Extraction and Conversion Script
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.

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 is 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

```
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.

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 us take a look at the function transferFrom from the ETQuality.sol smart contract to see how its structure resembles 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 [45] and SmartEmbed [**?**], which use ML approaches to discovering vulnerabilities in smart contracts are gradually showing promising results, with fewer 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 [20]).

We still face the challenge of data scarcity concerning the scale of smart contracts available for analysis and the frequency of interesting vulnerabilities appearing in them. We can focus on the ML model because it is a more facilitated process, but it does not do much good for us in the case of Solidity, where even the language itself is very young, and we need to tread carefully in how we treat the amount of data we have at our disposal. Archiving previous client data was a job since we had to deal with the different solc versions to compile each project separately.

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 several 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 an unsupervised baseline 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 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 developing a proprietary database of our previous clients and their publicly available vulnerable smart contracts and references in papers and other audits. Together they will form one unified, comprehensive database of Solidity vulnerabilities for queries, later training, and testing newer models.

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

Furthermore, we are looking forward to checking out the performance SLITHER-SIMILon 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 will also test it on Mainnet when it becomes more mature and automatically scalable.

SLITHER-SIMIL is now available on Github and ready to use. For end users, it is 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 research community's efforts to propose machine-learning-based methods to discover and mitigate vulnerabilities in smart contracts compared to the existing approaches used in the industry. We also went over SLITHER-SIMIL, a powerful tool with the potential to measure the similarity between function snippets of any size written in Solidity. 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. Nevertheless, 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 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-executing program stored on the Ethereum blockchain that runs when pre-defined conditions are satisfied. In the past few years, the research community has developed automated analysis tools, and frameorks [27] that locate and eliminate vulnerabilities in smart contracts to make 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 potentially vulnerable. [37] 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. [33], [44] Comparing and reproducing such research is not an effortless process. [44] Most datasets used to test and benchmark those tools proposed in the research literature are not publicly or readily available to the research community. This makes reproduction efforts immensely hard and time-intensive to carry out.

The current approach to comparing 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 must start from scratch and create their datasets, a non-trivial and slow process. What makes it worse is the data bias, which can be introduced in a dataset in different phases of data acquisition and cleaning. This can quickly become a threat to the validity of the research. [44]

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 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 ETHERBASEand make its pipelines open-source. It obtains historical data and facilitates benchmarking and reproduction in research and development for new toolsets. It is more up-to-date than existing datasets and gets automatically reviewed and renewed compared to the previous manual one-time data gathering efforts.

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

4. We propose the first automatically up-to-date labeled dataset of Ethereum smart contracts with vulnerabilities.

4.2 Related Work

We assume the reader is familiar with blockchain technology, Ethereum blockchain, and its primary high-level programming language, Solidity. Ethereum Smart contracts are developed mostly in the programming language Solidity, and for execution, they get compiled to their corresponding bytecode through EVM. EVM takes bytecode as input and works in a stack-based architecture with a word size of 256 bits. [32] 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 [12] 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 nine analysis tools on a small (< 100) labeled 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 tremendous improvement for both tool development and reproduction efforts. However, because of the lack of some degree of ground truth in the dataset, it is 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 suitable for initial benchmarking. However, only the contract source code is available, and they do not provide any bytecode for them.

Ren et al. [44] created a benchmark suite that integrates annotated and unlabelled raw smart contracts from a variety of sources such as Etherscan [15], SolidiFI repository, CVE Library, and Smart Contract Weakness (SWC) Registry. They collected 45,622 real-world diversified Ethereum smart contracts, 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 not appropriately annotated, or a minimal subset of them is labeled with the vulnerabilities' identification properties. This is because the number of the contracts -most of them clones of other smart contracts- is so huge that they cannot be manually annotated. To the best of our knowledge, there has been only one annotated dataset, released publicly, from Yashavant et al. [55] We strive to build upon that work and facilitate reproduction processes 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 concerning those contracts.

Luu et al. [34] collected 19,366 smart contracts from the blockchain and provided their blockchain addresses alongside analysis results on each contract on whether they contain any of their selected four vulnerabilities or not.

In 2022, Yashavant et al. [55]

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; [40] this makes it hard to determine whether the identified smart contracts have been tested or used on the Ethereum blockchain. 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 ABIs or Opcodes to the best of our knowledge.

As a user of the Website Etherscan, one can easily search the Ethereum blockchain for any specific smart contract given the availability of its address, but when it comes to downloading the data for that smart contract or any other batch of smart contracts, using Etherscan has its limits: [40]: Smart contracts' data is massive, based on the estimation from [40], and the daily limits on the APIs provided by Etherscan make retrieving that data even harder. The API provided by Etherscan does not allow the users to obtain a list of the addresses of their desired smart contracts. The API calls currently available only allow navigation at the block level. Researchers cannot easily explore the source code for their collected smart contracts. First, they would have to inspect any block on-chain and search for transactions with a receive/send address associated with that specific smart contract.

4.3 Methodology

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

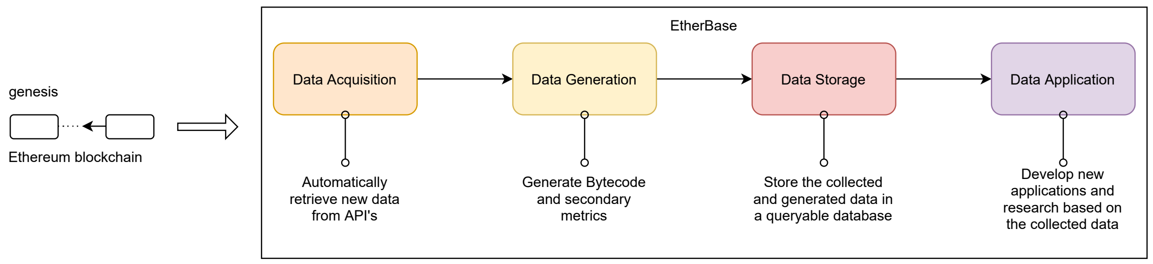
genesis
Ethereum blockchain
EtherBase
Data Acquisition
Data Generation
Data Storage
Data Application
Automatically retrieve new data from API's
Generate Bytecode and secondary metrics
Store the collected and generated data in a queryable database
Develop new applications and research based on the collected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

Figure 4: EtherBase Worflow

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 steps of the workflow as mentioned earlier 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 corresponding addresses, In order to collect the contracts, we retrieve the addresses for every contract with more than one transaction through Google BigQuery, like the process done by [44]. Using Google BigQuery query request service, we obtain 1,712,347 separate contract addresses with more than one associated transaction.

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, Crytic-compile. All the previous works of research work on providing either source code or bytecode of smart contracts to the researchers, but that will not be enough for the users who want to compile the smart contracts or build tools upon such data. Compiling smart contracts is also tricky due to the rapid pace of changing versions of the dominant programming language, Solidity, used to write and develop smart contracts. We utilized Crytic-compile, a library to help compile smart contracts to help with this problem. It helps the user avoid maintaining an interface with solc and automatically finds and uses the correct version of solc or a better compatible version of solc to compile the input smart contracts. This works under the hood because crytic-compile compiles an input smart contract and outputs a compilation unit in the standard solc output format, written in a JSON file, alongside the source code and other metadata. 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 [13] to remove the duplicated smart contracts by checking the MD5 checksums (32-character hexadecimal numbers computed for each file) of each of the two source files in the collected dataset to see if they are the same and after removing the whitespace among the lines of code. After the process of deduplication is done, we get down to 48,622 smart contracts. For this paper, and as of now, we have released 5,000 smart contracts for tool comparison purposes. Many metrics exist that we have access to, can calculate, and add to ETHERBASE. However, not much research has been conducted on the applicability of these many different metrics to empirical research on the Ethereum blockchain or how much it appeals to the researchers active in this field. In the following, we describe the initial set of the metrics we selected to include in EtherBase in the form of a table, to be followed by more, after more discussion and research on their applicability to research on smart contracts.

The built-in metrics relating to smart contracts are those features that depend on the internal properties of a smart contract, 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 deployer of the smart contract.
`Source Code`	Source code of the smart contract, specific to Solidity programming language.
`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 they hold.
`Transaction Count`	Number of internal transactions from each smart contract.

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

- `Pragma`: Source files can (and should) be annotated with a 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 homogeneous set of contracts with a consistent syntax.

- `Contract Address`: Contract address is the main key in the database of ETHERBASEfor distinguishing smart contracts from each other. It is usually given when a contract is deployed to the Ethereum.

- `Creator Address`: The contract address is usually given when a contract is deployed to the Ethereum Blockchain. The address comes from the creator's address, where the contract has been initially deployed, alongside the number of transactions sent from that address (the "nonce"). [3]

- `Source Code`: The source code of each Ethereum smart contract is written in Solidity and helps researchers do all sorts of analyses on the smart contracts.

- `Bytecode (bin)`: The regular `bin` output is the code placed on the blockchain plus the code needed to get this code placed on the blockchain, the code of the constructor.

- `Bytecode (bin-runtime)`: `bin-runtime` is the code that is actually placed on the blockchain.

- `ABI`: `ABI` stands for Application Binary Interface. It is the standard way of interaction with smart contracts on Ethereum, both from outside the blockchain ecosystem and for contract-to-contract interaction. [4]

- `Block Number`: `Block Number`is the length of the blockchain in blocks, more specifically, the block on which the smart contract exists.

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

- `Transaction Count`: Like ETH Value, transaction count is an important metric for us 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. Afterward, users can analyze their queried data according to their specific research queries in the data application stage.

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

```
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 enjoys a graphical user interface (GUI) to allow the less technical end users to access and browse through the database. We integrated ETHERBASEwith Apache Superset, a powerful business intelligence tool that allows one to create charts and dashboards using the data from the database.

4.3.4 Data Application

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

- **Smartcheck:** Smartcheck [48] is an extensible static analysis tool written in Java. It detects vulnerabilities and other code issues in Ethereum smart contracts. It locates vulnerabilities by searching for pre-defined patterns in a transformed version of the Solidity source code of the contract.

- **Mythril:** Mythril is another frequently used static analyzer in the form of a CLI tool developed in Python that does security analysis of Ethereum smart contracts.

- **Slither:** Slither [17] is a static analyzer for analyzing Ethereum smart contracts before deploying them and evaluating them in runtime.

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

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

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

Table 2: Supported Vulnerabilities

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

- **Unchecked Ether:** also known as silent failing sends, this vulnerability can cause unexpected/undefined behavior if the return values are not managed properly before executing the smart contract. [23] The SWC Registry ID of this contract is SWC-104. [7]

Like the work done by [55], we also leverage the methodology given in the paper by Ren et al. [43] to detect the selected vulnerabilities in smart contracts.

When using tools like these static analyzers, we face many false positive results because of the methods those tools employ. Because of that, we cannot rely on one tool only, as projects that rely upon 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 [55], the following explains the step-by-step methodology concerning the majority voting mechanism as mentioned earlier:

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

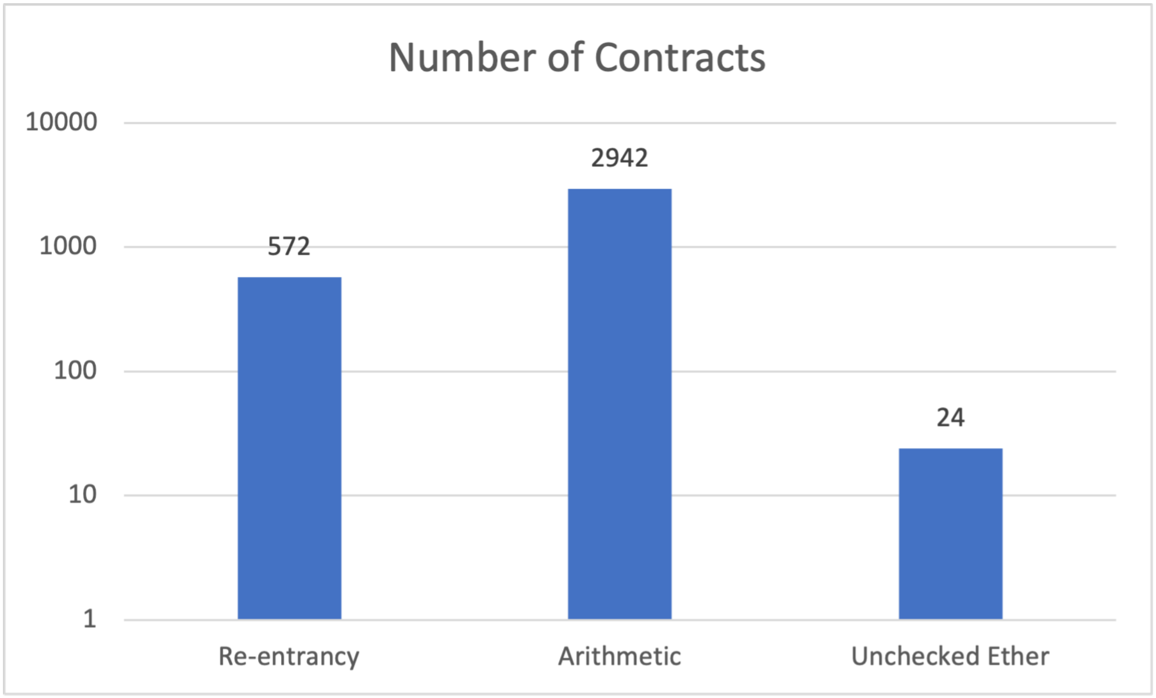

Figure 5: No. of Contracts containing vulnerabilities (log-scale)

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

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

Figure 5 showcases the number of smart contract files which contain at least one potential vulnerability. For instance, consider a specifically targeted vulnerability. The literature tells us that all three static analyzers in the benchmark support the detection of this vulnerability. [55] notes that they only say the contract at hand contains that vulnerability only if an agreed-upon threshold of the tools reports that it is present at the same location, based on the majority voting system proposed by them. We also take on the same system for determining whether a vulnerability is present.

4.4 Concluding Remarks

This chapter introduces an up-to-date database centered around Ethereum smart contracts, namely ETHERBASE, 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:

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

In addition, our current method is restricted by the rate limit imposed by the APIs of different services like those of Etherscan and Infura. For example, Etherescan restricts the daily frequency of queries to its API (5 per day). [55] states this as a serious issue that we would like to solve in future work.

The Ethereum security research community can use ETHERBASEfor evaluating the 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 [55] notices this as well. There are always many false positive/negative results in an audit report generated by a static analyzer. Nevertheless, we utilized the mechanism of majority voting suggested by [55] to determine a vulnerability's presence or lack thereof in a smart contract. We should, however, be wary of the generated false positive results as too many of them will lead to increasing inaccuracies in the released dataset. Such issues can be overcome by adding more tools to the benchmark process or having some auditors manually review the discovered potential vulnerabilities.

Our competitive advantage in comparison to the work done by [55] is that ETHERBASEgets updated regularly, is more comprehensive with regards to the various versions of smart contracts it contains and that it leverages offline powerful compilation tools to retrieve more metadata about the collected smart contracts instead of going through the time-consuming process of validating each collected contract with online services separately and through manual development of data collection pipelines.

Chapter 5

Conclusion and Future Work

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

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

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

We proposed a version of Slither-simil as the first machine-learning-based tool based on a static analysis tool and how it extended into developing EtherBase.

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