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, state-based program that is written in high level languages such as 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. [39] 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. Base 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. Afterwards, we propose our own solution, SLITHER-SIMILand how it led us to the development of ETHERBASE.

3.2 Existing Analysis Tools

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:

Static Analysis Static analysis, which can be done at both the source code and bytecode levels, is a technique for studying a computer programme without running it. 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. Normally, other tools will first obtain binary bytecode, and then use it to construct a custom intermediate representation, which have a series of forms, like SSA used in Slither [14], Datalog used in Securify [42] and MadMax [17], XML parsing tree used in SmartCheck [41] and XCFG used in ClairvOyance [46]. Based on this representation, vulnerability pattern definition and matching are performed to screen out suspected code snippets. Formal verification, a static analysis technique based on mathematical proof, is frequently used to check the logical integrity of smart contracts like Zeus [25], VeriSmart [38], and EtherTrust [18].

Dynamic Analysis Fuzzing is a technique for finding software bugs that involves creating erroneous input data and watching the target program's unusual output while it runs. It allows developers to ensure a uniform standard of quality through prepared tests, but does not narrow down the causes of detected bugs. When applied to smart contracts, a fuzzing engine will first try to generate initial seeds to form executable transactions. With reference to the feedback of test results, it will dynamically adjust the generated data to explore as much smart contract state space as possible. Finally, it will analyze the status of each transaction based on the finite state machine to detect whether there is an attackable threat. ContractFuzzer [23] is the first to apply fuzz testing to smart contracts. Later, other researchers start to study improvements to different parts of fuzzing. ReGuard [26] and Harvey [45] are dedicated to generating diverse inputs and transactions that are more likely to reveal vulnerabilities, ILF [20] and sFuzz [30] target at designing more effective generation or mutation strategy.

Symbolic Execution When using symbolic execution to analyze a program, it will use symbolic values as input instead of the specific values during the execution. When a fork

is reached, the analyzer will collect the corresponding path constraints, and then use a constraint solver to obtain specific values that can trigger each branch. Symbolic execution can simultaneously explore multiple paths that the program can take under different inputs, but it also faces unavoidable problems such as path explosion. In most cases, the symbolic executor will first build a control flow graph based on Ethereum bytecode, then design corresponding constraints based on the characteristics of smart contract vulnerabilities, and finally use the constraint solver to generate satisfying test cases; for example, Oyente [28], Mythril [6]. In recent years, there has been continuous research to optimize the process of symbolic execution. Manticore [29] adds the support of exotic execution environments, DefectChecker [4] extracts defect related features to help improve efficiency, sCompile [2] identifies critical paths which involve monetary transaction and VerX [34] focuses on verifying effectively external callback free contracts.

3.3 Deep Learning in Smart Contracts

In this section, we will go over some of the literature focusing their efforts on replacing the existing tools' capabilities explained in the previous section with 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 [16]. 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 [17]. 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 [16].

Liao et al. have adopted a sequence learning approach to detect smart contract security threats [17]. 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 [18]. 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 [18]. 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. [21] 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) [21]. These three models were each tested on three different vulnerability classification tasks: greedy contract vulnerability, arithmetic overflow/underflow vulnerability and short address vulnerability. While RFBOOF achieved the best results in all three cases based on precision, recall and F1 evaluation metrics, CNNBOSM performed slightly better than NNBOOF in general. The authors mention that the slice matrix feature need further exploring.

In 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 [22]. 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 [22].

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. [9] 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 [16], 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 [16]. 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 [18] 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 [16]. As a result, the word embedding process has been enhanced through the use of self-attention. When compared to the vulnerability detection performance of [18], they have both used a CNN but [18] 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 [23] 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 [14]. 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 [20], 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.

ContractWard model is proposed as a faster alternative for Oyente [28]. The dataset consisted of 49502 smart contracts, where each of them contained six possible vulnerabilities: integer overflow/underflow, transaction ordering dependency, call stack depth attack, timestamp dependency, and re-entrancy vulnerability. Each contract's source code is converted 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 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. They employed synthetic minority oversampling technique to extend the number of minority class. 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). In evolution stage, Micro-F1 and MicroF1 (variations of F1 metric) is utilized to rank the predictors. The XGBoost model showed a robust performance by achieving over 96% F1, Micro-F1, and Macro-F1score.

3.4 SLITHER-SIMIL

Our efforts with regard to automating smart contract security assessments was to work on an addition to an already prominent static analysis tool, 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 make life a bit easier for both auditors and clients.

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 [35], 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. This year, we took it to the next level and applying it directly to vulnerable code.

Slither-simil currently uses its own representation of Solidity code, 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).

SlithIR is the hybrid intermediate representation used in Slither to represent Solidity code. Each node of the control flow graph 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.

SlithIR uses fewer than 40 instructions. 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 [32].

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

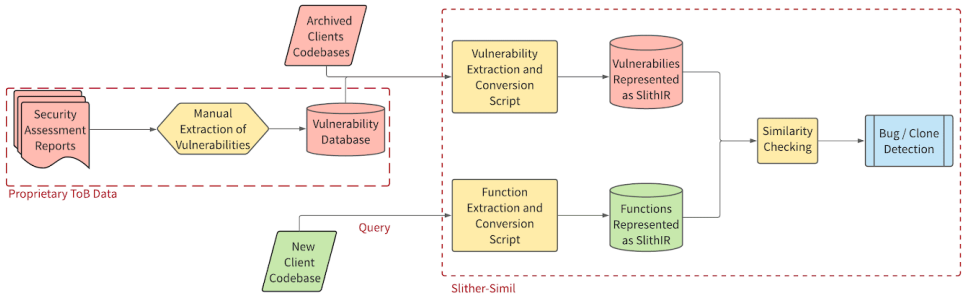


Figure 3: A high-level view of the process workflow of Slither-simil.

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

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

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.

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:

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

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

```
type_conversion(uint256)

binary(**)

binary(*)

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

index(uint256)

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

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

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

...

```

Listing 3.3: Normalized SlithIR tokens of the previous expressions.

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 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 and SmartEmbed, which use ML approaches to discovering vulnerabilities in smart contracts, are showing promising results.

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).

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.)

The pie chart shows how 89 vulnerabilities were distributed among the 10 client security assessments we surveyed. We documented both the notable vulnerabilities and those that were not discoverable by Slither.

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: 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

We also proposed 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

This chapter is adapted from work supervised by Jeremy Clark and Amir G. Aghdam, to-be-submitted at the 7th Workshop on Trusted Smart Contracts (WTSC) co-located with Financial Cryptography and Data Security (FC).

4.1 Introductory Remarks

Ethereum blockchain handles financial assets and contracts with millions of USD in value. It is the most widely used blockchain platform with millions of smart contracts written on it. A smart contract is simply a program stored on the Ethereum blockchain that runs when predetermined conditions are met. There has been some effort from the research community to develop automated analysis tools and frameorks [24] that locate and eliminate vulnerabilities in smart contracts. These tools and frameworks analyze smart contracts and produce vulnerability reports. In a preliminary study performed on nearly one million

Ethereum smart contracts, using one analysis framework for verifying correctness, the researchers flagged 34,200 of them as vulnerable. [31] 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. [27]

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 [33].

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 intends to compare their new tool with existing work and projects, the current approach is to contact the authors of alternative tools and hope for access to the same datasets or 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. [37]

In this chapter, we present ETHERBASE, an extensible, queryable, and easy-to-use database that facilitates and enhances the verification and reproducibility of previous empirical research and lays the groundwork for faster, more rapid production of research on smart contracts. ETHERBASEis open-source and publicly available online at: **link TBD**. 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.

We also note that although ETHERBASEhas been implemented for Ethereum, it can be easily extended to support other blockchains which have the similar design for smart contracts with Ethereum.

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 ETHERBASEafter addressing several technical challenges. It obtains historical data and enables new functionalities. It is more up-to-date than existing datasets, gets systematically 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 written in Solidity are compiled to bytecode to be executed on the Ethereum Virtual Machine (EVM). 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. 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 [10] makes the top of the list as one of the most used benchmarks in the research space. Durieux et al. crafted a dataset of annotated and non-annotated smart contracts. The annotated part contains 69 contracts tagged with 115 vulnerabilities. The annotated part has ten categories of vulnerabilities. In contrast, the non annotated part contains 47,518 unique contracts, each with at least one transaction on the Ethereum network. Researchers can't use the unlabeled dataset to evaluate the tool as there is no ground truth associated with it. Moreover, the labeled dataset doesn't have more than a thousand smart contracts and only the source code (and no bytecode) of the contracts are available.

Ren et al. [37] crafted a benchmark suite that integrates labeled and unlabeled Smart Contracts from a variety of sources such as Etherscan [13], SolidiFI repository, Common Vulnerabilities and Exposures Library, and Smart Contract Weakness Classification and Test Cases library. Their pblicly released dataset has 45,622 real-world unlabeled Ethereum smart contracts having more than one transaction on the Ethereum blockchain. The labeled dataset has artificially constructed contracts (350 contracts) and confirmed vulnerable contracts (214 contracts).

Many of the authors of the tools proposed in this practice leverage unlabelled datasets to evaluate their toolset. Due to huge number of contracts in these datasets, either the dataset is not annotated or a very small subset is annotated with the vulnerabilities present. To the best of our knowledge, there has been only one annotated dataset, realeased publicly, from Yashavant et al. [47] 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. [25] published their analysis results for 1,524 smart contracts with no other information or metadata in relation to those contracts.

Luu et al. [28] 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.

The smart contract source codes collected in GitHub typically do not directly reference smart contracts deployed on the blockchain through an Ethereum address; therefore, it is hard to determine whether it has been tested or used on the 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. On a GitHub repository, there is no information on smart contracts' use in a real blockchain scenario, on the number of transactions invoking smart contracts or on the number of tokens associated with each smart contract. 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.

Concerning the block explorer Etherscan, they allow for exploration and search of Ethereum blockchain for smart contracts. However, when downloading the smart contracts' source code, the block explorer presents some limitations: Smart contracts' data and numbers in quantity are massive (on the Giga scale, based on our estimation), but there is a limited API rate of 100 submissions per day per user to retrieve just a smart contract, making the complete download of data an impossible endeavour. Etherscan's API does not provide facilities to obtain a list of the smart contracts' addresses, as the existing API calls mainly allow navigation from one block to another. A researcher cannot directly and easily explore the smart contracts' source code but has first to inspect any block in Ethereum and then look for all the transactions that involve an address associated with the smart contract.

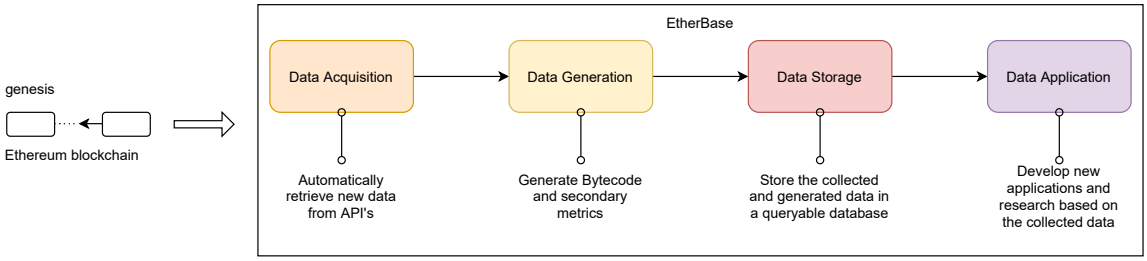


Figure 4: EtherBase Worflow

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 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 is collected from Etherscan, which is the largest decentralized platform for Ethereum smart contracts. Every contract stored on Etherscan's database is indexed

by its own corresponding addresses, In order to collect these smart contracts, We first retrieve the addresses for every contract which has more than one transaction through Google BigQuery. 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 writing scripts to obtain the bytecode and/or source code thrpugh Etherscan's API, 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 [11] 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.
`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 reject compilation with future compiler versions that might introduce incompatible changes. 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.

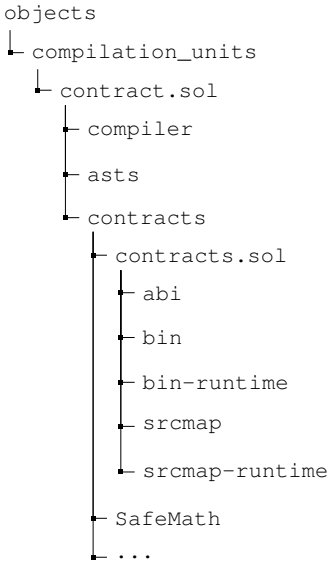
- `Transaction Count`: Like ETH Value, transaction count is an important metric for us to be able to exclude contracts that do not participate much on the chain and hence, work on the contracts thast have a higher probability of interaction with more contracts.

4.3.3 Data Storage

In the data storage stage, all extracted data are stored in PostgreSQL-alongside a GitHub repository- for the ease of data management. Users can get access to the collected data in PostgreSQL. In the application stage, users can conduct various analyses on the collected data.

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

The `artifact.zip` file contains



In addition to making the datasets available on GitHub, 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:

- **Slither:** Slither [14] is a static analysis tool designed to analyze Ethereum smart contracts. It has four prominent use cases: automated detection of vulnerabilities, automated detection of code optimization opportunities, improvement of users' understanding of the contracts, and assistant with code review.

- **Mythril:** Mythril is a tool that does security analysis of Ethereum smart contracts. It detects various security issues [6].

- **Smartcheck:** Smartcheck [41] is an extensible static analysis tool that detects vulnerabilities in smart contracts. It converts Solidity code into XML-based intermediate representation and checks it with XPATH patterns.

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.

Table 2: Supported Vulnerabilities

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

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

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

The ✓symbol shows that the tool can detect a particular vulnerability, whereas symbol ×indicates that the tool can't detect the vulnerability.

We use the methodology given in the paper by Ren et al. [36] to detect the selected vulnerabilities in smart contracts.

Because of the presence of false positives in the reports generated by the static analysis tools, we cannot depend on just one of them. We use majority voting, that is, at least 50% of the tools must report the same vulnerability at the same location. For example, assume that tools T1, T2, and T3 can detect the re-entrancy vulnerability. Suppose, at least two tools report that vulnerability is present in a smart contract in test. Then we say that the contract is vulnerable; Otherwise, we say that the contract is not vulnerable.

The following steps explain the methodology given the majority voting mechanism explained above:

1. First, collect the output of the selected tools for the smart contract to be analyzed.

2. Note the vulnerability name and line number for the vulnerability per tool.

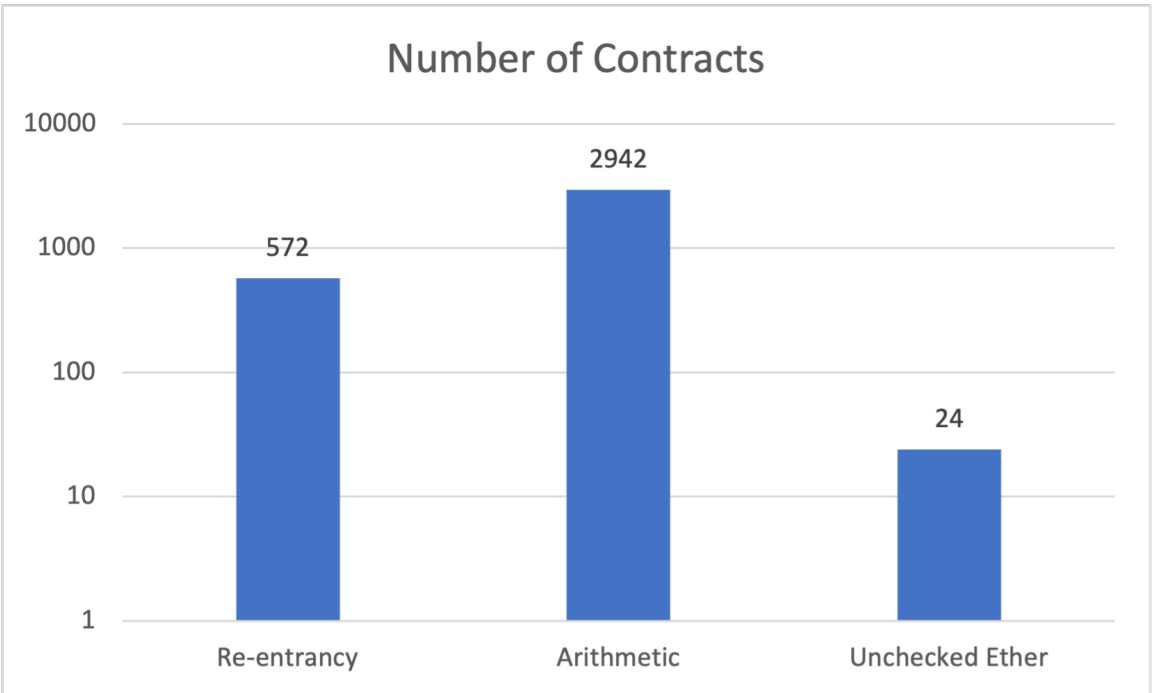


Figure 5: No. of Contracts containing vulnerabilities (log-scale)

3. For the same vulnerability, if different tools show different locations, consider them as different warnings. We consider two warnings as the same only if the vulnerability name and location match for different tools.

4. The methodology says that the vulnerability exists at a location if more than 50% of the tools (2 out of 3 in this experiment scenario) confirm the same vulnerability at the exact location. In that case, label the location with the vulnerability.

Figure 5 shows the number of smart contracts that contain at least one vulnerability. For instance, consider the Re-entrancy vulnerability. The literature tells us that all fo the the three static analyzers in the benchmark support the detection of this vulnerability. We say that the contract contains reentrancy only if two of these three tools report that it is present at the same location.

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 restricts the frequency of invoking its API to 5 queries per second. We plan to solve this as well to speed the automated processes.

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 one. There are always many false positive / negatives results generated by a static analyser. Nevertheless, we utilized the mechanism of majority voting to determine the presence or lacke of precense of a vulnerability in a smart contract. But, this approach may fail if majority of the tools generate false positives or false negatives. Such issues can be overcome by adding more tools to the benchamrk process or have some auditors manually review the discovered

potential vulnerabilities.. However, these approaches are time and resource-intensive, and we can plant to implement it incrementally.