Investigating Static Analyzers Detection Capabilities on Ethereum Smart Contracts

Mirko Staderini, Andrea Bondavalli
Department of Mathematics and Informatics
University of Florence
Florence, Italy
{mirko.staderini, andrea.bondavalli}@unifi.it

Abstract—Ethereum smart contracts had ever-increasing development in recent years. Hidden vulnerabilities can not be patched once a smart contract is deployed on a blockchain because of the code immutability. The use of static analyzers reduces the number of vulnerabilities in smart contracts. The paper focuses on the outcomes of some static analyzers for Solidity smart contracts. Based on a language-independent systematization of vulnerabilities, the work performs an analysis of positives detection of some static analyzers on a smart contracts dataset. Such analysis permits (i) identifying a positive correlation among some smart contracts’ metrics and tools’ outcomes, and (ii) investigating where tools detect specific classes of the systematization.

Keywords—smart contracts, tools, classification, metrics, correlation, location detection.

I. INTRODUCTION

Smart contracts are one of the most important innovations of the second generation of the Blockchain. The basic idea is to execute computerized transactions automatically. Their diffusion has allowed the development of applications in different areas (e.g., financial, medical, insurance, gaming, betting). Nevertheless, design and coding faults can cause weaknesses in implementing smart contracts. Weaknesses could lead to exploitable vulnerabilities. The problem is even more remarkable, considering that developers can not patch smart contracts once deployed on the Blockchain. Ethereum is one of the most used platforms, and it offers Solidity as the main (and Turing complete) programming language. Thus, the analysis of vulnerabilities for Solidity smart contracts is extremely important for the platform security.

The main methods to analyze code are the following [1]:

- **Static analysis** inspects the code without running it.
- **Dynamic analysis** executes the program and acts as an attacker looking for vulnerabilities.
- **Formal analysis** uses theorem provers or formal methods to analyze specific patterns.

Static analysis is widely used to discover vulnerabilities in the early stages of the software life cycle. It can cover 100% of the code at a low cost, despite the incomplete fault coverage [2]. The work focuses on selected static analysis tools (SATs) that analyze Solidity smart contracts. Each line of a contract under test can have two different outcomes: a negative result or a positive result. A positive result can be a true positive (TP - correct detection of an existing vulnerability) or a false positive (FP - wrong detection of a non-existing vulnerability). For our purpose, the paper investigates the positive results, referring to them as positives (P).

There are several vulnerability databases, e.g., BugTraq, and **Common Vulnerabilities and Exposures** (CVE). The National Vulnerable Database (NVD), the U.S: Government repository for vulnerabilities, uses the Common Weakness Enumerator (CWE) to classify CVE entries. CWE is a wide-spread used list of software weaknesses based on a hierarchical structure. Proceeding top-down, root is the most generic abstraction, and it represents the view. **Pillars** (or categories) and **classes** are independent of any specific language. **Bases** and **variants** describe the weakness at a lower level of detail. The entire CWE list has three hierarchical representations that focus on a specific aspect. The **research concept** is a language-independent representation based on the behavior of weaknesses [3].

Some studies show the link between the software's complexity and the outcomes of static analyzers (e.g. [4]). More recent work analyzes software metrics distribution in a set of Ethereum smart contracts extracted from Etherscan [5]. Our paper analyzes positives of SATs in checking Solidity smart contracts, focusing on software metrics and positives locations.

- As the first step, the work identifies a set of Solidity vulnerabilities, mapping them to a CWE taxonomy developed in [6]. Then, the work selects the SATs.
- Through a random extraction from Etherscan, a subset of smart contracts defines the dataset. The work identifies metrics for the analysis.
- Checking the dataset by all static analyzers permits to determine the correlation between some software metrics and positives.
- Finally, the work investigates locations in smart contracts where tools detect specific classes of vulnerability.

The organization of the paper is as follows. Section II provides the taxonomy and the SATs selection. Section III focuses on the dataset and software metrics. Section IV focuses on experiments, and Section V concludes the work.

II. TAXONOMY AND STATIC ANALYZERS SELECTION

To provide a list of vulnerabilities, we examine several papers, among them [7], [8], [9], [10], [11]. Moreover, we investigate some online documentation (e.g., the Smart Contract Weakness Classification and Test Case (SWC) registry) and Github repositories. For our purpose, we consider only vulnerabilities that can be exploited in the Solidity release >= 0.5. At the same time, we group vulnerabilities with a similar or overlapped definition. At the end of the selection process, we end up with 34 vulnerabilities. Using the methodology detailed in [6], the paper maps the vulnerabilities in ten categories that **TABLE I**
describes\(^1\). The complete taxonomy is available.

To define a set of SATs, the paper identifies a set of candidates of interest, starting with the survey of Di Angelo et al. [12]. Then, we look for Github repositories and research papers that treat static analyzers. Identifying some publicly available tools, we refine our research with the following criteria. At first, the standard release of the tool supports the Solidity release 0.5. Then, SATs can perform the analysis without assertions or user-defined properties. Finally, tools can detect vulnerabilities.

The process ends up with six tools. TABLE II summarizes the tool name and the tool release under analysis. Moreover, for each tool, it highlights the input mode (BC - bytecode or SC - source code) and the internal representation (CFG – control flow graph or AST – abstract syntax tree).

### III. DATASET AND METRICS

We build a dataset in two steps. At first, a Java crawler extracts randomly smart contracts from the public repository Etherscan. Next, we select smart contracts with Solidity release 0.5, ending up the process with 400 smart contracts.

Software metrics measure software complexity. Complexity is often related to the number of software faults. TABLE III summarizes software metrics that the work uses, grouped in four categories: length metrics, contract-oriented metrics, the cyclomatic complexity, and Halstead metrics. Note that the *.sol represents the smart contract file. Moreover, the table highlights the average and the standard deviation of each metric in the whole dataset.

Length metrics represent the line of code of the program, and they are strongly dependent on the programming style. Contracts oriented metrics are related to the number of logical contracts, functions, libraries, and parameters that a smart contract file contains. A Perl script parses the code of smart contracts and then calculates these metrics. The cyclomatic complexity is a quantitative measure computed using the CFG of the program. The paper uses a variant of the original metric, calculated explicitly for the Solidity language. It results in the sum of each function’s cyclomatic complexity (SCC [13]). Halstead metrics interprets the software as a sequence of tokens, in turn, classified into operators (OP) and operands (OD). Different combinations between OP and OD highlight distinct software characteristics. The paper calculates these metrics using a Solidity grammar for Another Tool for Language Recognition (ANTLR).

The work uses location metrics. The location of detection (LoD) is the line of a smart contract where a tool detects a positive. The relative location of detection (RLoD) is the LoD compared with the number of lines of the smart contract under analysis.

### IV. EXPERIMENTS

#### A. Methodology

The work performs an analysis of positives detection of SATs on a dataset. The basic idea consists of processing dataset by each SAT. Tools use several checking rules to detect vulnerabilities; each positive is related at least to one checking rule. Moreover, each tool delivers results in a different format; then, results are harmonized, mapped into the CWE category, and refined. Finally, each positive is represented by the tuple (tool, smart contract, location, CWE category). We use software metrics to correlate positives and contracts’ complexity. Finally, location metrics permits to highlight where tools find the different category of vulnerabilities.

#### B. Results

Fig. 1 and Fig. 2 show a preliminary data overview. Fig. 1 highlights the frequency distribution of positives. More than 65% of smart contracts have a number of positives between 0 and 49. Fig. 2 shows the distribution of positives for each tool, highlighting that one tool (Sfy2) detects the relative majority of positives compared with the other ones.

---

1 The complete taxonomy is available at this [link](#).
1) Correlation Analysis

This section shows the correlation between positives and software metrics. The use of the Pearson coefficient permits identify which metrics of TABLE III have the correlation coefficient greater than 0.80: (i) LOC (0.85) and LLOC (0.84) for the lines of code; (ii) functions (0.83), parameters (0.82), and statements (0.82) for the metrics smart contract oriented; (iii) SCC (0.82) for McCabe's complexity; (iv) H_LEN (0.86), H_VOC (0.85), H_VOL (0.85) for Halstead metrics.

Figure 3 depicts scatterplots showing, for each group, the metric that has the highest correlation with positives (functions, LOC, SCC, H_LEN). Observing evidence, simple contracts (low values on the x-axis) have a higher correlation with positives than complex contracts. On the other side, the high complexity of contracts (e.g., the number of variables) makes it hard to find vulnerabilities.

There is also a strong correlation among metrics and positives of a tool (except Mythril and Securify). Mythril focuses on detecting specific vulnerabilities depending on language features more than contracts’ complexity. The low correlation of Securify indicates its lower effectiveness. Positives grouped by CWE classes have a low correlation with each metric (coefficient between 0.2 and 0.4). These results suggest that smart contracts' complexity impacts the number of positives more than the class they belong to.

From these results, we can argue that contract developers should pay attention in the complexity of contracts. Complexity leads to error-prone contracts. Conversely, tools’ developers should focus on more complex contracts.

2) Analysis of classes distribution

This section highlights the distribution of CWE categories and the relation between classes and location metrics. Fig.4 highlights the category distribution: CWE-20 and CWE-284 compose 65% of the positives in the dataset. Referring to a specific class, the vulnerability detection capability of SAT affects the CWE distribution. Fig. 5 shows that every tool detects the class CWE 284; conversely, only two tools identify positives in CWE20, CWE-345, and CWE-400. As a consequence, subsequent tool updates should focus on finding bugs in less investigated classes.

Fig. 6 shows the frequency distribution of positives detection. The absolute location distribution reflects the distribution of LOC (the average value is 247). More than 60% of positives are located in the first 250 lines of contracts. The related location shows that positives are distributed throughout each contract.

Combining evidence of Fig. 7 permits investigating the location of contracts where tools can find a specific class of vulnerability. On the left of Fig. 7, a boxplot compares the location distribution for each class. For seven CWE classes, except CWE-400 and CWE-703, the lower hinge has a maximum value of 50. For the same CWE classes, half of the positives have location among lines 50 and 250 of contracts. Class CWE-703 (error in handling exceptions) has a slightly different distribution. CWE-400 (exhausting resources) is the class that shows the highest 50th percentile and upper hinge. Moreover, each distribution has many outliers caused by big smart contracts (in terms of LOC).

On the right of Fig. 7, a boxplot highlights the distributions of the relative location of positives for each class. Each distribution has a lower hinge greater than 25%. In the first part of each contract, there are preliminary definitions, libraries and interface declarations, and comments: they are not error-prone. CWE-330 has the lowest 25th percentile: this class refers to contracts that misuse random generation. Definition of random generators is typically located in the first part of a contract; moreover, the vulnerability affects the entire contract.
propagation: missing checks can cause vulnerability. The analysis of 15 sample contracts shows that functions with this vulnerability are located in the second part of contracts, although they have low complexity. CWE-400 refers to resources exhaustion. Both boxplots highlight that we find this class in big contracts and, in particular, it is located in their second part where functions are complex.

V. CONCLUSION

This work focused on investigating static analyzers detection on Ethereum smart contract dataset. At first, it highlights a strong positives correlation between some software metrics and positives outcomes. Then, the paper investigates where tools can find some classes of vulnerabilities. Results of the work are affected by the dataset composition (randomly made). Future works can extend results by analyzing the true and false positives. For this purpose, it is planned to perform a manual inspection of a subset of smart contracts to create a ground truth to refer to. Moreover, we plan to analyze the overlap of positives to identify common detected faults.

REFERENCES


