Static Analysis Tools Applied to Smart Contracts

Mirko Staderini[1], András Pataricza[2], Andrea Bondavalli[1]
[1]Department of Mathematics and Informatics, University of Florence, Florence, Italy
[2]Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary
{mirko.staderini, andrea.bondavalli}@unifi.it, pataricza.andras@vik.bme.hu

***Abstract*— Smart contracts are one of the most important innovations of the second generation of the Blockchain. They are widely used in various contexts, including financial, insurance, gaming, and betting. Once a smart contract is deployed on a Blockchain, due to its code immutability, residual vulnerabilities cannot be patched. Static analysis is an efficient method for vulnerability detection. This paper addresses the security evaluation and improvement of Solidity smart contracts through the use of static analysis tools, discussing: (i) the motivation and background, (ii) the evaluation of how good the tools are for improving security, (iii) their combination, and (iv) main results.**

Keywords—security, vulnerability, smart contracts, Solidity, static analysis.

I. Motivation and Background

The *Blockchain* [1] improves the security, especially the immutability, integrity, and non-repudiability of data in distributed systems. *Smart contracts* [2] execute computerized transactions automatically while maintaining a synchronous system state. Any design and coding flaws can lead to exploitable weaknesses in the code. Therefore, security flaws in the code must be detected before the deployment into the Blockchain. The importance is driven by the fact that software is immutable, so it is crucial to identify vulnerabilities at the early stage in the life cycle of smart contracts. Thus, vulnerabilities in smart contracts can lead to severe consequences as the financial losses caused by the DAO attack [3] that drained contract funds by tens of millions of euros.

Ethereum is one of the most widely used platforms for smart contracts, and *Solidity* is the leading Turing-complete programming language to develop Ethereum smart contracts. Smart contracts are executed within the *Ethereum Virtual Machine* (EVM), and each execution modifies the EVM *state* through the exchange of signed packets between users (*transactions*). Both transactions and the EVM state are stored in the Blockchain.

The novelty and rapid growth of Solidity result in the scarcity of vulnerability records and immaturity of monitoring tools. On the other hand, Solidity is a language influenced by C++, Javascript, and Python; therefore, classical analysis methods can be used to ensure security.

Static analysis is the most widely used method of code inspection because of its low effort requirement; moreover, although defect coverage is incomplete, it is sufficient for most applications [4]. Static analysis inspects code without executing it. After extracting a model (typically an *abstract syntax tree* or *control flow graph*) of the code under analysis, it looks for vulnerable patterns (*antipatterns*) in the code on the model. Several static analyzers that focus explicitly on Solidity smart contracts have been developed in recent years.

Regarding vulnerabilities, the National Vulnerability Database (NVD) has adopted the Common Weakness Enumeration (CWE) to classify vulnerability-related weaknesses. CWE is the most widely used list of weaknesses in software and systems related to security aspects. CWE applies a stepwise abstraction scheme in the form of a hierarchical taxonomy. By referring to language-independent classification and, in particular, abstract general classes [5], it eliminates the separate treatment of virtually the same vulnerabilities that appear in a version-dependent form of syntax.

Below, we propose our methodology to increase the security of Solidity contracts through static analysis tools. After summarizing the main points for the security evaluation in Section II, we show how to improve the security in Section III. Finally, we list the main results in Section IV.

II. Security Evaluation

To assess the smart contract security through static analyzers, we need to identify the set of vulnerabilities to be analyzed, the tools to analyze them, what (the dataset), and how (a set of metrics) to perform the analysis. Finally, we need to perform a performance evaluation of the tests.

A. Vulnerabilities

To provide a list of vulnerabilities, we performed a Google Scholar search using the keywords "*Ethereum survey*," "*smart contracts analysis*," "*smart contracts vulnerabilities*." In addition, we checked the *Smart Contract Weakness Classification and Test Case* (SWC) registry and leading papers (e.g., [6]). Finally, by grouping vulnerabilities with similar or overlapping definitions and focusing on the *Solidity release >= 0.5*, we obtained a list of 33 items. We mapped the vulnerabilities with a CWE-based classification into ten classes according to the method proposed in [5].

B. Tools selection and preliminary analysis

We first identified the tools that analyze Solidity code, then we extracted those that fulfill the following criteria: (i) handling of contracts written in Solidity version 0.5 or up; (ii) stand-alone tools targeting vulnerabilities detection; (iii) analysis of smart contracts without user-defined properties or assertions; (iv) free public availability. At the end of this process, we selected nine *static analysis tools* (SAT): *Securify*, *Securify2*, *Slither*, *Mythril*, *Oyente*, *Osiris*, *HoneyBadger*, *Remix*, *SmartCheck*.

The tools identify software anomalies unevenly and with varying granularity. In addition, their diagnostic messages lack a uniform and comparable form. Thus, through reverse engineering, we (i) identified the applied control rules, (ii) mapped the rules to weaknesses or vulnerabilities, and (iii) estimated the vulnerabilities and classes that remained uncovered.

C. Datasets

To build a *reference dataset* (about 400 smart contracts), we randomly extracted smart contracts (with Solidity release >= 0.5) from *Etherscan* through a Java crawler. By focusing only on defect detection capabilities, we refer to all cases with a defective contract result as *positive*. However, we

cannot know whether a "*positive*" is a *true positive* (TP - correct detection of an existing vulnerability) or *false positive* (FP - incorrect detection of a non-existent vulnerability). Thus, we extracted a subset of contracts from the reference dataset: this constitutes our *pilot set* to manually inspect it and determine a ground truth.

To build the pilot set, 15 smart contracts (with a total number of 9732 lines of codes) were randomly selected from the reference dataset with the following constraints: (i) coverage of each of the five most frequent types in Ethereum: gambling, other games, exchange, finance, property; (ii) representativeness of the main features of the language; (iii) representativeness of vulnerabilities of the reference dataset. For our analysis, we consider vulnerable lines and not total vulnerabilities. A line identified as vulnerable must be manually analyzed regardless of whether it contains one or more vulnerabilities. After finding the first vulnerability with a manual inspection, the probability of finding a second one (if any) is still very high. Our analysis determined that in the pilot set, 86 lines of code out of 1000 were vulnerable.

D. Testing performances

For binary classification, the *confusion matrix* is a 2x2 matrix used to describe the behavior of the classifier. The main diagonal of the matrix identifies the correct classification in terms of TP and *true negatives* (TN) - correct assessment of no vulnerability -. The anti-diagonal matrix contains the false classification: *false negatives* (FN - failure to detect an existing vulnerability) and FP.

The confusion matrix is used to define different performance indicators. Among the most used *Recall* defined as TP / (TP + FN) - the term *coverage* is often used - and *Precision* defined as TP / (TP + FP). F_1 *score* is the harmonic mean between precision and recall. *Balanced accuracy* normalizes the prediction of true negatives and true positives as the average of true positive and negative rates.

We use previous metrics to evaluate the security that each tool can guarantee smart contract processing both globally and for each class in our taxonomy. We first identify how good the tools are, focusing specifically on their *working domain* (how they perform in what they propose to do). Then we evaluate the security of smart contracts when processed by tools; in this case, we consider tools as *black boxes* that analyze a given set of vulnerabilities.

III. Security Improvement

We investigated a way to improve the security of smart contracts by using a *combination* of available tools that increase the coverage (at the price to increase the number of false positives). Then we define the *severity* of false negatives to *prioritize* their analysis.

A. Tools combination

There are several ways to combine tools. We define a TP of the combination as a vulnerable row in which at least one of the tools finds a positive. An FP is a non-vulnerable row in which at least one of the tools finds a positive. The definition of TN and FN follows accordingly. We run our experiments with two, three or four tools, identifying the cost of the combination (increase of FPs, execution in terms of time and additional resources required).

B. Vulnerabilities Prioritization

We observed that there are FNs even using tool combinations. Therefore, we want to determine whether FNs are equally important to each other. We identified two types of scoring suitable for this purpose: i) the typical severity of attack patterns employed in exploiting vulnerabilities - according to the *Common Attack Pattern Enumeration and Classification* (CAPEC); ii) the severity determined through the *Common Weakness Scoring System* (CWSS) developed by the CWE.

Thus, for both CAPEC and CWSS, we first defined the severity of each vulnerability (*critical*, *high*, *medium*, *low*) and classes in our taxonomy. The identified severity does **not** depend on the smart contract sets. Next, our work identified the types (and occurrences) of FNs not covered by the tool combinations; this way, we determine FNs whose analysis should be prioritized. Finally, a comparison between the results of the two scoring systems is performed.

IV. Main Results

Following, the main results obtained so far:

1) Preliminary analysis of tools - no tools detect the following vulnerabilities: *missing protection against replay attacks, malicious library, requirement violation, typecast, insufficient gas griefing*.

2) Tools that are best built (best performance in what they propose to do - the pilot set): considering *overall rating*, *Slither* (in all performance indicators); for the *coverage* of some specific classes *Remix* and *Securify2*.

3) Best performing tools (pilot set): Securify2 for coverage, *Slither* for accuracy; not a meaningful difference between them in combined assessment (F_1 *score* and *balanced accuracy*).

4) Combination of tools (pilot set): Slither-Securify2-SmartCheck-Remix has the highest coverage (0.91).

5) Prioritization of FNs (pilot set): with CAPEC and CWSS, we identified (respectively) 6 types of false negatives with critical or high severity.

Our results show that real-world smart contracts contain many vulnerabilities. We also identified the coverage limits of individual tools. In order to deploy more secure smart contracts, we need to previously process them with a combination of tools and analyze critical FNs.

References

[1] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf. [Accessed: 01-Mar-2021].

[2] N. Szabo, "Formalizing and Securing Relationships on Public Networks," 1997.

[3] Coindesk, "Understanding the DAO Attack," 2016. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists. [Accessed: 21-Jan-2021].

[4] V. Okun, W. F. Guthrie, R. Gaucher, and P. E. Black, "Effect of static analysis tools on software security: Preliminary investigation," in *Proceedings of the ACM QoT* 2007, doi: 10.1145/1314257.1314260.

[5] M. Staderini, C. Palli, and A. Bondavalli, "Classification of Ethereum Vulnerabilities and their Propagations," in *Proceedings of BCCA* 2020, pp. 44–51, doi: 10.1109/BCCA50787.2020.9274458.

[6] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–43, Jul. 2020, doi: 10.1145/3391195.