Journal of Discrete Mathematical Sciences & Cryptography
ISSN 0972-0529 (Print), ISSN 2169-0065 (Online)
Vol. 26 (2023), No. 5, pp. 1499–1517
DOI : 10.47974/JDMSC-1815

An efficient approach to secure smart contract of Ethereum blockchain using hybrid security analysis approach

Satpal Singh Kushwaha[†]
Sandeep Joshi *
Amit Kumar Gupta [§]
Department of Computer Science and Engineering
Manipal University Jaipur
Jaipur
Rajasthan
India

Abstract

The technology behind blockchain is quickly becoming one of the most crucial innovations in recent years. The Smart contracts are digital agreements, made in between two untrusted parties. Smart contracts are self-executable small piece of code that gets executed due to some predefined triggering conditions. Smart contracts store cryptocurrencies as their balances and deal in cryptocurrencies on network transactions. Because of this, smart contracts are constantly open to the possibility of being attacked. A single security vulnerability can make the smart contract very much insecure. The immutability property of the blockchain ensures that, once a smart contract has been placed on the blockchain, cannot be modified in any way. So, the smart contract must be analyzed for any kind of security vulnerability before its deployment on the blockchain. Existing analysis approaches detect vulnerabilities with high false positive rates. Our proposed approach analyses the smart contracts using a hybrid combination of pattern matching and symbolic execution, which produces results with a low false positive rate. We have performed a comparative analysis of our proposed approach to prove its efficiency with the existing research approaches on a data set of 453 smart contracts with tagged vulnerabilities.

Subject Classification: *68M25 Computer security.*

Keywords: *Ethereum, Smart contract, Security, Vulnerability, Hybrid analysis, Decentralized, Blockchain.*

[†] *E-mail:* **singh_satpal25@rediffmail.com**
* *E-mail:* **sjoshinew@yahoo.com (Corresponding Author)**
[§] *E-mail:* **dramitkumargupta1983@gmail.com**

1. Introduction

Blockchain technology can be defined as decentralized, distributed ledger that records transactions in an immutable way. So, once some transaction is deployed on the blockchain then it becomes immutable[2]. It became popular in 2008 when Satoshi Nakamoto's published a research article for the double spending problem's solution. These digital contracts are auto-triggered and get executed when some mutually agreed terms and conditions are met. So, smart contracts auto-execute the mutual agreement without any trusted third party. The trust between parties is made by consensus algorithms. The smart contracts have their states and store cryptocurrencies as their balances. As smart contracts deal in cryptocurrencies, malicious attackers have an interest in the same. These are small pieces of code, so any security issue makes them vulnerable to attack.

Due to the Immutability characteristic of blockchain such vulnerable smart contracts are more insecure. Because the smart contracts can no longer be changed after they have been uploaded to the blockchain and implemented[23][24][25]. So, the smart contracts must be adequately analyzed [9] to ensure all safety properties before deployment on the blockchain. For analyzing such smart contracts, we have proposed an efficient approach based on combined analysis for Ethereum blockchain smart contracts. It employs static rule-based pattern matching using regular expressions combined with the static symbolic execution. The proposed approach accurately detects vulnerabilities and reduces false positives.

1.1 *Motivation*

The limitations of Existing Ethereum smart contracts security analysis approaches are that they are less accurate in terms of security vulnerability detection by producing false positives at a high rate. Existing approaches analyze the smart contract for security vulnerability detection either at the solidity source code level or the EVM byte code level. Still, it is neither sound nor complete, which results in several false alarms. Due to these false alarms, the smart contract is not thoroughly decontaminated from security vulnerabilities which leads to perilous conditions and losses.

1.2 *Contributions*

During this work, we will look for security flaws in Ethereum smart contracts and try to limit the number of false positives so that we can attain a high level of accuracy. These are the contributions that were made:

- In this work, a combined scheme is proposed to analyze the solidity code using pattern matching and EVM byte code using symbolic execution separately.
- It establishes a set of regular expressions for vulnerable patterns to analyze solidity source code.
- It establishes a set of 453 vulnerable smart contracts with tagged vulnerabilities.
- The experimental results show high accuracy with a low false positive rate.

1.3 *Outline*

The following outline constitutes the paper's structure: In section 2, we will review a summary of the many research methodologies that have been taken to analyze Ethereum smart contracts. In section 3, we discussed our proposed approach. In section 4, we discussed evaluation and comparison results with existing approaches. At last, Section 5 concluded the whole paper and suggested future research directions.

2. Related Work

Since the inception of the Ethereum blockchain, several approaches have been proposed and developed. **Luu L. et. al.**[10] suggested a symbolic execution bases dynamic analysis scheme analyzing the Ethereum smart contracts. The authors named the approach "Oyente". They proposed an approach that symbolically analyzes the solidity of smart contracts path by path. **Tikhomirov S. et. al.**[12] suggested a static analysis scheme that performs pattern matching with XPath patterns. First, it gets XML-based intermediate representation translated from the Solidity source code. Then it performs pattern matching using XP path patterns on XML-based intermediate representation to detect vulnerable security. **Chen T. et. al.**[4] suggested a static analysis scheme. It accepts EVM byte code as input, for analyzing it to find gas-consuming patterns. They distinguish seven gas-intensive patterns into two groups: loop-related patterns and patterns related to unnecessary code. In essence, it is a gas-intensive pattern-checking approach for symbolic execution that uses byte code. **Zhou E. et. al.** [14]suggested a static analysis plan with two key components. A Relationship Analysis of Invocation B Logic Risk Expansion and Location To identify potential logic concerns, it provides a topology map showing the relationship between invocations. Together with syntax analysis, they

pinpoint logic vulnerabilities to functions. **Chang J. et. al.**[3] suggested an approach to automatically identify critical software routes that involve financial transactions as crucial paths and prioritize those that may violate important properties to uncover security concerns was presented as the basis for a static analysis scheme. **Feist J. et. al.** [5]proposed a static analysis method that analyses solidity code as input. They made use of SlithIR, their internal intermediate representation. To find vulnerabilities, it constructs a control flow graph, analyses data flow, and tracks taint. **Kalra S. et. al.** [8]suggested a static analysis method that uses symbolic execution and inputs the solidity smart contract and the policies that the smart contract must pass must be suggested. Then, at the appropriate program locations, it inserts policy predicates as assertions. The code that this policy asserts is then translated into LLVM bytecode. Finally, the verifier looks for violations of policy. **Norvill R. et. al.** [11]presented a static analysis method that displays the smart contract's simulated execution on the Ethereum Virtual Machine. It runs the smart contract's byte code, displaying the current operational code, stack state, and control flow at each level of the program's execution. **Akca S. et. al.**[1] suggested a three-phase static analysis method for finding vulnerabilities. Instrumentation with assertion via SIF (Phase 1). Phase 2). Creation of input for smart contracts with instruments. Phase 3: Instrumented contract execution and analysis. **Grech N. et. al.** [7]suggested a method for performing static analysis that makes use of Gigahose IR. With the use of a lifter called Gigahose IR, low-level EVM bytecode can be transformed into some high-level intermediate representations. The method primarily finds problems that are gas related.

3. Proposed Approach

The proposed approach uses a static examination of the solidity source code as its foundation for Ethereum smart contract. Dynamic analysis requires the execution of the code to detect security vulnerabilities, but some coding errors or security issues may not be detected using dynamic analysis. But the static code analysis can cover all program paths without executing the program's source code. The proposed approach is a hybrid combination of pattern matching with static symbolic execution. Figure 1 depicts the architecture and algorithm 1 depicts the step-by-step workflow of the proposed approach. The proposed approach is divided into three modules, whose working is as follows:

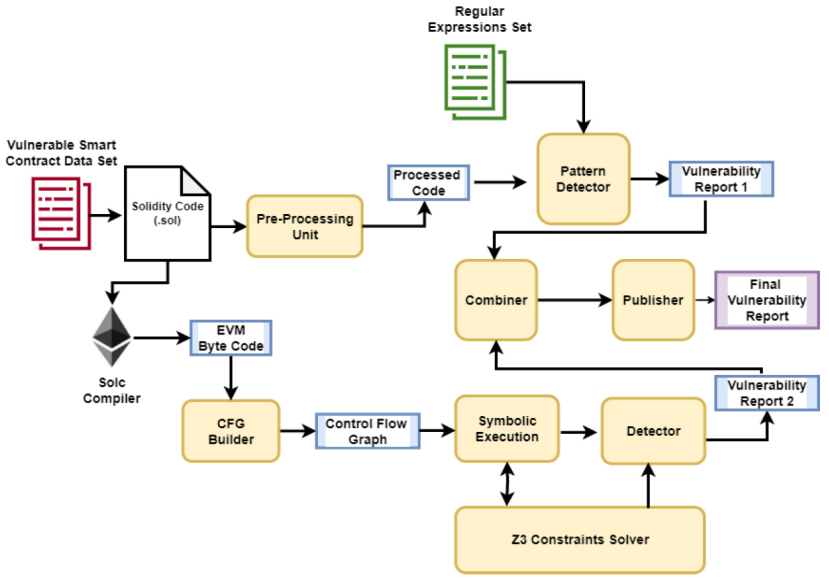

Figure 1

Proposed approach architecture

3.1 *Module 1*

This module analysis starts with some prepossessing following pattern detection. This module directly works on the solidity code to perform pattern matching using regular expressions to detect vulnerable patterns. Regular expressions can be used very efficiently to detect such patterns. For this purpose, we have created a set of regular expressions for vulnerable patterns. The regular expression patterns are created and edited using Regex Generator[15] and tested for accuracy on Pythex[16].

Algorithm 1: Proposed Approach

Input: α-Ethereum smart contract data set, β- Regular expression set

Output: Vulnerability analysis report

1. BEGIN
2. Set $Vul\_R1[] = \phi$
3. Set $Vul\_R2[] = \phi$
4. for each solidity smart contract $P_{Sol} \in \alpha$

   $\rho \leftarrow$ Pre-Processor (P_{Sol})

for each regular expression in β
Vul ← Pattern_Detector (ρ, β)
If (*Vul* != ϕ)
Vul_R1[]← *Vul*
end if
end for
end for

5. for each smart contract $P_{Sol} \in \alpha$
   EVM_{BC} ← SolC_Compiler (P_{Sol})
   CFG ←– CFG_Builder (EVM_{BC})
   Vul ←– Symbolic_Execution_Engine (*CFG*)
   if (*Vul* != ϕ)
   Vul_R2[] ←– *Vul*
   end if
   end for
6. *Vul_R3[]* ←– Combiner(*Vul_R1[] , Vul_R2[])*
7. Publisher(*Vul_R3[]*)
8. END

The working of Pre-Processing unit and Pattern detector is as follows:

- **Pre-Processing Unit**

  The Solidity source code format is not appropriate for matching patterns directly. So, we need to do some pre-processing on the solidity source code. Solidity code is converted into a processed string that does not contain newline characters, multiple spaces, single lines, or double-line comments. To do so, all of these are replaced by a single space character. Now, this processed string is ready to use for pattern matching in Pattern Detector.

- **Pattern Detector**

  The pattern detector takes the input of processed solidity code and the set of regular expressions. We have provided an identifier for each vulnerable pattern according to Smart Contract Weakness Classification Registry (SWC)[17]. The Pattern Detector produces

the security vulnerability report, which faded into the combiner of module 3.

Following is a detailed description of some famous vulnerabilities along with their drafted regular expressions for pattern matching.

3.1.1 Reentrancy Vulnerability

SWC-107 identifier is provided for this vulnerability in Smart Contract Weakness Classification Registry. A reentrancy vulnerability might occur when a malicious contract is called. When one contract communicates with another contract that is located outside of its environment. The called contract has the potential to assume control of the communicating contract and make unauthorized changes to its state. Since the detection of the same using pattern matching leads to false results, we are unable to identify a definitive pattern for reentrancy vulnerability in terms of smart contracts. Simple and direct patterns may result in false positives and precise patterns may result in false negatives. The vulnerable pattern for reentrancy vulnerability may contain 'call.value' to transfer ether following the change in the smart contract state. Table 1 shows the regular expression used to detect reentrancy vulnerability

Table 1

Regular expressions to detect patterns of reentrancy vulnerability

Solidity Text pattern	Regular Expression
require(msg.sender.call.value(amount) -());credit[msg.sender] =amount;	^(require||if)+\([a-zA-Z]+\.[a-zA-Z]+\.call+\.value+\([a-zA-Z]+\)\(\)\);[a-zA-Z]+\[[a-zA-Z]+\.[a-zA-Z]+]-=[a-zA-Z]+;.*
require(msg.sender.call.value(amount) ()); credit[msg.sender]=credit[msg.sender]-amount;	^(require||if)+\([a-zA-Z]+\.[a-zA-Z]+\.call+\.value+\([a-zAZ]+\)\(\)\);[a-zA-Z]+\[[a-zA-Z]+\.[a-zA-Z]+]=[a-zA-Z]+\[[azA-Z]+\.[a-zA-Z]+]-[a-zA-Z]+;.*

3.1.2 Integer Overflow/Underflow

SWC-101 identifier is provided for this vulnerability in Smart Contract Weakness Classification Registry. This vulnerability is related to arithmetic operations. During integer calculation, it may be possible that

Table 2

Regular expressions to detect patterns related to Integer/Overflow

Solidity Text pattern	Regular Expression
vname-=1; OR vname+=1; OR vname=vname-1; OR vname=vname+1; OR vname++; vname–; OR vname*=var; OR vname=vname*var;	^([a-zA-Z]+(*|\ +|\-)=[0-9azA-Z]+;)|([a-zA-Z]+=[a-zAZ]+(*|\+|\-)+[0-9a-zA-Z]+;)|([a-zA-Z]+((**)|(\+\+)|(\-\-))+;)$

the calculated value may exceed the size limit of the variable size. Due to this, a certain variable will not be able to hold the computed value, resulting in wrong calculations. A definite pattern cannot be possible for this issue. The pattern detects this issue in the statements with arithmetic operations like addition, subtraction, division, or multiplication. Table 2 shows the regular expression used to detect Overflow/Underflow vulnerability.

3.1.2 Tx.origin

SWC-115 identifier is provided for this vulnerability in Smart Contract Weakness Classification Registry. When the "tx.origin" variable is used for authorization, it is quite simple for the adversary to determine the initial address of the person who started the transaction. That address can be used to drain the initiator's account. A vulnerable smart contract can have the presence of the "tx.origin == owner" or "tx.origin != owner". Table 3 shows the regular expression used to detect "tx.origin" vulnerable patterns.

Table 3

Regular expressions to detect "tx.origin" patterns

Solidity Text pattern	Regular Expression
require(tx.origin == owner);	^[a-zA-Z]+$tx.origin + + == [a - zA - Z]+;$

3.1.4 Weak Sources of Randomness from chain attributes:

SWC-116 identifier is provided for this vulnerability in Smart Contract Weakness Classification Registry. It is another severe vulnerability due to the use of block timestamps. When a block timestamp is used as a triggering condition for critical operation execution like a monetary transaction from one contract to another or can be used as a source of the

Table 4

Regular expressions to detect weak sources of randomness

Solidity Text pattern	Regular Expression
block.difficulty()	^block+\.difficulty+(\(\)|\s)$
block.gaslimit()	^block+\.gaslimit+(\(\)|\s)$
block.coinbase()	^block+\.coinbase+(\(\)|\s)$
block.number()	^block+\.number+(\(\)|\s)$

random number, then it can be misused by some malicious minor. Since the minors have the right from smart contracts to put a timestamp within 10 seconds of block validation. So, to divert the contract execution result in his favor, the minor can modify the timestamp by a few seconds, resulting in the wrong outcome. Table 4 shows the regular expression used to detect bad randomness using block timestamp.

3.1.5 Self-Destruct / Suicidal contract

SWC-106 identifier is provided for this vulnerability in Smart Contract Weakness Classification Registry. We cannot delete a smart contract from the Ethereum blockchain, but we can delete the code and storage of the same by using a special function known as "Self-Destruct". On the execution of the "Self-Destruct" by a smart contract, its ether balance is transferred to a specified address, and the code and storage of the same are deleted forever. Then no one can interact with that smart contract. But this feature can be used by a malicious attacker to drain all the ether balance of the target smart contract by creating a contract with a self-destruct function. Table 5 shows the regular expression used to detect "self-destruct/suicidal contracts.

Table 5

Regular expressions to detect "Self-Destruct Suicidal" patterns

Solidity Text pattern	Regular Expression
require(tx.origin == owner);	^[a-zA-Z]+$tx.origin + + == [a - zA - Z]+;$

3.2 *Module 2*

This module analyzes smart contracts using symbolic execution. Symbolic execution is the way of analyzing a program by constructing

```
pragma solidity 0.4.24;

contract SDAO {
  mapping (address => uint) public credit;

  function donateto(address to) payable public{
    credit[to] += msg.value;
  }

  function withdrawal(uint amount) public{
    if (credit[msg.sender]>= amount) {
      require(msg.sender.call.value(amount)());
      credit[msg.sender]-=amount;
    }
  }

  function qCredit(address to) view public returns(uint){
    return credit[to];
  }
}
```

Figure 2

SDAO smart contract solidity code

formulas that represent the program states at different locations or points in program paths. The symbolic execution simply explores all the program paths without executing the program. This module takes the input of EVM byte code generated by the SolC-compiler. The EVM byte code is faded into the CFG builder component to create the control flow graph. Critical paths of the control flow graph are symbolically executed for constraint generation, which is solved using the Z3 constraint solver for path satisfiability. This information is used by the detector to detect vulnerabilities and mapping with SWC id, which are finally faded into the module 3 combiner for further processing. Figure 2 depict the solidity code of a smart contract named SDAO[6] and its respective EVM byte code.

The following are module 2 components:

Control Flow Graph extraction: It is a pictorial representation of the computation or control flow of the program during execution. Flow inside the program can be accurately represented by CFG. In a program, a basic block is the simplest unit of control flow or a sequence of operations without any exception or branch-free code. A CFG is a Digraph G = (N, E), where a node n ∈ N and an edge e ∈ (n_a, n_b) ∈ E. Figure 3 depicts an instance of CFG of figure 2 smart contract. CFG construction involves the following steps:

a) Extraction of EVM byte code from the Solidity code.

b) Extraction of Opcode sequence from EVM byte code.

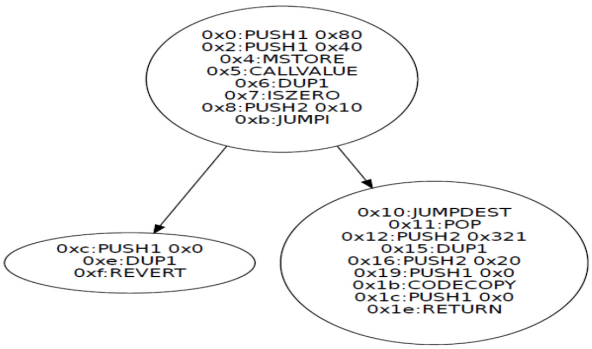

Figure 3

An instance of CFG of SDAO smart contract

c) Extraction of basic blocks of CFG from opcode sequence. Identification of basic block boundaries using special opcodes instructions like JUMP, JUMPI, JUMPDEST, CALL, REV ERT, RETURN, and STOP. Connection establishment using direct jumps, between basic blocks with edges. In the case of indirect jumps use stack simulation to complete CFG.

Symbolic Execution: This component explores the paths of the control flow graph for constraint generation for each path. The Z3 constraint solver is used to check the satisfiability of each path condition. The satisfiability results are used by the detector component to detect the vulnerabilities.

Detection logic for vulnerable paths: A path is considered vulnerable if certain properties are violated. EVM contains special instructions to modify contracts stated in the Ethereum smart contract. Based on these special instructions, we have defined these properties or detection logic for the vulnerabilities discussed in module 1.

- **Reentrancy:** Smart contracts are on target of malicious attackers because it stores tokens or ether as their balances. Solidity has some special instructions which are involved in ether transfer or modifying the state of the smart contract. A malicious attacker can utilize these special instructions to perform an attack or steal ether. If, after an external call, the state of the contract is modified using the special instructions, then the such contract is tagged vulnerable for reentrancy vulnerability.

- **Integer Overflow/Underflow:** Integer overflow and real underflow issue is a little complex to detect because sometimes they can be created by a compiler or developer intentionally for some functionality or to implement some logic. So, we can predict only the possibility of the same. EVM opcodes that can cause integer overflow and underflow are SUB, ADD, MUL, and EXP. If, after this operation, any state change is detected or some unauthorized operation after a conditional branch is detected, then the contract is tagged vulnerable for integer overflow/underflow vulnerability.
- **Self-Destruct/Suicidal contract:** The execution of Self-destruct instruction can be triggered by any sender. If a path is traced with Self-destruct instruction and the transaction is not initiated by the owner of the contract, then the contract is tagged vulnerable Self-destruct Suicidal contract.
- **Weak sources of randomness from chain attributes:** If block details like block number, block gas limit, block difficulty, block timestamp, or block coin-base are detected in a path. Then if any control flow change is detected after the execution of these statements, such a smart contract is tagged vulnerable for predictable variable dependency.
- **Tx.origin:** The use of tx.origin can lead to a drastic situation where a smart contract authorizes another smart contract or user to carry out activity on its behalf. So, the flow decisions can be influenced by the tx.origin variable. We check for control flow change after the use of tx.origin in a path. If a path contains such type of behavior, then it is considered an unauthorized control transfer, and the contract is tagged vulnerable due to tx.origin.

3.3 *Module 3*

This module contains the following two components:

Combiner: This component receives vulnerability reports from module 1 and module 2 in the form of SWC id. The major problem in the literature review is false positives produced by existing approaches. To overcome this, we assume that if both the modules detect the same vulnerability, then only the same vulnerability will be added to the final vulnerability report, otherwise discarded. This approach ensures true positive results.

Publisher: This component publishes the vulnerabilities by mapping them with SWC ID.

4. Evaluation and comparison

We created a data set of 453 vulnerable Ethereum smart contract solidity source codes with 1143 tagged vulnerabilities, out of which 213 are downloaded from SolidiFI benchmark[18], SB Curated[19], Smart-Bugs[20], and 240 were downloaded from Ether-scan API[21]. Figure 4 depicts the share of each vulnerability in the created data set. The evaluation is performed on two aspects of the proposed approach, the first is vulnerability detection, and the second is less false positive. Existing approaches and tools are mainly focused on vulnerability detection only. For comparison purposes, we selected existing approaches Securify[13], Mythril[22], Oyente[10], Slither[5], and SmartCheck[12].

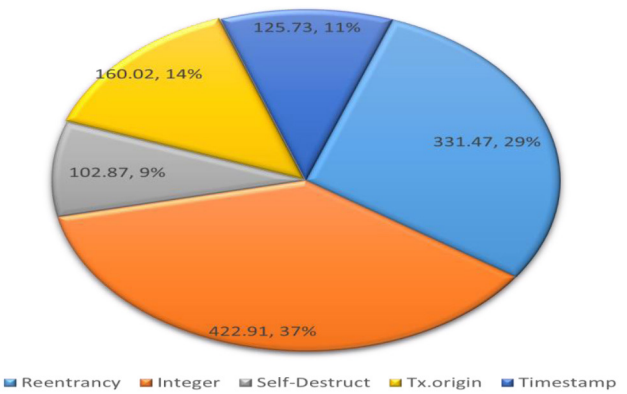

Figure 4

Vulnerability share in created Dataset of 453 smart contracts

The selection of these tools is based on two aspects. The first aspect is these tools are mostly used in all survey research articles and newly proposed tools research articles. The second aspect is these tools detect mostly all the vulnerabilities discussed to be detected in the proposed work. We have focused on five vulnerabilities discussed in our work for comparison. Figures 5 to 9 show the comparison results of the proposed work and existing approaches on the created Dataset. The results show that the proposed approach is efficient as compared to the existing approaches and tools in terms of vulnerability detection. Figure 10 shows the detection

capability comparison of the proposed work with existing tools and approaches, and the proposed approach detects 84% of vulnerabilities in the created dataset. Securify detects 65% of vulnerabilities, which is low compared to others.

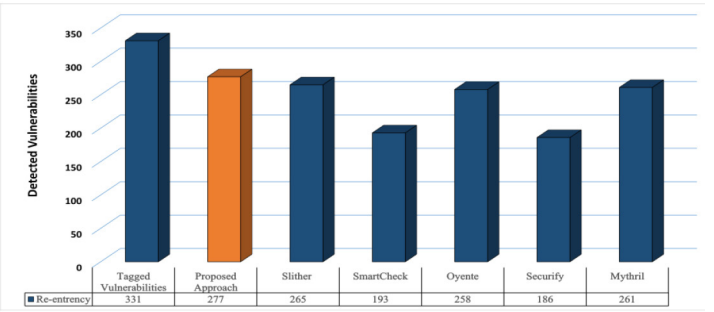

Figure 5

Comparison results for "Reentrancy" vulnerability

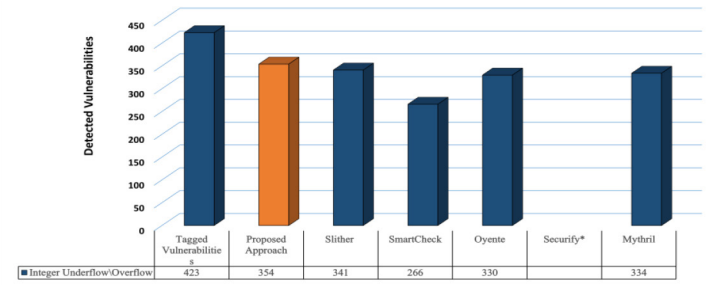

Figure 6

Comparison results for "Integer Overflow/Underflow" vulnerability

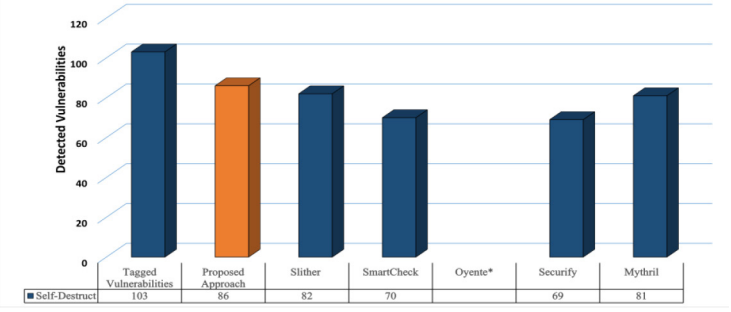

Figure 7

Comparison results for "Self-Destruct" vulnerability

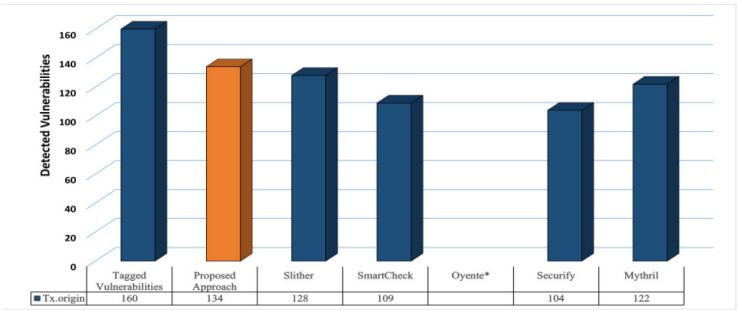

Figure 8

Comparison results for "Tx.origin" vulnerability

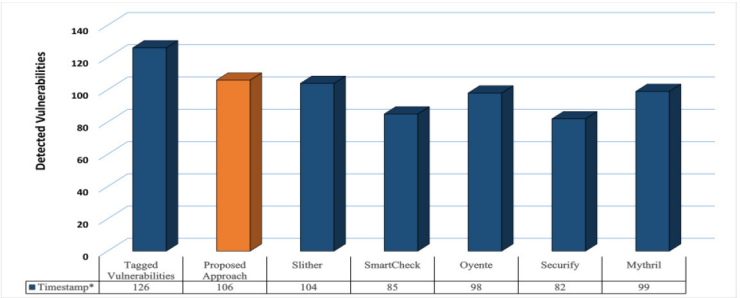

Figure 9

Comparison results for "Weak sources of randomness"

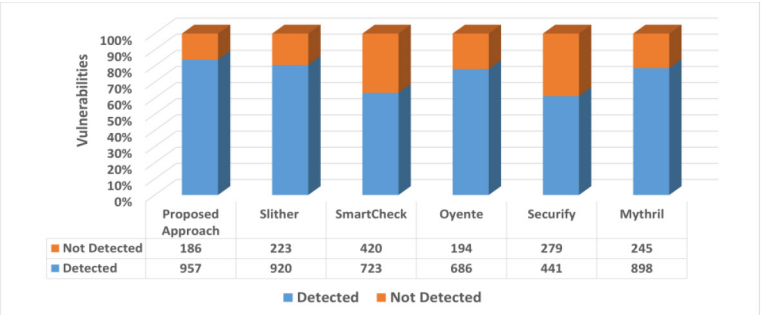

Figure 10

Comparison results for detected and not detected vulnerabilities

The false positive production rate of the existing approaches is high because they mainly focus on the maximum detection and coverability of security vulnerabilities. Figure 11 shows the comparison results of

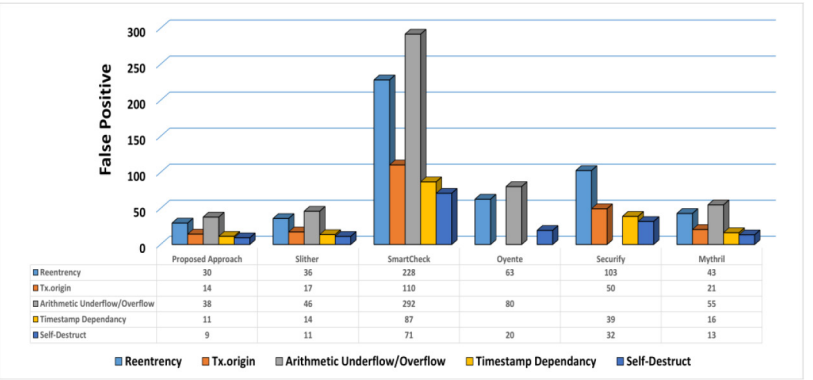

Figure 11

Comparison results for False Positive

the false positive rate of the proposed work with existing approaches. The proposed approach produces fewer false positives as compared to others. False positive produced by SmartCheck and Securify is high. Both the tools produce most false positives in reentrancy vulnerabilities. Our experiments show that the proposed approach's false positive rate is better than Slither and Mythril.

5. Discussion and Conclusion

This paper proposes an efficient hybrid approach to analyzing Ethereum blockchain smart contracts for security issues. The proposed approach leverages pattern matching and symbolic execution. For evaluation purposes, we created a data set of 453 vulnerable Ethereum smart contracts. We found that the proposed approach outperforms in terms of accuracy on the created data set. The proposed approach reports accurate results with fewer false positives as compared to existing approaches. As of now, this work focused on five vulnerabilities, which are mostly involved in attacks on Ethereum smart contracts. The proposed approach can be improved by including all the unresolved security vulnerabilities, adding new patterns, and fully automating the proposed work. We tested the proposed work's ability to detect bugs by contrasting its performance on a variety of security issues with that of other state-of-the-art technologies that were readily accessible at the time. In comparison to the other tools, we discovered that the proposed work is superior in terms of its performance, robustness, and accuracy.

References

[1] Sefa Akca, Ajitha Rajan, and Chao Peng. SolAnalyser: A Framework for Analysing and Testing Smart Contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 482-489 (2019). DOI:https://doi.org/10.1109/APSEC48747.2019.00071.

[2] A. Averin and O. Averina. Review of Blockchain Technology Vulnerabilities and Blockchain-System Attacks. In *2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*, IEEE, 1-6 (2019). DOI:https://doi.org/10.1109/FarEastCon.2019.8934243.

[3] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, Yan Cai, and Zijiang Yang. sCompile: Critical Path Identification and Analysis for Smart Contracts. 286-304 (2019). DOI:https://doi.org/10.1007/978-3-030-32409-4_18.

[4] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiaopu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts. *IEEE Trans Emerg Top Comput* 9, 3 (July 2021), 1433-1448 (2021). DOI:https://doi.org/10.1109/TETC.2020.2979019.

[5] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, IEEE, 8-15 (2019). DOI:https://doi.org/10.1109/WETSEB.2019.00008.

[6] Baraq Ghaleb, Ahmed Al-Dubai, Elias Ekonomou, Mamoun Qasem, Imed Romdhani, and Lewis Mackenzie. Addressing the DAO Insider Attack in RPL's Internet of Things Networks. *IEEE Communications Letters* 23, 1 (January 2019), 68-71 (2019). DOI:https://doi.org/10.1109/LCOMM.2018.2878151.

[7] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (October 2018), 1-27 (2018). DOI:https://doi.org/10.1145/3276486.

[8] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings 2018 Network and*

Distributed System Security Symposium, Internet Society, Reston, VA. (2018). DOI:https://doi.org/10.14722/ndss.2018.23082.

[9] Gupta S, Bairwa AK, Kushwaha SS, Joshi S. Decentralized Identity Management System using the amalgamation of Blockchain Technology. In2023 3rd International Conference on Intelligent Communication and Computational Techniques (ICCT), pp. 1-6 (2023 Jan 19). IEEE.

[10] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, New York, NY, USA, 254-269 (2016). DOI:https://doi.org/10.1145/2976749.2978309.

[11] Robert Norvill, Beltran Borja Fiz Pontiveros, Radu State, and Andrea Cullen. Visual emulation for Ethereum's virtual machine. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, IEEE, 1-4 (2018). DOI:https://doi.org/10.1109/NOMS.2018.8406332.

[12] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, ACM, New York, NY, USA, 9-16. DOI:https://doi.org/10.1145/3194113.3194115.

[13] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, New York, NY, USA, 67-82 (2018). DOI:https://doi.org/10.1145/3243734.3243780.

[14] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yoshihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. Security Assurance for Smart Contract. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, IEEE, 1-5 (2018). DOI:https://doi.org/10.1109/NTMS.2018.8328743.

[15] Regex Generator. Retrieved September 5, 2022, from https://regex-generator.olafneumann.org/.

[16] Pythex. Retrieved November 5, 2022, from https://pythex.org/.

[17] SWC Registry: Smart Contract Weakness Classification and Test Cases. Retrieved October 5, 2022, from https://swcregistry.io/.

[18] SolidiFI-Benchmark. Retrieved October 5, 2022, from https://github.com/smartbugs/SolidiFI-benchmark.

[19] SB Curated: A Curated Dataset of Vulnerable Solidity Smart Contracts n.d. Retrieved September 5, 2022, from https://github.com/smartbugs/smartbugs/blob/master/dataset.

[20] SmartBugs Wild Dataset n.d. Retrieved October 5, 2022, from https://github.com/smartbugs/smartbugs-wild.

[21] Etherscan: The Ethereum Blockchain Explorer n.d. Retrieved November 5, 2022, from https://etherscan.io.

[22] Mythril. Retrieved September 5, 2022, from https://github.com/ConsenSys/mythril.

[23] Amit Kumar Gupta, Pushpa Gothwal, Dinesh Goyal & Carlos M. Travieso-Gonzalez. IoT-Galvanized pandemic special E-Toilet for generation of sanitized environment, *Journal of Discrete Mathematical Sciences and Cryptography* (2022), DOI: 10.1080/09720529.2022.2068607.

[24] Amit Kumar Gupta, Vijander Singh, Priya Mathur & Carlos M. Travieso-Gonzalez. Prediction of COVID-19 pandemic measuring criteria using support vector machine, prophet and linear regression models in Indian scenario, *Journal of Interdisciplinary Mathematics* (2020), DOI:https://doi.org/10.1080/09720502.2020.1833458.

[25] Anil Kumar, Ravinder Kumar, Sartaj Singh Sodhi. A novel privacy preserving blockchain based secure storage framework for electronic health records. *Journal of Information and Optimization Sciences* 43:3, pages 549-570 (2022).